2019-08-30

1 Logistics

CS 6210 is a graduate-level introduction to numerical linear algebra. We will study direct and iterative methods for linear systems, least squares problems, eigenvalue problems, and singular value problems. There is a detailed syllabus and rough schedule at the class web:

http://www.cs.cornell.edu/courses/cs6210/2019fa/

The web page is also your source for the homework, lecture notes, and course announcements. The web page also points to a discussion forum (Piazza) and the Course Management System (CMS) used for homework submissions. There is also a link to a GitHub repository with the source for these notes. If you find an error, feel free to help me fix it!

I will assume that you have already had a course in linear algebra and that you know how to program. A previous numerical methods course is not required. The course will involve programming in MATLAB or Julia, so it will also be helpful – but not strictly necessary – for you to have prior exposure to one of these languages. If you know neither language but you have prior programming experience, you can learn them quickly enough. Octave is a viable substitute for MATLAB for the work in this course.

2 Linear algebra references

We assume some mathematical pre-requisites: linear algebra and "sufficient mathematical maturity." This should be enough for most students, but some will want to brush up on their linear algebra. For those who want to review, or who want a reference while taking the course, I have a few recommendations.

There are a few options when it comes to linear algebra basics. At Cornell, our undergraduate linear algebra course uses the text by Lay [4]; the texts by Strang [5, 6] are a nice alternative. Strang's *Introduction to Linear Algebra* [6] is the textbook for the MIT linear algebra course that is the basis for his enormously popular video lectures, available on MIT's OpenCourseWare site; if you prefer lecture to reading, Strang is known as an excellent lecturer.

These notes reflect the way that I teach matrix computations, but this is far from the only approach. The style of this class is probably closest to

the style of the text by Demmel [1], which I recommend as a course text. The text of Trefethen and Bau [7] covers roughly the same material, but with a different emphasis; I recommend it for an alternative perspective. At Cornell, our subscription to the SIAM book series means that both of these books are available as e-books to students on the campus network.

For students who intend to use matrix computations as a serious part of their future research career, the canonical reference is *Matrix Computations* by Golub and Van Loan [3].

3 Basic notational conventions

In this section, we set out some basic notational conventions used in the class.

1. The complex unit is i (not i or j).
2. By default, all spaces in this class are finite dimensional. If there is only one space and the dimension is not otherwise stated, we use n to denote the dimension.
3. Concrete real and complex vector spaces are $\mathbb{R}^n$ and $\mathbb{C}^n$, respectively.
4. Real and complex matrix spaces are $\mathbb{R}^{m\times n}$ and $\mathbb{C}^{m\times n}$.
5. Unless otherwise stated, a concrete vector is a column vector.
6. The vector e is the vector of all ones.
7. The vector e_i has all zeros except a one in the ith place.
8. The basis $\{e_i\}_{i=1}^n$ is the *standard basis* in $\mathbb{R}^n$ or $\mathbb{C}^n$.
9. We use calligraphic math caps for abstract space, e.g. $\mathcal{U}, \mathcal{V}, \mathcal{W}$.
10. When we say U is a basis for a space $\mathcal{U}$, we mean U is an isomorphism $\mathcal{U} \to \mathbb{R}^n$. By a slight abuse of notation, we say U is a matrix whose columns are the abstract vectors $u_1, \ldots, u_n$, and we write the linear combination $\sum_{i=1}^n u_i c_i$ concisely as Uc.
11. Similarly, $U^{-1}x$ represents the linear mapping from the abstract vector x to a concrete coefficient vector c such that $x = Uc$.

12. The space of univariate polynomials of degree at most d is $\mathcal{P}_d$.

13. Scalars will typically be lower case Greek, e.g. α, β. In some cases, we will also use lower case Roman letters, e.g. c, d.

14. Vectors (concrete or abstract) are denoted by lower case Roman, e.g. x, y, z.

15. Matrices and linear maps are both denoted by upper case Roman, e.g. A, B, C.

16. For $A \in \mathbb{R}^{m \times n}$, we denote the entry in row i and column j by a_{ij}. We reserve the notation A_{ij} to refer to a submatrix at block row i and block column j in a partitioning of A.

17. We use a superscript star to denote dual spaces and dual vectors; that is, $v^* \in \mathcal{V}^*$ is a dual vector in the space dual to $\mathcal{V}$.

18. In $\mathbb{R}^n$, we use x^* and x^T interchangeably for the transpose.

19. In $\mathbb{C}^n$, we use x^* and x^H interchangeably for the conjugate transpose.

20. Inner products are denoted by angles, e.g. $\langle x, y \rangle$. To denote an alternate inner product, we use subscripts, e.g. $\langle x, y \rangle_M = y^* M x$.

21. The standard inner product in $\mathbb{R}^n$ or $\mathbb{C}^n$ is also $x \cdot y$.

22. In abstract vector spaces with a standard inner product, we use v^* to denote the dual vector associated with v through the inner product, i.e. $v^* = (w \mapsto \langle w, v \rangle)$.

23. We use the notation $\|x\|$ to denote a norm of the vector x. As with inner products, we use subscripts to distinguish between multiple norms. When dealing with two generic norms, we will sometimes use the notation $|||y|||$ to distinguish the second norm from the first.

24. We use order notation for both algorithm scaling with parameters going to infinity (e.g. $O(n^3)$ time) and for reasoning about scaling with parameters going to zero (e.g. $O(\epsilon^2)$ error). We will rely on context to distinguish between the two.

25. We use *variational notation* to denote derivatives of matrix expressions, e.g. $\delta(AB) = \delta A \, B + A \, \delta B$ where δA and δB represent infinitesimal changes to the matrices A and B.

26. Symbols typeset in Courier font should be interpreted as MATLAB code or pseudocode, e.g. `y = A*x`.

27. The function notation $\mathrm{fl}(x)$ refers to taking a real or complex quantity (scalar or vector) and representing each entry in floating point.

4 Matrix algebra versus linear algebra

> We share a philosophy about linear algebra: we think basis-free, we write basis-free, but when the chips are down we close the office door and compute with matrices like fury.
>
> — Irving Kaplansky on the late Paul Halmos [2],

Linear algebra is fundamentally about the structure of vector spaces and linear maps between them. A matrix represents a linear map with respect to some bases. Properties of the underlying linear map may be more or less obvious via the matrix representation associated with a particular basis, and much of matrix computations is about finding the right basis (or bases) to make the properties of some linear map obvious. We also care about finding changes of basis that are "nice" for numerical work.

In some cases, we care not only about the linear map a matrix represents, but about the matrix itself. For example, the *graph* associated with a matrix $A \in \mathbb{R}^{n \times n}$ has vertices $\{1, \ldots, n\}$ and an edge (i, j) if $a_{ij} \neq 0$. Many of the matrices we encounter in this class are special because of the structure of the associated graph, which we usually interpret as the "shape" of a matrix (diagonal, tridiagonal, upper triangular, etc). This structure is a property of the matrix, and not the underlying linear transformation; change the bases in an arbitrary way, and the graph changes completely. But identifying and using special graph structures or matrix shapes is key to building efficient numerical methods for all the major problems in numerical linear algebra.

In writing, we represent a matrix concretely as an array of numbers. Inside the computer, a *dense* matrix representation is a two-dimensional array data structure, usually ordered row-by-row or column-by-column in order to accomodate the one-dimensional structure of computer memory address spaces. While much of our work in the class will involve dense matrix layouts, it is important to realize that there are other data structures! The "best" representation for a matrix depends on the structure of the matrix and on what we want to do with it. For example, many of the algorithms we will

discuss later in the course only require a black box function to multiply an (abstract) matrix by a vector.

5 Dense matrix basics

There is one common data structure for dense vectors: we store the vector as a sequential array of memory cells. In contrast, there are *two* common data structures for general dense matrices. In MATLAB (and Fortran), matrices are stored in *column-major* form. For example, an array of the first four positive integers interpreted as a two-by-two column major matrix represents the matrix

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}.$$

The same array, when interpreted as a *row-major* matrix, represents

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Unless otherwise stated, we will assume all dense matrices are represented in column-major form for this class. As we will see, this has some concrete effects on the efficiency of different types of algorithms.

5.1 The BLAS

The *Basic Linear Algebra Subroutines* (BLAS) are a standard library interface for manipulating dense vectors and matrices. There are three *levels* of BLAS routines:

- **Level 1**: These routines act on vectors, and include operations such scaling and dot products. For vectors of length n, they take $O(n^1)$ time.
- **Level 2:** These routines act on a matrix and a vector, and include operations such as matrix-vector multiplication and solution of triangular systems of equations by back-substitution. For $n \times n$ matrices and length n vectors, they take $O(n^2)$ time.
- **Level 3:** These routines act on pairs of matrices, and include operations such as matrix-matrix multiplication. For $n \times n$ matrices, they take $O(n^3)$ time.

All of the BLAS routines are superficially equivalent to algorithms that can be written with a few lines of code involving one, two, or three nested loops (depending on the level of the routine). Indeed, except for some refinements involving error checking and scaling for numerical stability, the reference BLAS implementations involve nothing more than these basic loop nests. But this simplicity is deceptive — a surprising amount of work goes into producing high performance implementations.

5.2 Locality and memory

When we analyze algorithms, we often reason about their complexity abstractly, in terms of the scaling of the number of operations required as a function of problem size. In numerical algorithms, we typically measure *flops* (short for floating point operations). For example, consider the loop to compute the dot product of two vectors:

```
function [result] = mydot(x,y)
  n = length(x);
  result = 0;
  for i = 1:n
    result = result + x(i)*y(i); % two flops/iteration
  end
```

Because it takes n additions and n multiplications, we say this code takes $2n$ flops, or (a little more crudely) $O(n)$ flops.

On modern machines, though, counting flops is at best a crude way to reason about how run times scale with problem size. This is because in many computations, the time to do arithmetic is dominated by the time to fetch the data into the processor! A detailed discussion of modern memory architectures is beyond the scope of these notes, but there are at least two basic facts that everyone working with matrix computations should know:

- Memories are optimized for access patterns with *spatial locality*: it is faster to access entries of memory that are close to each other (ideally in sequential order) than to access memory entries that are far apart. Beyond the memory system, sequential access patterns are good for *vectorization*, i.e. for scheduling work to be done in parallel on the vector arithmetic units that are present on essentially all modern processors.

- Memories are optimized for access patterns with *temporal locality*; that is, it is much faster to access a small amount of data repeatedly than

to access large amounts of data.

The main mechanism for optimizing access patterns with temporal locality is a system of *caches*, fast and (relatively) small memories that can be accessed more quickly (i.e. with lower latency) than the main memory. To effectively use the cache, it is helpful if the *working set* (memory that is repeatedly accessed) is smaller than the cache size. For level 1 and 2 BLAS routines, the amount of work is proportional to the amount of memory used, and so it is difficult to take advantage of the cache. On the other hand, level 3 BLAS routines do $O(n^3)$ work with $O(n^2)$ data, and so it is possible for a clever level 3 BLAS implementation to effectively use the cache.

5.3 Matrix-vector multiply

Let us start with a very simple MATLAB program for matrix-vector multiplication:

```
function y = matvec1(A,x)
  % Form y = A*x (version 1)

  [m,n] = size(A);
  y = zeros(m,1);
  for i = 1:m
    for j = 1:n
      y(i) = y(i) + A(i,j)*x(j);
    end
  end
```

We could just as well have switched the order of the i and j loops to give us a column-oriented rather than row-oriented version of the algorithm. Let's consider these two variants, written more compactly:

```
function y = matvec2_row(A,x)
  % Form y = A*x (row-oriented)

  [m,n] = size(A);
  y = zeros(m,1);
  for i = 1:m
    y(i) = A(i,:)*x;
  end
```

```
function y = matvec2_col(A,x)
  % Form y = A*x (column-oriented)

```

```
  [m,n] = size(A);
  y = zeros(m,1);
  for j = 1:n
    y = y + A(:,j)*x(j);
  end
```

It's not too surprising that the builtin matrix-vector multiply routine in MATLAB runs faster than either of our `matvec2` variants, but there are some other surprises lurking. Try timing each of these matrix-vector multiply methods for random square matrices of size 4095, 4096, and 4097, and see what happens. Note that you will want to run each code many times so that you don't get lots of measurement noise from finite timer granularity; for example, try

```
tic;        % Start timer
for i = 1:100 % Do enough trials that it takes some time
  % ...       Run experiment here
end
toc         % Stop timer
```

On my machine (a late MacBook Pro 13 inch, late 2013 model) and using MATLAB 2016a, we show the GFlop rates (billions of flops/second) for the three matrix multiply routines in Figure 5.3. There are a few things to notice:

- The performance of the built-in multiply far exceeds that of any of the manual implementations.
- The peak performance occurs for moderate size matrices where the matrix fits into cache, but there is enough work to hide the MATLAB loop overheads.
- The time required for the built-in routine varies dramatically (due to so-called *conflict misses*) when the dimension is a multiple of a large integer power of two.
- For $n = 1024$, the column-oriented version (which has good spatial locality) is $10\times$ faster than the row-oriented code, and $45\times$ faster than the two nested loop version.

If you are so inclined, consider yourself encouraged to repeat the experiment using your favorite compiled language to see if any of the general trends change significantly.

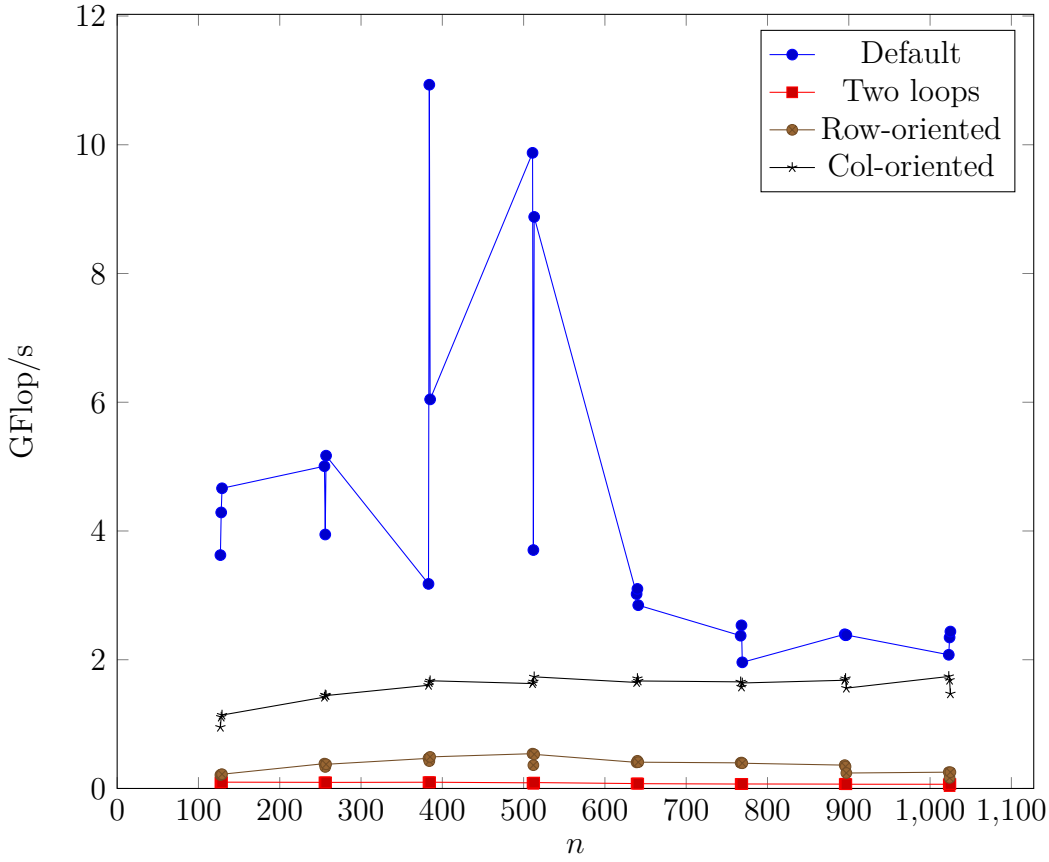

Figure 1: Timing of four matrix-vector multiply implementations. In each case, we report the effective time in GFLop/s. The line labeled "default" is the built-in MATLAB matvec.

5.4 Matrix-matrix multiply

The classic algorithm to compute $C := C + AB$ involves three nested loops

```
function [C] = matmul_3loop(A,B)
  % Three nested loops version of matrix multiply (ijk order)

  [m,p1] = size(A);
  [p2,n] = size(C);
  assert(p1 == p2, 'Dimension mismatch');
  C = zeros(m,n);

  for i = 1:m
    for j = 1:n
      for k = 1:p
        C(i,j) = C(i,j) + A(i,k)*B(k,j);
      end
    end
  end
```

This is sometimes called an *inner product* variant of the algorithm, because the innermost loop is computing a dot product between a row of A and a column of B. But addition is commutative and associative, so we can sum the terms in a matrix-matrix product in any order and get the same result. And we can interpret the orders! A non-exhaustive list is:

- `ij(k)` or `ji(k)`: Compute entry c_{ij} as a product of row i from A and column j from B (the *inner product* formulation)
- `k(ij)`: C is a sum of outer products of column k of A and row k of B for k from 1 to n (the *outer product* formulation)
- `i(jk)` or `i(kj)`: Each row of C is a row of A multiplied by B
- `j(ik)` or `j(ki)`: Each column of C is A multiplied by a column of B

At this point, we could write down all possible loop orderings and run a timing experiment, similar to what we did with matrix-vector multiplication. But the truth is that high-performance matrix-matrix multiplication routines use another access pattern altogether, involving more than three nested loops, and we will describe this now.

5.5 Blocking and performance

The basic matrix multiply outlined in the previous section will usually be at least an order of magnitude slower than a well-tuned matrix multiplication routine. There are several reasons for this lack of performance, but one of the most important is that the basic algorithm makes poor use of the *cache*. Modern chips can perform floating point arithmetic operations much more quickly than they can fetch data from memory; and the way that the basic algorithm is organized, we spend most of our time reading from memory rather than actually doing useful computations. Caches are organized to take advantage of *spatial locality*, or use of adjacent memory locations in a short period of program execution; and *temporal locality*, or re-use of the same memory location in a short period of program execution. The basic matrix multiply organizations don't do well with either of these. A better organization would let us move some data into the cache and then do a lot of arithmetic with that data. The key idea behind this better organization is *blocking*.

When we looked at the inner product and outer product organizations in the previous sections, we really were thinking about partitioning A and B into rows and columns, respectively. For the inner product algorithm, we wrote A in terms of rows and B in terms of columns

$$\begin{bmatrix} a_{1,:} \\ a_{2,:} \\ \vdots \\ a_{m,:} \end{bmatrix} \begin{bmatrix} b_{:,1} & b_{:,2} & \cdots & b_{:,n} \end{bmatrix},$$

and for the outer product algorithm, we wrote A in terms of colums and B in terms of rows

$$\begin{bmatrix} a_{:,1} & a_{:,2} & \cdots & a_{:,p} \end{bmatrix} \begin{bmatrix} b_{1,:} \\ b_{2,:} \\ \vdots \\ b_{p,:} \end{bmatrix}.$$

More generally, though, we can think of writing A and B as *block matrices*:

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1,p_b} \\ A_{21} & A_{22} & \dots & A_{2,p_b} \\ \vdots & \vdots & & \vdots \\ A_{m_b,1} & A_{m_b,2} & \dots & A_{m_b,p_b} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1,p_b} \\ B_{21} & B_{22} & \dots & B_{2,p_b} \\ \vdots & \vdots & & \vdots \\ B_{p_b,1} & B_{p_b,2} & \dots & B_{p_b,n_b} \end{bmatrix}$$

where the matrices A_{ij} and B_{jk} are compatible for matrix multiplication. Then we we can write the submatrices of C in terms of the submatrices of A and B

$$C_{ij} = \sum_k A_{ij} B_{jk}.$$

5.6 The lazy man's approach to performance

An algorithm like matrix multiplication seems simple, but there is a lot under the hood of a tuned implementation, much of which has to do with the organization of memory. We often get the best "bang for our buck" by taking the time to formulate our algorithms in block terms, so that we can spend most of our computation inside someone else's well-tuned matrix multiply routine (or something similar). There are several implementations of the Basic Linear Algebra Subroutines (BLAS), including some implementations provided by hardware vendors and some automatically generated by tools like ATLAS. The best BLAS library varies from platform to platform, but by using a good BLAS library and writing routines that spend a lot of time in *level 3* BLAS operations (operations that perform $O(n^3)$ computation on $O(n^2)$ data and can thus potentially get good cache re-use), we can hope to build linear algebra codes that get good performance across many platforms.

This is also a good reason to use MATLAB: it uses pretty good BLAS libraries, and so you can often get surprisingly good performance from it for the types of linear algebraic computations we will pursue.

References

[1] James Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[2] John Ewing and F. W. Gehring, editors. *Paul Halmos: Celebrating 50 Years of Mathematics*. Springer, 1991.

[3] Gene Golub and Charles Van Loan. *Matrix Computations*. Johns Hopkins University Press, fourth edition, 2013.

[4] David Lay, Steven Lay, and Judi McDonald. *Linear Algebra and its Applications*. Pearson, fifth edition, 2016.

[5] Gilbert Strang. *Linear Algebra and its Applications*. Brooks/Cole Publishing, fourth edition, 2006.

[6] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, fourth edition, 2009.

[7] Lloyd N Trefethen and David Bau, III. *Numerical Linear Algebra*. SIAM, 1997.