

Week 1: Wednesday, Aug 28

Logistics

1. We will go over the syllabus in the first class, but in general you should look at the class web page at

<http://www.cs.cornell.edu/~bindel/class/cs6210-f13/>

The web page is your source for the syllabus, homework, lecture notes and readings, and course announcements.

2. For materials that are private to the class (e.g. grades and solutions), we will use the Course Management System:

<http://cms.csuglab.cornell.edu/web/guest/>

If you are taking the class for credit – or even if you think you may take the class for credit – please sign the list and include your Cornell netid. That way, I can add you to the Course Management System.

3. Please read Golub and Van Loan, 1.1-1.5. You may also want to start on the first homework (due Sep 9).

Course Overview

CS 6210 is a graduate-level introduction to numerical linear algebra. The required text is Golub and Van Loan, *Matrix Computations* (4e), and I will assign readings and some problems from this text. As supplementary reading, you may want to consider Trefethen and Bau, *Numerical Linear Algebra* and Demmel, *Applied Numerical Linear Algebra*. Organized by topics, the course looks something like this:

1. Background (Ch 1–2)
2. Linear systems and Gaussian elimination (Ch 3–4)
3. Least squares problems (Ch 5–6)

4. Unsymmetric eigenvalue problems (Ch 7)
5. Symmetric eigenvalue problems (Ch 8)
6. Iterative methods (Ch 10–11)

We could also organize the course by four over-arching themes:

1. *Algebra*: matrix factorizations and such
2. *Analysis*: perturbation theory, convergence, error analysis, etc.
3. *Algorithms*: efficient methods for modern machines.
4. *Applications*: coming from all over

The interplay between these aspects of the subject is a big part of what makes the area so interesting! The point of the next few lectures will be to introduce these themes in the context of two very simple operations: multiplication of a matrix by a vector, and multiplication of two matrices

I will assume that you have already had a course in linear algebra and that you know how to program. It will be helpful if you've had a previous numerical methods course, though maybe not strictly necessary; talk to me if you feel nervous on this point. The course will involve MATLAB programming, so it will also be helpful – but not strictly necessary – for you to have prior exposure to MATLAB. If you don't know MATLAB already but you have prior programming experience, you can learn MATLAB quickly enough.

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

```

Basic matrix-matrix multiply

The classic algorithm to compute $C := C + AB$ is

```

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 . We can express this concisely in MATLAB as

```

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

```

There are also *outer product* variants of the algorithm that put the loop over the index k on the outside, and thus computing C in terms of a sum of outer products:

```

for k = 1:p
  C = C + A(:,k)*B(k,:);
end

```

One can also think of a matrix-matrix multiply as a sequence of matrix vector multiplies, either by columns or by rows

```

% Column-by-column
for j = 1:n
  C(:,j) = C(:,j) + A*B(:,j);
end
% Row-by-row
for i = 1:m
  C(i,:) = C(i,:) + A(i,:)*B;
end

```

Organization and performance

What are the implications of the different ways of organizing matrix multiplication? Let's compare the time taken to run the different versions of the multiply on a pair of square matrices of dimension 510 through 515.

	510	511	512	513	514	515
Standard	6.67716	7.47196	7.34847	7.44501	7.80189	7.8066
Dot product	1626.19	1659.33	2078.02	1642.69	1637.36	1707.38
Outer product	260.018	706.812	692.756	737.445	698.42	720.05
Column-oriented	23.1434	25.7763	22.6276	25.1109	24.7495	26.3315
Row-oriented	26.5856	27.8777	31.892	27.5754	26.1164	27.1533

The difference is huge!

Blocking and performance

As we have seen, 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

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

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

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

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

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

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

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

$$C_{ij} = \sum_k A_{ik} B_{kj}.$$

A Timing matrix multiply

The MATLAB script used to produce the timing table will appear on the web page. It's worth looking at it to see how to do this sort of thing. In particular, note that each experiment is run several times in order to avoid issues with the timer granularity.