Notes for 2016-01-27

What are we about?

Welcome to "Numerical Analysis: Linear and Nonlinear Equations" (CS 4220, CS 5223, and Math 4260). This is part of a pair of courses offered jointly between CS and math that provide an introduction to scientific computing. My own tongue-in-cheek summary of scientific computing is that it is the art of solving problems of continuous mathematics fast enough and accurately enough. Of course, what constitutes "fast enough" and "accurately enough" depends on context, and learning to reason about that context is also part of the class.

Because our survey is partitioned into two semesters, we do not cover all the standard topics in a single semester. In particular, this class will (mostly) not cover interpolation and function approximation, numerical differentiation and quadrature, or the solution of ordinary and partial differential equations. We will focus instead on numerical linear algebra, nonlinear equation solving, and optimization. Broadly speaking, we will spend the first half of the semester on *factorization* methods for linear algebra problems, and the latter half on *iterative* methods for both linear and nonlinear problems. As currently planned, the schedule also includes time for one special topic exercise on randomized methods in linear algebra that I expect will bring together several of the themes from the course.

Mathematics, Computation, Application

Our focus will be the mathematical and computational structure of numerical methods. But we use numerical methods to solve problems from applications, and a scientific computing class with no applications is far less rich and interesting than it ought to be. So we will, when possible, try to bring in application examples.

The majority of students in the class come from computer science. We also have students from various disciplines in the mathematical sciences (math, applied math, statistics, operations research), physical sciences (physics, applied physics, astronomy), engineering disciplines (mechanical, civil, electrical, and chemical engineering), and a few others (economics, undeclared). This means that students come to the class with different levels of background and interest in a variety of application domains. Because of the nature of the enrollment, many of my examples will come from areas conventionally associated with computer science and mathematics, but there will also be the odd example from physics or engineering. So if we dig into an application problem and you get lost, don't worry – I don't expect you to know this already! Also, ask questions, as there are bound to be others the class who are equally confused.

Cross-cutting themes

There are some themes that cut across topics in the syllabus, and I expect we will touch on these themes frequently through the semester. These include:

- **Knowing the answer in advance** – It's dangerous to go into a computation with no idea what to expect. The structure of the problem and the solution affect how we choose methods and how we evaluate success. A qualitative analysis or ballpark estimate of solution behavior is usually the first step to intelligently applying a numerical method.
- **Pictures and plots** – Careful pictures tell us a lot. Plot an approximate solution. Are there unexpected oscillations or negative values, or crazy-looking behaviors near the domain of the soution? Maybe you should investigate! Similarly, plots of error with respect to a spatial variable or a step number often provide key insights into whether a method is working as desired.
- **Documentation, testing, and error checking** – When we write numerical codes, the implied agreement between the author of the code and the user of the code is often more subtle than the agreements behind other software interfaces. Call a sort routine, and it will sort your data in some specified time. Call a linear solver, and it will solve your problem in an amount of time that depends on the problem structure and with a level of accuracy that depends on the problem characteristics. This makes good software hygiene – careful documentation, testing, error checking, and design for reproducibility – both tricky and important!
- **Modularity and composability** – When we compose numerical methods, we have to worry about error. Even if you expect only to use numerical building blocks (and never build them yourself), it is important

to understand the types of error and performance guarantees one can make and how they are useful in reasoning about large computational codes.

- **Problem formulation and choice of representation** – Often, the same problem can be posed in many different ways. Some suggest simple, efficient numerical methods. Others are impossibly hard. The key difference between the two is often in how we represent the problem data and the thing we seek.

- **Numerical anti-patterns** – Some operations, such as computing explicit inverses and determinants, are perfectly natural in symbolic mathematics but turn out to be terrible ideas in numerical computations. We will point these out as we come across them.

- **Time and memory scalability** – We often want to solve big problems, and it is important to understand before we start whether we think we can solve a problem on a laptop in a second or two or if we really need a month on a supercomputer. This means we would like a rough estimate – usually posed in terms of order notation – of the time and memory complexity of different algorithms.

- **Blocking and building with high-performance blocks** – Building fast codes is hard. As numerical problem solvers, we would like someone else to do much of this hard work so that we can focus on other things. This means we need to understand the common building blocks, and a little bit about not only their complexity, but also why they are fast or slow on real machines.

- **Performance tradeoffs in iterations** – Iterative methods produce a sequence of approximate solutions that (one hopes) get closer and closer to the right answer. To choose iterations intelligently, we need to understand the tradeoffs between the time to compute an iteration, the progress that one can make, and the overall stability of an iterative procedure.

- **Convergence monitoring and stopping** – One of the hardest parts of designing an iterative method is often deciding when to stop. This point will recur several times in the second half of the semester.

- **Use of approximations and surrogates** – Simple surrogate models are an important part of the design of nonlinear iterations. We will be particularly interested in local polynomial approximations, but we may talk about some others as well.

Logistics

We will go through the syllabus in detail, but at a high level you should plan on six homeworks (individual) and three projects (in pairs), a midterm, and a final. I will also ask you for feedback at the middle and end of the semester, and this counts for credit. I will give "problems of the day" to help study, but we will not use these directly to grade you.

Homework and projects are due via CMS by midnight on Fridays; we allow some "slip days" so that you can work on an assignment through the weekend if needed. We have office hours scheduled before class on Wednesday, 10-11 Thursday, and 10-12 on Friday. You can also request office hours by appointment.

Infrastructure

Class notes and assignments, as well as class announcements, will be posted on the course home page. For submissions, solutions, and grades, we will use the CS Course Management System (CMS) software. For class discussion, we will use Piazza. There are links from each of these pages to the others; I recommend you use the class web page as your starting point.

We will use MATLAB in our notes, but programming assignments may be done in MATLAB (or Octave) or in Python.

The course web page is maintained from a repository on GitHub. I encourage you to submit corrections or enhancements by pull request!

Background

The formal prerequisites for the class are linear algebra at the level of Math 2210 or 2940 or equivalent and a CS 1 course in any language. We also recommend one additional math course at the 3000 level or above; this is essentially a proxy for "sufficient mathematical maturity."

In practice: I will assume you know some multivariable calculus and linear algebra, and that your CS background includes not only basic programming but also some associated mathematical concepts (e.g. order notation and a little graph theory). If you feel your background is weak in these areas, please talk to us.

Matrix algebra versus linear algebra

1. Matrices are extremely useful. So are linear transformations. But note that matrices and linear transformations are *different* things! Matrices *represent* finite-dimensional linear transformations with respect to particular bases. Change the bases, and you change the matrix, if not the underlying operator. Much of the class will be about finding the right basis to make some property of the underlying transformation obvious, and about finding changes of basis that are "nice" for numerical work.

2. A linear transformation may correspond to different matrices depending on the choice of basis, but that doesn't mean the linear transformation is always the thing. For some applications, the matrix itself has meaning, and the associated linear operator is secondary. For example, if I look at an adjacency matrix for a graph, I probably really do care about the matrix – not just the linear transformation.

3. Sometimes, we can apply a linear transformation even when we don't have an explicit matrix. For example, suppose $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and I want to compute $\partial F/\partial v|_{x_0} = (\nabla F(x_0)) \cdot v$. Even without an explicit matrix for ∇F, I can compute $\partial F/\partial v|_{x_0} \approx F(x_0 + hv) - F(x_0))/h$. There are many other linear transformations, too, for which it is more convenient to apply the transformations than to write down the matrix – using the FFT for the Fourier transform operator, for example, or fast multipole methods for relating charges to potentials in an n-body electrostatic interaction.

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
```

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

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

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.

Problems to ponder

Unless otherwise stated, assume $A, B \in \mathbb{R}^{n \times n}$ (square real $n \times n$ matrices), u, v, x, y are vectors in $\mathbb{R}^n$, and $D = \operatorname{diag}(d)$ is a diagonal $n \times n$.

1. Describe the effect of pre- and post-multiplying A by D; that is, what are DA and AD?
2. How many floating point operations are needed to evaluate the following (assuming ordinary order of operations)?
   (a) $(uv^T)A$
   (b) $u(v^T A)$
   (c) $A(uv^T)B$
   (d) $(Au)(v^T V)$
   (e) ADx
   (f) $A(Dx)$
3. Describe a brief snippet of MATLAB code to form the most efficient versions of the above expressions.
4. The standard tridiagonal matrix $T_N \in \mathbb{R}^{N \times N}$ acts on the vector u in the following way:
   $$(Tu)_i = -u_{i-1} + 2u_i - u_{i+1}$$
   with the convention $u_0 = u_{N+1} = 0$.
   (a) What is T_5, written explicitly?
   (b) Write a MATLAB snippet to evaluate Tu in $O(N)$ time.
5. Let $E \in \mathbb{R}^{n \times n}$ be the matrix of all ones. Describe an $O(n)$ approach to compute Ev.
6. The operation $\operatorname{triu}(E)$ takes the upper triangular part of E; for example, for $n = 3$, we have
   $$\operatorname{triu}(E) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$
   In general, describe an $O(n)$ approach to compute $\operatorname{triu}(E)v$.