1 Introduction

CS 4220/5210/Math 4260 teaches numerical methods for linear algebra, nonlinear equation solving, and optimization. We used some examples from differential equations and function approximation, but these topics are mostly left to CS 4210/Math 4250. Ideally, you should now understand some basic methods, well enough to choose methods suited to your problems (and vice-versa). You should also know how to tweak the standard strategies for particular types of problems.

Like most of CS[1], numerical methods come in layers. For example, solving a large, difficult optimization problem or nonlinear systems is likely to involve

- A *continuation strategy* to "sneak up" on the hard problem by solving a sequence of easier nonlinear problems.
- A *Newton iteration* (or a related iteration) to solve each of these nonlinear problems by repeatedly approximating them with linear ones.
- A *Krylov subspace iteration* to solve the large linear systems of equations that appear during Newton iteration.
- A *sparse matrix factorization* to solve a *preconditioner* problem that approximates the solution of the large linear systems without the cost of solving them directly.
- And *dense matrix factorizations* that solve dense blocks that appear in the course of factoring the sparse matrix.

Each layer involves choices about methods, parameters, or termination criteria. These are guided by a mix of general error analysis, an understanding of complexity of operations, and problem-specific concerns. Putting together a full solver stack like this is beyond the scope of a first course, but we have seen all these ingredients this semester. Through projects, and some of the special topics at the end of the semester, we have also seen how these ideas come together.

The point of these review notes is not to supplant earlier lecture notes or the book, but to give a retrospective survey of ground we've covered in the past couple months. This is also an excuse for me – and you! – to think about the types of problems I might ask on a final exam.

[1] And, according to Shrek, like ogres and onions.

2 Overview

The meat of this class is *factorizations* and *iterations*.

Factorization involves rewriting a matrix as a product of matrices with some special structure; the big examples from the course are:

$PA = LU$	LU factorization / Gaussian elimination
$A = R^T R$	Cholesky
$A = QR$	QR factorization
$A = U\Sigma V^T$	Singular Value Decomposition (SVD)
$A = UTU^T$	Schur factorization

We discussed the computation of the first three factorizations in enough detail to implement something, and touched more lightly on decompositions for the SVD and Schur factorization. These factorizations provide an efficient way to solve linear systems and least squares problems, but can also be used for various other purposes, from determinants to data compression.

We use iterations to solve nonlinear problems, and even for large linear problems where factorizations are too expensive. The chief building block is *fixed point iterations* of the form

$$x_{k+1} = G(x_k).$$

The most important fixed point iteration is Newton's iteration, which plays a central role in nonlinear equation solving and optimization. Though Newton on its own only converges locally, and though Newton steps may be too expensive, the Newton framework gives us a way of reasoning about a variety of iterations. Fixed point iterations (stationary iterations) for linear systems, such as Jacobi and Gauss-Seidel iteration, are also an important building block for preconditioning modern Krylov subspace iterations such as GMRES and CG.

When we solve a problem numerically, we care about getting the answer fast enough and right enough. To understand the "fast enough" part, we need to understand the cost of computing and using factorizations and the rate of convergence of iterations. To understand the "right enough" part, we need to understand how errors are introduced into a numerical computation through input error, roundoff, or termination of iterations, and how those errors propogate. Our standard strategy is to relate forward error to the

backward error or residual error (which can often be bounded in the context of a particular algorithm or termination criterion) via a condition number (which depends on the problem). The key tools here are Taylor expansion (usually just to first order) and matrix and vector norm bounds.

3 Background

I assume intro courses in calculus and linear algebra, enough programming coursework to write and debug simple MATLAB scripts, and the magical "sufficient mathematical maturity." But people forget things, and some of the background needed for numerics isn't always taught in intro courses. So here are some things you should know that you might not remember from earlier work.

3.1 Linear algebra background

In what follows, as in most of what I've done in class, I will mostly stick with real vector spaces.

Vectors You should know a vector as:

- An object that can be scaled or added to other vectors.
- A column of numbers, often stored sequentially in computer memory.

We often map between the two pictures using a basis. For example, a basis for the vector space of quadratic polynomials in one variable is $\{1, x, x^2\}$; using this basis, we might concretely represent a polynomial $1 + x^2/2$ in computer memory using the coefficient vector

$$c = \begin{bmatrix} 1 \\ 0 \\ 0.5 \end{bmatrix}.$$

In numerical linear algebra, we use column vectors more often than row vectors, but both are important. A row vector defines a linear function over column vectors of the same length. For example, in our polynomial example, suppose we want the row vector corresponding to evaluation at -1. With

respect to the power basis $\{1, x, x^2\}$ for the polynomial space, that would give us the row vector

$$w^T = \begin{bmatrix} 1 & -1 & 1 \end{bmatrix}$$

Note that if $p(x) = 1 + x^2/2$, then

$$p(-1) = 1 + (-1)^2/2 = w^T c = \begin{bmatrix} 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0.5 \end{bmatrix}.$$

Vector norms and inner products A *norm* $\|\cdot\|$ measures vector lengths. It is positive definite, homogeneous, and sub-additive:

$$\begin{aligned} \|v\| &\geq 0 \text{ and } \|v\| = 0 \text{ iff } v = 0 \\ \|\alpha v\| &= |\alpha| \|v\| \\ \|u + v\| &\leq \|u\| + \|v\|. \end{aligned}$$

The three most common vector norms we work with are the Euclidean norm (aka the 2-norm), the ∞-norm (or max norm), and the 1-norm:

$$\begin{aligned} \|v\|_2 &= \sqrt{\sum_j |v_j|^2} \\ \|v\|_\infty &= \max_j |v_j| \\ \|v\|_1 &= \sum_j |v_j| \end{aligned}$$

Many other norms can be related to one of these three norms.

An *inner product* $\langle \cdot, \cdot \rangle$ is a function from two vectors into the real numbers (or complex numbers for an complex vector space). It is positive definite, linear in the first slot, and symmetric (or Hermitian in the case of complex vectors); that is:

$$\begin{aligned} \langle v, v \rangle &\geq 0 \text{ and } \langle v, v \rangle = 0 \text{ iff } v = 0 \\ \langle \alpha u, w \rangle &= \alpha \langle u, w \rangle \text{ and } \langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle \\ \langle u, v \rangle &= \overline{\langle v, u \rangle}, \end{aligned}$$

where the overbar in the latter case corresponds to complex conjugation. Every inner product defines a corresponding norm

$$\|v\| = \sqrt{\langle v, v \rangle}$$

The inner product and the associated norm satisfy the *Cauchy-Schwarz* inequality

$$\langle u, v \rangle \leq \|u\|\|v\|.$$

The *standard inner product* on $\mathbb{R}^n$ is

$$x \cdot y = y^T x = \sum_{j=1}^{n} y_j x_j.$$

But the standard inner product is not the only inner product, just as the standard Euclidean norm is not the only norm.

Matrices You should know a matrix as:

- A representation of a linear map
- An array of numbers, often stored sequentially in memory.

A matrix can *also* represent a bilinear function mapping two vectors into the real numbers (or complex numbers for complex vector spaces):

$$(v, w) \mapsto w^T A v.$$

Symmetric matrices also represent *quadratic forms* mapping vectors to real numbers

$$\phi(v) = v^T A v$$

We say a symmetric matrix A is *positive definite* if the corresponding quadratic form is positive definite, i.e.

$$v^T A v \geq 0 \text{ with equality iff } v = 0.$$

Many "rookie mistakes" in linear algebra involve forgetting ways in which matrices differ from scalars:

- Not all matrices are square.
- Not all matrices are invertible (even nonzero matrices can be singular).
- Matrix multiplication is associative, but not commutative.

Don't forget these facts!

Block matrices We often partition matrices into submatrices of different sizes. For example, we might write

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ c_1 & c_2 & d \end{bmatrix} = \begin{bmatrix} A & b \\ c^T & d \end{bmatrix}, \text{ where } A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, c = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

We can manipulate block matrices in much the same way we manipulate ordinary matrices; we just need to remember that matrix multiplication does not commute.

Matrix norms The matrices of a given size form a vector space, and we can define a norm for such a vector space the same way we would for any other vector space. Usually, though, we want matrix norms that are compatible with vector space norms (a "submultiplicative norm"), i.e. something that guarantees

$$\|Av\| \leq \|A\|\|v\|$$

The most common choice is to use an *operator norm*:

$$\|A\| \equiv \sup_{\|v\|=1} \|Av\|.$$

The operator 1-norm and ∞ norm are easy to compute

$$\|A\|_1 = \max_j \sum_i |a_{ij}|$$

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|$$

The operator 2-norm is theoretically useful, but not so easily computed.

In addition to the operator norms, the *Frobenius norm* is a common matrix norm choice:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

Matrix structure We considered many types of *structure* for matrices this semester. Some of these structures are what I think of as "linear algebra structures," such as symmetry, skew symmetry, orthogonality, or low rank. These are properties that reflect behaviors of an operator or quadratic

form that don't depend on the specific basis for the vector space (or spaces) involved. On the other hand, matrices with special nonzero structure – triangular, diagonal, banded, Hessenberg, or sparse – tend to lose those properties under any but a very special change of basis. But these nonzero structures or matrix "shapes" are very important computationally.

Matrix products Consider the matrix-vector product

$$y = Ax$$

You probably first learned to compute this matrix product with

$$y_i = \sum_j a_{ij} x_j.$$

But there are different ways to organize the sum depending on how we want to think of the product. We could say that y_i is the product of row i of A (written $A_{i,:}$) with x; or we could say that y is a linear combination of the columns of A, with coefficients given by the elements of x. Similarly, consider the matrix product

$$C = AB.$$

You probably first learned to compute this matrix product with

$$c_{ij} = \sum_k a_{ik} b_{kj}.$$

But we can group and re-order each of these sums in different ways, each of which gives us a different way of thinking about matrix products:

$$\begin{aligned}
C_{ij} &= A_{i,:} B_{:,j} && \text{(inner product)} \\
C_{i,:} &= A_{i,:} B && \text{(row-by-row)} \\
C_{:,j} &= A B_{:,j} && \text{(column-by-column)} \\
C &= \sum_k A_{:,k} B_{k,:} && \text{(outer product)}
\end{aligned}$$

One can also think of organizing matrix multiplication around a partitioning of the matrices into sub-blocks. Indeed, this is how tuned matrix multiplication libraries are organized.

Fast matrix products There are some types of matrices for which we can compute matrix-vector products very quickly. For example, if D is a diagonal matrix, then we can compute Dx with one multiply operation per element of x. Similarly, if $A = uv^T$ is a rank-one matrix, we can compute Ax quickly by recognizing that matrix multiplication is associative

$$Ax = (uv^T)x = u(v^T x).$$

Thus, we can apply A with one dot product (between v and x) and a scaling operation.

Singular values and eigenvalues A square matrix A has an eigenvalue λ and corresponding eigenvector $v \neq 0$ if

$$Av = \lambda v.$$

A matrix is *diagonalizable* if it has a complete basis of eigenvectors $v_1, \ldots, v_n$; in this case, we write the *eigendecomposition*

$$AV = V\Lambda$$

where $V = \begin{bmatrix} v_1 & \ldots & v_n \end{bmatrix}$ and $\Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$. If a matrix is not diagonalizable, we cannot write the eigendecomposition in this form (we need Jordan blocks and generalized eigenvectors). In general, even if the matrix A is real and diagonalizable, we may need to consider complex eigenvalues and eigenvectors.

A real *symmetric* matrix is always diagonalizable with real eigenvalues, and has an orthonormal basis of eigenvectors $q_1, \ldots, q_n$, so that we can write the eigendecomposition

$$A = Q\Lambda Q^T.$$

For a nonsymmetric (and possibly rectangular) matrix, the natural decomposition is often not the eigendecomposition, but the *singular value decomposition*

$$A = U\Sigma V^T$$

where U and V have orthonormal columns (the left and right *singular vectors*) and $\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots)$ is the matrix of *singular values*. The singular values are non-negative; by convention, they should be in ascending order.

3.2 Calculus background

Taylor approximation in 1D If $f : \mathbb{R} \to \mathbb{R}$ has k continuous derivatives, then Taylor's theorem with remainder is

$$f(x+z) = f(x) + f'(x)z + \ldots + \frac{1}{(k-1)!}f^{(k-1)}(x) + \frac{1}{k!}f^{(k)}(x+\xi)$$

where $\xi \in [x, x+z]$. We usually work with simple linear approximations, i.e.

$$f(x+z) = f(x) + f'(x)z + O(z^2),$$

though sometimes we will work with the quadratic approximation

$$f(x+z) = f(x) + f'(x)z + \frac{1}{2}f''(x)z^2 + O(z^3).$$

In both of these, when say the error term $e(z)$ is $O(g(z))$, we mean that for small enough z, there is some constant C such that

$$|e(z)| \leq Cg(z).$$

We don't need to remember a library of Taylor expansions, but it is useful to remember that for $|\alpha| < 1$, we have the geometric series

$$\sum_{j=0}^{\infty} \alpha^j = (1-\alpha)^{-1}.$$

Taylor expansion in multiple dimensions In more than one space dimension, the basic picture of Taylor's theorem remains the same. If $f : \mathbb{R}^n \to \mathbb{R}^m$, then

$$f(x+z) = f(x) + f'(x)z + O(\|z\|^2)$$

where $f'(x) \in \mathbb{R}^{m \times n}$ is the *Jacobian matrix* at x. If $\phi : \mathbb{R}^n \to \mathbb{R}$, then

$$\phi(x+z) = \phi(z) + \phi'(x)z + \frac{1}{2}z^T\phi''(z)z + O(\|z\|^3).$$

The row vector $\phi'(x) \in \mathbb{R}^{1 \times n}$ is the derivative of ϕ, but we often work with the *gradient* $\nabla\phi(x) = \phi'(x)^T$. The *Hessian* matrix $\phi''(z)$ is the matrix of second partial derivatives of ϕ. Going beyond second order expansion of ϕ (or going beyond a first order expansion of f) requires that we go beyond matrices and vectors to work with tensors involving more than two indices. For this class, we're not going there.

Variational notation A *directional derivative* of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ in the direction u is

$$\frac{\partial f}{\partial u}(x) \equiv \left.\frac{d}{ds}\right|_{s=0} f(x + su) = f'(x)u.$$

A nice notational convention, sometimes called *variational* notation (as in "calculus of variations") is to write

$$\delta f = f'(x)\delta u,$$

where δ should be interpreted as "first order change in." In introductory calculus classes, this sometimes is called a total derivative or total differential, though there one usually uses d rather than δ. There is a good reason for using δ in the calculus of variations, though, so that's typically what I do.

Variational notation can tremendously simplify the calculus book-keeping for dealing with multivariate functions. For example, consider the problem of differentiating A^{-1} with respect to every element of A. I would compute this by thinking of the relation between a first-order change to A^{-1} (written $\delta[A^{-1}]$) and a corresponding first-order change to A (written δA). Using the product rule and differentiating the relation $I = A^{-1}A$, we have

$$0 = \delta[A^{-1}A] = \delta[A^{-1}]A + A^{-1}\delta A.$$

Rearranging a bit gives

$$\delta[A^{-1}] = -A^{-1}[\delta A]A^{-1}.$$

One *can* do this computation element by element, but it's harder to do it without the computation becoming horrible.

Matrix calculus rules There are some basic calculus rules for expressions involving matrices and vectors that are easiest to just remember. These are naturally analogous to the rules in 1D. If f and g are differentiable maps whose composition makes sense, the multivariate chain rule says

$$\delta[f(g(x))] = f'(g(x))\delta g, \quad \delta g = g'(x)\delta x$$

If A and B are matrix-valued functions, we also have

$$\begin{aligned}
\delta[A + B] &= \delta A + \delta B \\
\delta[AB] &= [\delta A]B + A[\delta B], \\
\delta[A^{-1}B] &= -A^{-1}[\delta A]A^{-1}B + A^{-1}\delta B
\end{aligned}$$

and so forth. The big picture is that the rules of calculus work as well for matrix-valued functions as for scalar-valued functions, and the main changes account for the fact that matrix multiplication does not commute. You should be able to convince yourself of the correctness of any of these rules using the component-by-component reasoning that you most likely learned in an introductory calculus class, but using variational notation (and the ideas of linear algebra) simplifies life immensely.

A few other derivatives are worth having at your fingertips (in each of the following formulas, x is assumed variable while A and b are constant

$$\begin{aligned}
\delta[Ax - b] &= A\delta x \\
\delta[\|x\|^2] &= 2x^T \delta x \\
\delta\left[\frac{1}{2}x^T Ax - x^T b\right] &= (\delta x)^T (Ax - b) \\
\delta\left[\frac{1}{2}\|Ax - b\|^2\right] &= (A\delta x)^T (Ax - b)
\end{aligned}$$

and if $f : \mathbb{R}^n \to \mathbb{R}^n$ is given by $f_i(x) = \phi(x_i)$, then

$$\delta[f(x)] = \operatorname{diag}(\phi'(x_1), \ldots, \phi'(x_n))\, \delta x.$$

3.3 CS background

Order notation and performance Just as we use big-O notation in calculus to denote a function (usually an error term) that goes to zero at a controlled rate as the argument goes to zero, we use big-O notation in algorithm analysis to denote a function (usually run time or memory usage) that grows at a controlled rate as the argument goes to infinity. For instance, if we say that computing the dot product of two length n vectors is an $O(n)$ operation, we mean that the time to compute the dot products of length greater than some fixed constant n_0 is bounded by Cn for some constant C. The point of this sort of analysis is to understand how various algorithms scale with problem size without worrying about all the details of implementation and architecture (which essentially affect the constant C).

Most of the major factorizations of *dense* numerical linear algebra take $O(n^3)$ time when applied to square $n \times n$ matrices, though some building blocks (like multiplying a matrix by a vector or scaling a vector) take $O(n^2)$ or $O(n)$ time. We often write the algorithms for factorizations that take $O(n^3)$

time using block matrix notation so that we can build these factorizations from a few well-tuned $O(n^3)$ building blocks, the most important of which is matrix-matrix multiplication.

Graph theory and sparse matrices In *sparse* linear algebra, we consider matrices that can be represented by fewer than $O(n^2)$ parameters. That might mean most of the elements are zero (e.g. as in a diagonal matrix), or it might mean that there is some other low-complexity way of representing the matrix (e.g. the matrix might be a rank-1 matrix that can be represented as an outer product of two length n vectors). We usually reserve the word "sparse" to mean matrices with few nonzeros, but it is important to recognize that there are other *data-sparse* matrices in the world.

The *graph* of a sparse matrix $A \in \mathbb{R}^{N \times N}$ consists of a set of N vertices $\mathcal{V} = \{1, 2, \ldots, N\}$ and a set of edges $\mathcal{E} = \{(i, j) : a_{ij} \neq 0\}$. While the cost of general dense matrix operations usually depends only on the sizes of the matrix involved, the cost of sparse matrix operations can be highly dependent on the structure of the associated graph.

3.4 MATLAB background

Building matrices and vectors MATLAB gives us several standard matrix and vector construction functions.

```
I = eye(n);     % Build n-by-n identity
Z = zeros(n);   % n-by-n matrix of zeros
b = rand(n,1);  % n-by-1 random matrix (uniform)
e = ones(n,1);  % n-by-1 matrix of ones
D = diag(e);    % Construct a diagonal matrix
e2 = diag(D);   % Extract matrix diagonal
```

Concatenating matrices and vectors In addition to functions for constructing specific types of matrices and vectors, MATLAB lets us put together matrices and vectors by horizontal and vertical concatenation. This works with matrices just as well as with vectors!

```
x = [1;  2];        % Column vector
y = [1,  2];        % Row vector
M = [1, 2; 3,  4];  % 2-by-2 matrix
M = [I, A];         % Horizontal matrix concatenation
```

Transpose and rearrangemenent MATLAB lets us rearrange the data inside a matrix or vector in a variety of ways. In addition to the usual transposition operation, we can also do "reshape" operations that let us interpret the same data layout in computer memory in different ways.

```
% Reshape A to a vector, then back to a matrix
% Note: MATLAB is column-major
avec = reshape(A, prod(size(A)));
A = reshape(avec, n, n);

A = A'; % Conjugate transpose
A = A.'; % Simple transpose

idx = randperm(n); % Random permutation of indices
Ac = A(:,idx);     % Permute columns of A
Ar = A(idx,:);     % Permute rows of A
Ap = A(idx,idx);   % Permute rows and columns
```

Submatrices, diagonals, and triangles MATLAB lets us extract specific parts of a matrix, like the diagonal entries or the upper or lower triangle.

```
A = randn(6,6); % 6-by-6 random matrix
A(1:3,1:3)      % Leading 3-by-3 submatrix
A(1:2:end,:)    % Rows 1, 3, 5
A(:,3:end)      % Columns 3-6

Ad = diag(A);   % Diagonal of A (as vector)
A1 = diag(A,1); % First superdiagonal
Au = triu(A);   % Upper triangle
Al = tril(A);   % Lower triangle
```

Matrix and vector operations MATLAB provides a variety of *elementwise* operations as well as linear algebraic operations. To distinguish elementwise multiplication or division from matrix multiplication and linear solves or least squares, we put a dot in front of the elementwise operations.

```
y = d.*x; % Elementwise multiplication of vectors/matrices
y = x./d; % Elementwise division
```

```
z = x + y;  % Add vectors/matrices
z = x + 1;  % Add scalar to every element of a vector/matrix

y = A*x;    % Matrix times vector
y = x'*A;   % Vector times matrix
C = A*B;    % Matrix times matrix

% Don't use inv!
x = A\b;    % Solve Ax = b *or* least squares
y = b/A;    % Solve yA = b or least squares
```

Things best avoided There are few good reasons to compute explicit matrix inverses or determinants in numerical computations. MATLAB does provide these operations. But if you find yourself typing `inv` or `det` in MATLAB, think long and hard. Is there an alternate formulation? Could you use the forward slash or backslash operations for solving a linear system?

3.5 Floating point

Most floating point numbers are essentially *normalized scientific notation*, but in binary. A typical normalized number in double precision looks like

$$(1.b_1b_2b_3\ldots b_{52})_2 \times 2^e$$

where $b_1 \ldots b_{52}$ are 52 bits of the *significand* that appear after the binary point. In addition to the normalize representations, IEEE floating point includes subnormal numbers (the most important of which is zero) that cannot be represented in normalized form; $\pm\infty$; and Not-a-Number (NaN), used to represent the result of operations like $0/0$.

The rule for floating point is that "basic" operations (addition, subtraction, multiplication, division, and square root) should return the true result, correctly rounded. So a MATLAB statement

```
% Compute the sum of x and y (assuming they are exact)
z = x + y;
```

actually computes $\hat{z} = \mathrm{fl}(x + y)$ where $\mathrm{fl}(\cdot)$ is the operator that maps real numbers to the closest floating point representation. For numbers that are in

the normalized range (i.e. for which $\mathrm{fl}(z)$ is a normalized floating point number), the relative error in approximating z by $\mathrm{fl}(z)$ is smaller in magnitude than machine epsilon; for double precision, $\epsilon_{\mathrm{mach}} = 2^{-53} \approx 1.1 \times 10^{-16}$; that is,

$$\hat{z} = z(1+\delta), \quad |\delta| \leq \epsilon_{\mathrm{mach}}.$$

We can *model* the effects of roundoff on a computation by writing a separate δ term for each arithmetic operation in MATLAB; this is both incomplete (because it doesn't handle non-normalized numbers properly) and imprecise (because there is more structure to the errors than just the bound of machine epsilon). Nonetheless, this is a useful way to reason about roundoff when such reasoning is needed.

3.6 Sensitivity, conditioning, and types of error

There are several different ways we can think about error. The most obvious is the *forward error*: how close is our approximate answer to the correct answer? One can also look at *backward error*: what is the smallest perturbation to the problem such that our approximation is the true answer? Or there is *residual error*: how much do we fail to satisfy the defining equations?

For each type of error, we have to decide whether we want to look at the *absolute* error or the *relative* error. For vector quantities, we generally want the *normwise* absolute or relative error, but often it's critical to choose norms wisely. The *condition number* for a problem is the relation between relative errors in the input (e.g. the right hand side in a linear system of equations) and relative errors in the output (e.g. the solution to a linear system of equations). Typically, we analyze the effect of roundoff on numerical methods by showing that the method in floating point is *backward stable* (i.e. the effect of roundoffs lead to an error that is bounded by some polynomial in the problem size times ϵ_{mach}) and separately trying to show that the problem is *well-conditioned* (i.e. small backward error in the problem inputs translates to small forward error in the problem outputs).

We are usually concerned with *first-order* error analysis, i.e. error analysis based on a linearized approximation to the true problem.

3.7 Problems

1. Consider the mapping from quadratic polynomials to cubic polynomials given by $p(x) \mapsto xp(x)$. With respect to the power basis $\{1, x, x^2, x^3\}$,

what is the matrix associated with this mapping?

2. Consider the mapping from functions of the form $f(x, y) = c_1 + c_2 x + c_3 y$ to values at (x_1, y_1), (x_2, y_2), and (x_3, y_3). What is the associated matrix? How would you set up a system of equations to compute the coefficient vector c associated with a vector b of function values at the three points?

3. Consider the L^2 inner product between quadratic polynomials on the interval $[-1, 1]$:

$$\langle p, q \rangle = \int_{-1}^{1} p(x)q(x)\,dx$$

If we write the polynomials in terms of the power basis $\{1, x, x^2\}$, what is the matrix associated with this inner product (i.e. the matrix A such that $c_p^T A c_q = \langle p, q \rangle$ where c_p and c_q are the coefficient vectors for the two polynomials.

4. Consider the weighted max norm

$$\|x\| = \max_j w_j |x_j|$$

where $w_1, \ldots, w_n$ are positive weights. For a square matrix A, what is the operator norm associated with this vector norm?

5. If A is symmetric and positive definite, argue that the eigendecomposition is the same as the singular value decomposition.

6. Consider the block matrix

$$M = \begin{bmatrix} A & B \\ B^T & D \end{bmatrix}$$

where A and D are symmetric and positive definite. Show that if

$$\lambda_{\min}(A)\lambda_{\min}(D) \geq \|B\|_2^2$$

then the matrix M is symmetric and positive definite.

7. Suppose D is a diagonal matrix such that $AD = DA$. If $a_{ij} \neq 0$ for $i \neq j$, what can we say about D?

8. Convince yourself that the product of two upper triangular matrices is itself upper triangular.

9. Suppose Q is a differentiable *orthogonal* matrix-valued function. Show that $\delta Q = QS$ where S is skew-symmetric, i.e. $S = -S^T$.

10. Suppose $Ax = b$ and $(A + D)y = b$ where A is invertible and D is relatively small. Assuming we have a fast way to solve systems with A, give an algorithm to compute y to within an error of $O(\|D\|^2)$ in terms of two linear systems involving A and a diagonal scaling operation.

11. Suppose $r = b - A\hat{x}$ is the residual associated with an approximate solution $\hat{x}$. The *maximum componentwise relative residual* is

$$\max_i |r_i|/|b_i|.$$

How can this be written in terms of a norm?

4 Linear systems

We start with

$$Ax = b$$

where $A \in \mathbb{R}^{n \times n}$ is square and nonsingular. We initially consider *direct solvers* that compute x in a finite number of steps using a factorization of A.

4.1 Sensitivity and conditioning of linear systems

We care about the sensitivity of linear systems for two reasons. First, we compute using floating point, and the standard analysis of many numerical methods involves analyzing backward stability (a property purely of the algorithm) together with conditioning (a property purely of the problem). Second, many problems inherit error in the input data from measurements or from other computations, and sensitivity analysis is needed to analyze how sensitive a computation might be to these input errors.

In most of our error analysis, we assume that a standard norm (the 1-norm, 2-norm, or max-norm) is a reasonable way to measure the sizes of inputs and outputs. But if different elements of the input or output represent values with different units, the problem might be *ill-scaled*. For this reason, it usually makes sense to scale the system before doing any error analysis (or solves).

Matrix multiplication Suppose $A \in \mathbb{R}^{n \times n}$ is nonsingular, and consider the computation

$$y = Ax.$$

Here we treat x as an input and y as an output. The condition number relates relative perturbations to the input to relative perturbations to the output. That is, given

$$\hat{y} = A\hat{x},$$

we would like to compute a bound on $\|\hat{y} - y\|/\|y\|$ in terms of $\|\hat{x} - x\|/\|x\|$. For any consistent matrix norm,

$$\begin{aligned} \|x\| = \|A^{-1}y\| &\leq \|A^{-1}\|\|y\| \\ \|\hat{y} - y\| = \|A(\hat{x} - x)\| &\leq \|A\|\|\hat{x} - x\| \end{aligned}$$

and therefore

$$\frac{\|\hat{y} - y\|}{\|y\|} \leq \kappa(A) \frac{\|\hat{x} - x\|}{\|x\|}, \quad \kappa(A) \equiv \|A\|\|A^{-1}\|.$$

We call $\kappa(A)$ the *condition number with respect to multiplication.*

Another perspective is to consider perturbations not to x, but to A:

$$\hat{y} = \hat{A}x, \quad \hat{A} = A + E$$

In this case, we have

$$\|\hat{y} - y\| = \|E(\hat{x} - x)\| \leq \|E\|\|\hat{x} - x\|$$

and

$$\frac{\|\hat{y} - y\|}{\|y\|} \leq \kappa(A) \frac{\|E\|}{\|A\|},$$

where $\kappa(A) = \|A\|\|A^{-1}\|$ as before.

Linear systems Now suppose $A \in \mathbb{R}^{n \times n}$ is nonsingular and consider the linear solve

$$Ax = b.$$

If $\hat{x}$ is an approximate solution, the corresponding residual is

$$r = b - A\hat{x}$$

or, put differently

$$\hat{x} = A^{-1}(b + r).$$

Using the sensitivity analysis for matrix multiplication, we have

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}.$$

We can also look at the sensitivity with respect to perturbations to A. Let $\hat{A} = A + E$. Using the Taylor expansion of the inverse about A, we have

$$\hat{A}^{-1} = A^{-1} - A^{-1}EA^{-1} + O(\|E\|^2).$$

Therefore if $\hat{x} = \hat{A}^{-1}b$, we have

$$\hat{x} - x = -A^{-1}Ex + O(\|E\|^2),$$

and by manipulating norm bounds,

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \kappa(A) \frac{\|E\|}{\|A\|} + O(\|E\|^2).$$

Geometry of ill-conditioning In the case of the matrix two-norm, we have

$$\kappa_2(A) \equiv \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}.$$

where $\sigma_{\max}(A) = \|A\|$ and $\sigma_{\min}(A) = 1/\|A^{-1}\|$ are the largest and smallest singular values of A. The two-norm condition number can be interpreted geometrically as the ratio between the longest and the smallest axes of the elliptical region

$$\{Ax : \|x\| \leq 1\}.$$

4.2 Gaussian elimination

We think of Gaussian elimination as an algorithm for factoring a nonsingular matrix A as

$$PA = LU$$

where P is a permutation, L is unit lower triangular, and U is upper triangular. Given such a factorization, we can solve systems involving A by forward and backward substitution involving L and U.

The simplest case for Gaussian elimination is a (block) 2-by-2 system in which no pivoting is required:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}.$$

Reading off the matrix products, we have

$$\begin{aligned} L_{11}U_{11} &= A_{11} \\ L_{11}U_{12} &= A_{12} \\ L_{21}U_{11} &= A_{21} \\ L_{22}U_{22} &= A_{22} - L_{21}U_{12} = A_{22} - A_{21}A_{11}^{-1}A_{12}. \end{aligned}$$

That is, we can compute the L and U by solving the smaller subproblem of factoring A_{11}, then computing the off-diagonal blocks via triangular solves, and then factoring the *Schur complement* $A_{22} - L_{21}U_{12}$. Like matrix multiplication, we can think of Gaussian elimination in several different ways, and choosing the submatrices differently provides different strategies for Gaussian elimination.

We can also think of Gaussian elimination as applying a sequence of *Gauss transformations* (or *elementary transformations*). This perspective is particularly useful when we think about analogous algorithms based on orthogonal transformations (Givens rotations or Householder reflections) which lead to methods for QR factorization. However we think about the method, dense Gaussian elimination involves three nested loops (like matrix multiplication) and takes $O(n^3)$ time. Once we have a factorization, solving linear systems with it takes $O(n^2)$ time.

In general, of course, pivoting may be needed. The usual *row pivoting* strategy guarantees that all the entries of L below the main diagonal are less than one. Alternate pivoting strategies are possible, and are particularly attractive in a sparse or parallel settings (where the data motion associated with pivoting is annoyingly expensive).

Gaussian elimination is *usually* backward stable, i.e. the computed L and U correspond to some $\hat{A}$ which is close to A in the sense of normwise relative error. It is possible to construct examples where the backward error grows terribly, but these occur fairly rarely.

4.3 LU and Cholesky

If A is symmetric and positive definite, we may prefer *Cholesky* factorization to Gaussian elimination. The Cholesky factorization is

$$A = R^T R$$

where R is upper triangular (sometimes this is also written LL^T where L is lower triangular). The Cholesky factorization exists and is nonsingular iff A is positive definite (if A is semidefinite, a singular Cholesky factor may exist). Attempting to compute the Cholesky factorization is a standard method for testing positive definiteness of a matrix.

As with Gaussian elimination, we can think of the factorization in blocky form as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} = \begin{bmatrix} R_{11}^T & 0 \\ R_{12}^T & R_{22}^T \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

This again leads to an algorithm in which we factor the A_{11} block, use triangular solves to compute the off-diagonal block of the Cholesky factor, and then form and factor a Schur complement matrix. For an SPD matrix, Chokesky factorization is backward stable even without pivoting.

The Cholesky factorization must succeed because every Schur complement of a symmetric positive-definite matrix is again symmetric and positive-definite. The *strictly diagonally dominant* matrices share a similar property, and can also safely be factored (via LU) without pivoting.

4.4 Sparse solvers

If the matrix A is large and sparse, we might consider using a sparse direct factorization method. Typically, sparse LU or Cholesky look like

$$PAQ = LU \quad \text{or} \quad QAQ^T = R^T R$$

where the column permutation Q (or symmetric permutation, in the case of Cholesky) is chosen to minimize *fill*, and the row permutation P is chosen for stability. We say a nonzero element of L or U is a *fill element* if the corresponding location in A is zero.

The Cholesky factorization of a sparse SPD matrix involves *no* fill if the corresponding graph is a tree and if the ordering always places children before parents (a *bottom-up* ordering). Matrices that look "almost" like trees can be efficiently dealt with by sparse factorization methods. It turns out that 2D meshes are usually fine, 3D meshes get expensive fast, and most "small world" graphs generate tremendous amounts of fill.

Band matrices are sparse matrices in which all the nonzero elements are restricted to a narrow region around the diagonal. Band matrices are sufficiently common and regular that they are handled by libraries like LAPACK that are devoted to dense linear algebra algorithms. LAPACK does not deal with more general sparse matrices.

4.5 Iterative refinement

Suppose $\hat{A} = LU$ where $\hat{A}$ is an "okay" approximation of A. Such a factorization might be computed by ignoring pivoting, or by using lower-precision arithmetic at an intermediate stage, or we might just have a case where Gaussian elimination with partial pivoting is not quite backward stable. In this case, we can "clean up" an approximate solution by *iterative refinement*:

$$\begin{aligned} x_0 &= U^{-1}(L^{-1}b) \\ x_{k+1} &= x_k + U^{-1}(L^{-1}(b - Ax_k)) \end{aligned}$$

The error $e_k = x_k - x$ satisfies the iteration

$$e_{k+1} = (I - \hat{A}^{-1}A)e_k,$$

and iterative refinement converges quickly if $\|I - \hat{A}^{-1}A\| \ll 1$.

4.6 MATLAB backslash

For a square matrix, the MATLAB backslash operator checks whether the matrix is sparse or dense, triangular, permuted triangular, diagonal, upper Hessenberg, symmetric, etc. and chooses an appropriate solver strategy. It is smart about applying an appropriate fill-reducing ordering in the sparse case, takes only $O(n^2)$ time for triangular matrices, and in general does "the right thing." If you are solving a linear system, you should always use backslash instead of `inv`.

One thing that MATLAB backslash does *not* do is to see whether you've already solved a linear system involving the matrix in question. If you want to re-use a factorization, you need to do so yourself. This typically looks something like

```
[L,U,P] = lu(A);  % O(n^3)
x = U\(L\(P*b)); % O(n^2)
% Compute a new RHS d
y = U\(L\(P*b)); % O(n^2) again
% ...
```

4.7 Problems

1. Suppose A is square and singular, and consider $y = Ax$. Show by example that a *finite* relative error in the input x can lead to an *infinite* relative error in the output y.

2. Give a 2×2 example for which an $O(\epsilon_{\text{mach}})$ normwise relative residual corresponds to a normwise relative error near one.

3. Show that $\kappa_2(A) = 1$ iff A is a scalar multiple of an orthogonal matrix.

4. Suppose M is the elementary transformation matrix

$$M = \begin{bmatrix} 1 & 0 \\ m & I \end{bmatrix}.$$

What is M^{-1}?

5. Compute the Cholesky factorization of the matrix

$$A = \begin{bmatrix} 4 & 2 \\ 2 & 9 \end{bmatrix}$$

6. Consider the matrix

$$\begin{bmatrix} D & u \\ u^T & \alpha \end{bmatrix}$$

where D is diagonal with positive diagonal elements larger than the corresponding entries of u. For what range of α must u be positive definite?

7. If A is symmetric and positive definite with Cholesky factor R, show that $\kappa_2(A) = \kappa_2(R)^2$ (note: use the SVD).

8. If $\hat{A} = LU = A + E$, show that iterative refinement with the computed LU factors satisfies

$$\|e_{k+1}\| \leq \left(\kappa(\hat{A}) \frac{\|E\|}{\|\hat{A}\|} \right) \|e_k\|$$

5 Least squares problems

Consider the equations

$$Ax = b$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$. We typically consider $m > n$. The system of equations may be *over-determined*, i.e. b is not in the range of A. In this case, we usually instead solve the least squares problem

$$\text{minimize } \|Ax - b\|^2$$

The system may also be *ill-posed*, i.e. the columns of A are linearly dependent (or nearly linearly dependent). These conditions are not mutually exclusive.

When n and m are small and the matrix A is dense, we can solve either linear systems or least squares problems using a few standard matrix factorizations: LU/Cholesky, QR, or SVD. When n and m are large and sparse, we may be able to use a sparse direct solver (assuming the graph of the matrix in question is "nice"). Otherwise, we may prefer an iterative solver.

5.1 Sensitivity and conditioning of least squares

Suppose $A \in \mathbb{R}^{m \times n}$ with $m > n$ has singular values $\sigma_1 > \ldots > \sigma_n$. The *condition number for least squares* is $\kappa(A) = \sigma_1/\sigma_n$. If $\hat{x}$ is an approximate solution to the least squares problem with residual $\hat{r} = b - \hat{A}x$, then

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \kappa(A) \frac{\|\hat{r}\|}{\|b\|}.$$

This is extremely similar to one of the bounds we saw for linear systems. Of course, $\hat{r}$ will not necessarily be close to zero even if $\hat{x}$ is close to x!

It's possible to compute the sensitivity to perturbations to the matrix A as well, but this is much messier than in the linear system case.

5.2 Normal equations

The *normal equations* for minimizing $\|Ax - b\|$ are

$$A^T Ax = A^T b.$$

These equations exactly correspond to finding a stationary point $\nabla\phi(x) = 0$ where

$$\phi(x) = \frac{1}{2}\|Ax - b\|^2.$$

The equations are called the normal equations because they can be written as

$$A^T r = 0, \quad r = b - Ax,$$

i.e. the residual at the solution is orthogonal (normal) to everything in the range space of A.

The matrix $A^T A$ is sometimes called the *Gram matrix*. It is symmetric and positive definite (assuming that A has full column rank), but $\kappa(A^T A) = \kappa(A)^2$. Hence, if the conditioning of A is a concern, we might not want to solve the normal equations exactly.

5.3 QR

Given a matrix $A \in \mathbb{R}^{m\times n}$ with $m > n$, the *economy* QR decomposition of A is

$$A = QR, \quad Q \in \mathbb{R}^{m\times n}, R \in \mathbb{R}^{n\times n}$$

where Q has orthonormal columns and R is upper triangular. In the *full* QR decomposition, Q is square and R is a rectangular upper triangular matrix. The QR decomposition can be computed via the Gram-Schmidt process applied to the columns of A, though this is not backward stable and is not the preferred approach most of the time. The QR decomposition can be computed in a backward stable fashion via the *Householder QR* procedure, which applies n simple orthogonal transformations (Householder reflections of the form $I - 2uu^T$ where $\|u\| = 1$) that "zero out" the subdiagonal elements in each of the n columns in turn.

The QR decomposition is closely related to the normal equations system: R is the Cholesky factor of $A^T A$ (to within a sign-rescaling of the diagonal of R) and $Q = AR^{-1}$ has orthonormal columns. But while computing R by Cholesky factorization of $A^T A$ involves a sub-problem with condition number $\kappa(A)^2$, solving the system

$$R^T x = Q^T b$$

involves just a solve with R, which has the same condition number as A.

Solving a least squares problem via QR is moderately more expensive than solving the Cholesky problem. However, QR is somewhat more numerically stable.

5.4 SVD

Just as we can solve the least squares problem with the economy QR decomposition, we can also solve with the economy SVD $A = U\Sigma V^T$:

$$x = V\Sigma^{-1}U^T b.$$

If $A \in \mathbb{R}^{m\times n}$ has the economy QR decomposition $A = QR$, we can compute the economy SVD of A using the QR decomposition together with the SVD of R. If m is sufficiently larger than n, then most of the work goes into the QR step.

The SVD is even more numerically stable than QR decomposition, but the primary reason for using the SVD is that we can analyze the behavior of the singular values for reasoning about ill-posed problems.

5.5 Pseudo-inverses

The process of solving a least squares problem is a linear operation, which we write as

$$x = A^\dagger b.$$

The symbol $A^\dagger$ is the (Moore-Penrose) *pseudoinverse*, which we expand as

$$\begin{aligned} A^\dagger &= (A^T A)^{-1} A && \text{(Normal equations)} \\ &= R^{-T} Q && \text{(QR)} \\ &= V\Sigma^{-1} U^T && \text{(SVD)} \end{aligned}$$

5.6 Ill-posed problems and regularization

An *ill-posed* least squares problem is one in which the matrix A is ill-conditioned. This means that there is a large set of vectors $\hat{x}$ that explain the data equally well – that is $A(\hat{x} - x)$ is around the same order of magnitude as the error in the measurement vector b. Hence, the data is not good enough to tell us which of the possible solution vectors is really appropriate, and the usual

least-squares approach *overfits* the data, and if the coefficients x are later used to model behavior at a new data point, the prediction will be poor.

When the data does not provide enough information to fit a model, we need to incorporate prior knowledge that is not based on the data. This leads to *regularized least squares*. Some common approaches include

- Factor selection, i.e. predicting based on only a subset of the columns of A:

  $$\tilde{x}_{\mathcal{I}} = A^{\dagger}_{:,\mathcal{I}} b.$$

  The relevant subset $\mathcal{I}$ may be determined using QR with column pivoting or using more sophisticated heuristics based on an SVD.

- Truncated SVD, i.e. computing

  $$\tilde{x} = V_{:,k} \Sigma^{-1}_{1:k,1:k} U^T_{:,k} b.$$

  This completely discards the influence of "poorly-behaved" directions.

- Tikhonov regularization, i.e. minimizing

  $$\phi_{\text{Tik}}(x; \lambda) = \frac{1}{2}\|Ax - b\|^2 + \frac{\lambda^2}{2}\|x\|^2_M$$

  where M is some positive definite matrix and λ is a small regularization parameter. The first term penalizes mismatch to the data; the second term penalizes overly large coefficients.

Each of these approaches has a parameter that controls the balance between fitting the data and enforcing the assumptions. For methods based on subset selection or truncated SVD, one has to choose the number of retained directions k; for Tikhonov regularization, one has to choose the regularization parameter λ. If something is known in advance about the error, these parameters can be chosen a priori. Usually, though, one chooses the parameter in an adaptive way based on some criterion. Examples include the PRESS statistic, corresponding to the sum of squared prediction errors in a leave-one-out cross-validation process, or using the "L-curve" (topics we discussed briefly toward the end of the semester).

5.7 Problems

1. Suppose M is symmetric and positive definite, so that $\|x\|_M = \sqrt{x^T M x}$ is a norm. Write the normal equations for minimizing $\|Ax - b\|_M^2$.

2. Suppose $A \in \mathbb{R}^{n \times 1}$ is a vector of all ones. Show that $A^\dagger b$ is the sample mean of the entries of b.

3. Suppose $A = QR$ is an economy QR decomposition. Why is $\kappa(A) = \kappa(R)$?

4. Suppose we have economy QR decompositions for $A_1 \in \mathbb{R}^{m_1 \times n}$ and $A_2 \in \mathbb{R}^{m_2 \times n}$, i.e.
$$A_1 = Q_1 R_1, \quad A_2 = Q_2 R_2$$
Show that we can compute the QR decomposition of A_1 and A_2 stacked as
$$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = QR, \quad Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \tilde{Q}$$
where
$$\tilde{Q} R = \begin{bmatrix} R_1 \\ R_2 \end{bmatrix}$$
is an economy QR decomposition.

5. Give an example of $A \in \mathbb{R}^{2 \times 1}$ and $b \in \mathbb{R}^2$ such that a small relative change to b results in a large relative change to the solution of the least squares problem. What is the condition number of A?

6. Write the normal equations for a Tikhonov-regularized least squares problem.

7. Show that $\Pi = AA^\dagger$ is a projection ($\Pi^2 = \Pi$) and that Πb is the closest point to b in the range of A.

8. Using the normal equations approach, find the coefficients α and β that minimize
$$\phi(\alpha, \beta) = \int_{-1}^{1} (\alpha + \beta x - f(x))^2 \, dx$$

6 Eigenvalues

In this section, we discussed the eigenvalue problem

$$Ax = \lambda x.$$

Depending on the context, one might want all eigenvalues of A or only some; eigenvalues only, eigenvectors only, or both eigenvalues and eigenvectors; row and column eigenvectors or only one or the other. Different methods give different information.

6.1 Why eigenvalues?

There are several reasons why we might want to compute eigenvalues or eigenvectors

- Eigenvalue decompositions are often used to reason about systems of linear differential equations or difference equations. Eigenvalues give information about how special solutions grow, decay, or oscillate; eigenvectors give the corresponding "mode shapes".
- Eigenvalue problems involving tridiagonal matrices are common in the theory of special functions, and play an important role in numerical quadrature methods (for example).
- Eigenvalue problems play a special role in linear control theory, and eigenvalue decompositions can be efficiently solve a variety of problems that arise there. We gave one example (Sylvester equations) in lecture.
- Symmetric eigenvalue problems are among the few *non-convex* optimization problems that we know how to reliably solve. Many other optimization problems can be approximated by (relaxed to) eigenvalue problems. This is the basis of spectral graph partitioning and spectral clustering, for example.
- Eigenvalue finding and polynomial root finding are essentially equivalent. For example, in MATLAB, the `roots` command finds the roots of a polynomial via an equivalent eigenvalue problem.

There is also information that can be derived by eigenvalues *or* by other methods. Often an eigenvalue decomposition is useful for analysis, and another approach is useful for computation.

6.2 Jordan to Schur

In an introductory class, you may have learned about the Jordan canonical form. For almost all matrices, we have a basis of eigenvectors V and can write

$$AV = V\Lambda.$$

In some cases in which we have eigenvalues with high multiplicity, we may need generalized eigenvectors, and replace the eigenvalue matrix Λ with a matrix that has eigenvalues on the diagonal and some ones on the first superdiagonal. However, the Jordan form is *discontinuous* in the entries of the matrix; an infinitesimally small perturbation can change one of the superdiagonal elements from a zero to a one. Also, the eigenvector matrix V can in general be rather poorly behaved (unless A is symmetric or has other special structure).

Numerical analysts prefer the *Schur form* to the Jordan form. The (complex) Schur form is

$$AU = UT$$

where $U \in \mathbb{C}^{n\times n}$ is a unitary matrix and $T \in \mathbb{C}^{n\times n}$ is upper triangular. The real Schur form is

$$AQ = QT$$

where $Q \in \mathbb{R}^{n\times n}$ is an orthogonal matrix and $T \in \mathbb{R}^{n\times n}$ is a block upper triangular matrix with 1×1 blocks (corresponding to real eigenvalues) and 2×2 blocks (corresponding to complex conjugate pairs) on the diagonal. As in the Jordan form, the diagonal elements of T in the complex Schur form are the eigenvalues; but where the columns of V in the Jordan form correspond to eigenvectors (or generalized eigenvectors), the columns of U or Q in the Schur factor form bases for a sequence of nested invariant subspaces. That is, for each $1 \leq k \leq n$, we have

$$AU_{:,1:k} = U_{1:k,:}T_{1:k,1:k}$$

in the complex Schur form; and similarly for the real Schur form we have

$$AQ_{:,1:k} = Q_{1:k,:}T_{1:k,1:k}$$

for each $1 \leq k \leq n$ such that taking the first k columns does not split a 2×2 diagonal block.

If we insist, we can recover eigenvectors from the Schur form. Consider the complex Schur form, and suppose we are interested in the eigenvector associated with the eigenvalue t_{kk} (which we will assume for simplicity has multiplicity 1). Then solving the system

$$\left(T_{1:(k-1),1:(k-1)} - t_{kk} I\right) w + T_{1:(k-1),k} = 0$$

gives us a vector

$$v = U_{:,(1:k-1)} w + U_{:,k}$$

with the property that

$$Av = AU_{:,(1:k)} \begin{bmatrix} w \\ 1 \end{bmatrix} = U_{:,(1:k)} T \begin{bmatrix} w \\ 1 \end{bmatrix} = U_{:,(1:k)} t_{kk} \begin{bmatrix} w \\ 1 \end{bmatrix} = v t_{kk}.$$

Hence, computing eigenvectors from the Schur form can be done at the cost of one triangular solve per eigenvector.

We can go the other way as well: given a Jordan form, we can easily compute the corresponding Schur form. Suppose that

$$AV = V\Lambda$$

and let $V = UR$ be a complex QR decomposition of V. Then

$$AU = U(R\Lambda R^{-1}) = UT$$

is a complex Schur form for A.

6.3 Symmetric eigenvalue problems and SVDs

Broadly speaking, I tend to distinguish between two related perspectives on eigenvalues. The first is the linear map perspective: A represents an operator mapping a space to itself, and an eigenvector corresponds to an *invariant direction* for the operator. The second perspective is the quadratic form perspective: if A is a symmetric matrix representing a quadratic form $x^T Ax$, then the eigenvalues and eigenvectors are the stationary values and vectors for the *Rayleigh quotient*

$$\rho_A(x) = \frac{x^T Ax}{x^T x}.$$

If we differentiate $x^T Ax - \rho_A x^T x = 0$, we have

$$2\delta x^T (Ax - \rho_A x) - \delta\rho_a (x^T x) = 0$$

which means that setting $\delta\rho_A = 0$ implies

$$Ax - \rho_A(x)x = 0.$$

The largest eigenvalue is the maximum of the Rayleigh quotient, and the smallest eigenvalue is the minimum of the Rayleigh quotient.

The singular value decomposition can be thought of as a symmetric eigenvalue problem in several different ways. The simplest approach is to consider the stationary points of the function

$$\phi(x) = \frac{\|Ax\|^2}{\|x\|^2} = \frac{x^T A^T Ax}{x^T x}.$$

This is the (square) of the function that appears in the definition of the operator 2-norm, and it is the Rayleigh quotient for the Gram matrix $A^T A$. We can also consider the functional

$$\psi(u, v) = \frac{u^T Av}{\|u\|\|v\|},$$

which we can show, with some calculus and algebra, has stationary points at solutions to the matrix eigenvalue problem

$$\left(\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} - \psi I \right) \begin{bmatrix} u \\ v \end{bmatrix} = 0$$

whose eigenvalues are $\{\pm\sigma_i\}$ where $\{\sigma_i\}$ are the singular values of A,

The Rayleigh quotient is such a powerful tool that the symmetric eigenvalue problem behaves almost like a different problem from the nonsymmetric eigenvalue problems. There are types of error analysis and algorithms that work for the symmetric case and have no real useful analogue in the nonsymmetric case.

6.4 Power method and related iterations

Power method The simplest iteration for computing eigenvalues is the *power method*

$$\tilde{x}_{k+1} = Ax_k$$
$$x_{k+1} = x_{k+1}/\|x_k\|$$

The iterates actually satisfy

$$x_k = \frac{A^k x_0}{\|A^k x_0\|}.$$

and if $A = V\Lambda V^{-1}$ is an eigendecomposition, then

$$A^k = V\Lambda^k V^{-1} = \lambda_1^k (V D^k V^{-1})$$

where $D = \text{diag}(1, \lambda_2/\lambda_1, \ldots, \lambda_n/\lambda_1)$. Assuming $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$, we have

$$A^k x_0 \propto v_1 + O\left(|\lambda_2|^k/|\lambda_1|^k\right),$$

assuming x_0 has some component in the v_1 direction when expressed in the eigenbasis. Hence, the power iteration converges to the eigenvector associated with the largest eigenvalue, and the rate is determined by the ratio of the magnitudes of the largest two eigenvalues

Inverse iteration The problem with power iteration is that it only gives us the eigenvector associated with the dominant eigenvalue, the one farthest from the origin. What if we want the eigenvector associated with the eigenvalue nearest the origin? A natural strategy then is *inverse iteration*:

$$\begin{aligned} \tilde{x}_{k+1} &= A^{-1} x_k \\ x_{k+1} &= x_{k+1}/\|x_k\| \end{aligned}$$

Inverse iteration is simply power iteration on the inverse matrix, which has the eigendecomposition $A^{-1} = V\Lambda^{-1}V^{-1}$. Hence, inverse iteration converges to the eigenvector associated with the eigenvalue nearest zero, and the rate of convergence is determined by the ratio of magnitudes of that eigenvalue and the second-furthest-away.

Shifts If we want to find an eigenvalue close to some given target value (and not just zero), a natural strategy is *shift-invert*:

$$\begin{aligned} \tilde{x}_{k+1} &= (A - \sigma I)^{-1} x_k \\ x_{k+1} &= x_{k+1}/\|x_k\| \end{aligned}$$

The eigendecomposition of $(A-\sigma I)^{-1}$ is $V(\Lambda-\sigma I)^{-1}V^{-1}$, and the eigenvalues nearest σ correspond to the largest magnitudes for $(\lambda - \sigma)^{-1}$

Rayleigh quotient iteration A static shift-invert strategy will converge geometrically (unless the shift is an eigenvalue, in which case convergence is instantaneous). We can accelerate convergence by using increasingly accurate estimates for the eigenvalue as shifts. A natural way to estimate the eigenvalue is using the Rayleigh quotient, which gives us the iteration

$$\begin{aligned}\tilde{x}_{k+1} &= (A - \rho_A(x_k)I)^{-1}x_k \\ x_{k+1} &= x_{k+1}/\|x_k\|\end{aligned}$$

Rayleigh quotient iteration converges superlinearly to isolated eigenvalues – quadratically in the nonsymmetric case, cubically in the symmetric case. Unlike static shift-invert, though, the Rayleigh quotient iteration requires a factorization of a new shifted system at each step.

Subspace iteration So far, we have talked only about iterations for single vectors. *Subspace iteration* generalizes the power iteration idea to multiple vectors. The subspace iteration looks like

$$Q_{k+1}R_{k+1} = AQ_k;$$

that is, at each step we multiply an orthonormal basis for a subspace by A, then re-orthonormalize using a QR decomposition. Subspace decomposition converges like $O(|\lambda_{m+1}|^k/|\lambda_m|^k)$, where m is the dimension of the subspace and the eigenvalues are ordered in descending order of magnitude. The difference between the iterates and the "true" subspace has to be measured in terms of angles rather than vector differences.

The tricks for accelerating power iteration – shift-invert and adaptive shifting – can be applied to subspace iteration as well as to single vector iterations.

6.5 QR iteration

The QR iteration is the workhorse algorithm for solving nonsymmetric dense eigenvalue problems. Named one of the top ten algorithms of the 20th century, the modern QR iteration involves a beautiful combination of elementary ideas put together in a clever way. You are *not* responsible for recalling the details, but you *should* remember at least two ingredients: subspace iteration and Hessenberg reduction.

Nesting in subspace iteration One of the remarkable properties of subspace iteration is that it nests: inside the subspace iteration for a subspace of dimension m sits subspace iteration for subspaces of dimension $m-1, m-2, \ldots, 1$. Hence if A has eigenvalues with distinct moduli, then the iteration

$$Q_{k+1}R_{k+1} = AQ_k, \quad Q_k \in \mathbb{R}^{n\times n}$$

will produce $Q_k \to Q$ where Q is the orthogonal Schur factor for A. Of course, we again need to be careful to measure convergence by angles between corresponding columns of Q and Q_k rather than by the vectors themselves. If the eigenvalues do not have distinct moduli, then Q will correspond to a set of vectors that span nested invariant subspaces.

The first column of the Q_k matrix follows an ordinary power iteration:

$$Q_{k+1}R_{k+1}e_1 = (Q_{k+1}e_1)r_{k+1,11} = A(Q_k e_1),$$

and the last column of Q_{k+1} follows an *inverse* iteration with A^T:

$$R_{k+1}Q_k^T = Q_{k+1}^T A \implies (Q_k e_n)^T \propto (Q_{k+1}e_n^T)A \implies (Q_{k+1}e_n) \propto A^{-T}(Q_k e_n).$$

Hence, a step of *shifted* subspace iteration

$$Q_{k+1}R_{k+1} = (A-\sigma I)Q_k$$

effectively takes a step with a shift-invert transformation for the last vector.

QR iteration Subspace iteration puts the emphasis on the vectors. What about the triangular factor T? If $Q_k \in \mathbb{R}^{n\times n}$ is an approximation for the orthogonal Schur factor, an approximation for the triangular Schur factor is $A^{(k)}$ given by

$$A^{(k)} = Q_k^T A Q_k.$$

You may recognize this as a generalization of the Rayleigh quotient. The subspace iteration recurrence is $AQ_k = Q_{k+1}R_{k+1}$, so

$$A^{(k)} = Q_k^T Q_{k+1}R_{k+1} = \tilde{Q}_{k+1}R_{k+1}, \quad \text{where } \tilde{Q}_{k+1} \equiv Q_k^T Q_{k+1}.$$

Now, magic: we compute $A^{(k+1)}$ from the QR factorization $A^{(k)} = \tilde{Q}_k R_k$:

$$A^{(k+1)} = Q_{k+1}^T A Q_{k+1} = \tilde{Q}_{k+1}^T A^{(k)} \tilde{Q}_{k+1} = R_{k+1}\tilde{Q}_{k+1}.$$

This leads to the simplest version of the *QR iteration*:

$$\begin{aligned} A^{(0)} &= A \\ Q_{k+1}R_{k+1} &= A^{(k)} \\ A^{(k+1)} &= R_{k+1}Q_{k+1} \end{aligned}$$

Shifts in QR The simple QR iteration only converges to the quasi-triangular real Schur factor if all the eigenvalues have different magnitudes. Moreover, as with subspace iteration, the rate of convergence is limited by how close together the magnitudes of the different eigenvalues are. To get fast convergence, we need to include *shifts*:

$$\begin{aligned} A^{(0)} &= A \\ Q_{k+1}R_{k+1} &= A^{(k)} - \sigma_k I \\ A^{(k+1)} &= R_{k+1}Q_{k+1} + \sigma_k I \end{aligned}$$

Using the connection to subspace iteration, choosing $\sigma_k = A^{(k+1)}_{nn}$ ends up being equivalent to a step of Rayleigh quotient iteration.

Hessenberg reduction Incorporating shifts (and choosing the shifts in a clever way) is one of two tricks needed to make QR iteration efficient. The other trick is to convert A to *upper Hessenberg* form before running the iteration, i.e. factoring

$$A = QHQ^T$$

where Q is orthogonal and H is zero below the first subdiagonal. QR factorization of a Hessenberg matrix takes $O(n^2)$ time, and running one step of QR factorization maps a Hessenberg matrix to a Hessenberg matrix.

6.6 Problems

1. The *spectral radius* of a matrix A is the maximum modulus of any of its eigenvalues. Show that $\rho(A) \leq \|A\|$ for any operator norm.

2. Suppose $A \in \mathbb{R}^{n\times n}$ is a symmetric matrix an $V \in \mathbb{R}^{n\times n}$ is invertible. Show that A is positive definite, negative definite, or indefinite iff V^TAV is positive definite, negative definite, or indefinite.

3. Write a MATLAB fragment to take `numiter` steps of shift-invert iteration with a given shift. You should make sure that the cost per iteration is $O(n^2)$, not $O(n^3)$.

4. Suppose T is a block upper-triangular matrix with diagonal blocks in $\mathbb{R}^{1\times 1}$ or $\mathbb{R}^{2\times 2}$. Show that the eigenvalues of T are the diagonal values in the 1×1 blocks together with the eigenvalue pairs from the 2×2 blocks.

5. If $AU = UT$ is a complex Schur form, argue that $A^{-1}U = UT^{-1}$ is the corresponding complex Schur form for A^{-1}.

6. Suppose Q_k is the kth step of a subspace iteration, and Q_* is an orthonormal basis for the subspace to which the iteration is converging. Let θ be the biggest angle between a vector in the range of Q_* and the best approximation by a vector in the range of Q_k, and show that $\cos(\theta)$ is the smallest singular value of $Q_k^T Q_*$.

7. Show that the power method for the Cayley transform matrix $(\sigma I + A)(\sigma I - A)^{-1}$ for $\sigma > 0$ will first converge to an eigenvalue of A with positive real part, assuming such an eigenvalue exists and the iteration converges at all.

8. In control theory, one often wants to plot a *transfer function*

$$h(s) = c^T (A - sI)^{-1} b.$$

   The transfer function can be computed in $O(n^2)$ time using a Hessenberg reduction on A. Describe how.

7 Stationary iterations

Stationary iterations for solving linear systems are rarely used in isolation (except for particularly nicely structured problems). However, they are often used as preconditioners for Krylov subspace methods.

7.1 The splitting picture

Let $A = M - K$ be a *splitting* of the matrix A, and consider the iteration

$$Mx_{k+1} = Kx_k + b.$$

The fixed point equation for this iteration is

$$Mx = Kx + b,$$

which is equivalent to $Ax = b$. Using our usual trick of subtracting the fixed point equation from the iteration equation to get an equation for the errors $e_k = x_k - x$, we have

$$Me_{k+1} = Ke_k \implies e_{k+1} = Re_k, \quad R \equiv M^{-1}K.$$

The matrix R is sometimes called the *iteration matrix*. The iteration converges iff the spectral radius $\rho(R)$ is less than one; recall that the spectral radius is the maximum of the eigenvalue magnitudes of R. A sufficient condition for convergence is that some operator norm of R is less than one, and this is often easier to establish than a bound on the spectral radius.

Ideally, a splitting should have two properties:

1. It should give a convergent method.
2. Applying M^{-1} should be easy.

Some standard choices of splitting are taking M to be the diagonal of A (Jacobi iteration), taking M to be the upper or lower triangle of A (Gauss-Seidel iteration), or taking M to be the identity (Richardson iteration).

7.2 The sweeping picture

For *analysis*, the splitting picture is the "right" way to think about stationary iterations. In implementations, though, one often thinks not about splitting, but about *sweeping*. For example, consider the model tridiagonal system $Tu = h^2 b$ where T is a tridiagonal matrix with 2 on the diagonal and -1 on the first super- and subdiagonal. Written componentwise, this is

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 b_i, \text{ for } 1 \leq i \leq N$$

with $u_0 = u_{N+1} = 0$. A sweep operation takes each equation in turn and uses it to solve for one of the unknowns. For example, a Jacobi sweep looks like

```
% Jacobi sweep in MATLAB (U is u_0 through u_{N+1}, u_0 = u_{N+1} = 0)
for i = 1:N
  Unew(i+1) = ( h^2 * b(i) + U(i) + U(i+2) )/2;
end
U = Unew;
```

while a Gauss-Seidel sweep looks like

```
% G-S sweep in MATLAB (U is u_0 through u_{N+1}, u_0 = u_{N+1} = 0)
for i = 1:N
  U(i+1) = ( h^2 * b(i) + U(i) + U(i+2) )/2;
end
```

This formulation is equivalent to the splitting picture, but is arguably more natural, at least in the context of PDE discretizations.

7.3 Convergence examples

We usually need some structural characteristic to guarantee convergence of a stationary iteration. We gave two examples in class.

Jacobi and diagonal dominance Suppose A is strictly row diagonally dominant, i.e.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Then Jacobi iteration converges for linear systems involving A. The proof is simply that the iteration matrix $R = M^{-1}K$ by design has $\|R\|_\infty < 1$.

Gauss-Seidel and SPD problems Suppose A is symmetric and positive definite. Then Gauss-Seidel iteration converges for linear systems involving A. To prove this, note that each step in a Gauss-Seidel sweep is equivalent to updating x_i (holding all other entries fixed) to minimize the *energy* function

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b.$$

If x_* is the minimum energy point, then

$$\phi(x) - \phi(x_*) = \frac{1}{2}(x - x^*)^T A(x - x^*) = \frac{1}{2}\|x - x_*\|_A^2.$$

If $x^{(k)} \neq x_*$, then we can show that *some* step moving from $x^{(k)}$ to $x^{(k+1)}$ must reduce the energy, i.e.

$$\|e_{k+1}\|_A < \|e_k\|_A.$$

This is true regardless of the choice of $x^{(k)}$. Maximizing over all possible $x^{(k)}$ gives us that $\|R\|_A < 1$.

7.4 Problems

1. Consider using Richardson iteration to solve the problem $(I - K)x = b$ where $\|K\| < 1$ (i.e. $M = I$). If $x_0 = 0$, show that x_k corresponds to taking k terms in a truncated geometric series (a.k.a a Neumann series) for $(I - K)^{-1}$.

2. If A is strictly *column* diagonally dominant, Jacobi iteration still converges. Why?

3. Show that if A is symmetric and positive definite and x_* is a minimizer for the energy

   $$\phi(x) = \frac{1}{2}x^T Ax - x^T b$$

   then

   $$\phi(x) - \phi(x_*) = \frac{1}{2}(x - x_*)^T A(x - x_*).$$

4. The largest eigenvalue of the tridiagonal matrix $T \in \mathbb{R}^{n\times n}$ is $2 - O(n^{-2})$. Argue that the iteration matrix for Jacobi iteration therefore has spectral radius $\rho(R) = 1 - O(n^{-2})$, and therefore

   $$\log \rho(R) = -O(n^{-2})$$

Using this fact, argue that it takes $O(n^2)$ Jacobi iterations to reduce the error by a constant factor for this problem.

8 Krylov subspace methods

The m-dimensional *Krylov subspace* generated by A and b is

$$\mathcal{K}_m(A, b) = \text{span}\{b, Ab, \ldots, A^{m-1}b\} = \{p(A)b : p \in \mathcal{P}_{m-1}\}.$$

Krylov subspaces are phenomenally useful for two reasons:

1. All you need to explore a Krylov subspace is a subroutine that computes matrix-vector products.

2. An appropriately chosen Krylov subspace often contains good approximations to things we would like to compute (e.g. eigenvectors or solutions to linear systems). Moreover, we can use the connection to polynomials to reason about the quality of that space.

8.1 Arnoldi and Lanczos

While a Krylov subspace $\mathcal{K}_m(A, b)$ may be an attractive space, the power basis $b, Ab, \ldots, A^{m-1}b$ is not an attractive basis for that space. Because the power basis essentially corresponds to steps in a power iteration, successive vectors get ever closer to the dominant eigenvector for the system. Hence, the vectors become increasingly linearly dependent and the basis becomes increasingly ill-conditioned. What we would really like is an *orthonormal* basis for the nested Krylov subspaces. We can compute such a basis by the *Arnoldi process*

$$\begin{aligned}
q_1 &= b/\|b\| \\
v_{k+1} &= Aq_k \\
w_{k+1} &= v_{k+1} - \sum_{j=0}^{k} q_j h_{j,k+1}, & h_{j,k+1} &= q_j^T v_{k+1} \\
q_{k+1} &= w_{k+1}/h_{k+1,k}, & h_{k+1,k} &= \|w_{k+1}\|
\end{aligned}$$

That is, to get each new vector in turn we first multiply the previous vector by A, then orthogonalize.

If we write $Q_k = \begin{bmatrix} q_1 & \ldots & q_k \end{bmatrix}$, Arnoldi computes the decomposition

$$AQ_k = Q_k H_k + q_{k+1} h_{k+1,k} e_k^T$$

where H_k is a $k \times k$ upper Hessenberg matrix consisting of the coefficients that appear in the orthogonalization process. Note that $H_k = Q_k^T A Q_k$; hence, if A is symmetric, then H_k is both upper Hessenerg and symmetric, i.e. tridiagonal. In this case, we can compute the basis with a three-term recurrence, orthogonalizing only against two previous vectors at each step. The resulting algorithm is known as the *Lanczos* algorithm.

8.2 Krylov subspaces for linear systems

To solve a linear system with Krylov subspace we need two ingredients: a Krylov subspace and a method of choosing an approximation from that space. The two most common approaches are:

1. Choose the approximation $\hat{x}$ that gives the smallest residual (in the two norm). This is the basis of the GMRES algorithm, which is the default solver for nonsymmetric matrices, as well as the MINRES algorithm for symmetric indefinite problems.

2. If A is symmetric and positive definite, choose $\hat{x}$ to minimize the energy function $\phi(x)$ over all x in the space, where

$$\phi(x) = \frac{1}{2} x^T A x - x^T b.$$

   The equation $Ax_* = b$ is exactly the equation for a stationary point (aka a critical point), and the only critical point of ϕ is the global minimum. The energy minimization strategy is the basis for the method of conjugate gradients (CG), which is the default Krylov subspace solver for SPD problems.

We did not actually derive CG in lecture; even more than with the QR iteration, the magical-looking derivation of CG tends to obscure the fundamental simplicity of the approach. We did briefly discuss the properties of the error in CG, namely that the algorithm minimizes the energy norm of the error $\|x - x_*\|_A^2$ and the inverse energy norm of the residual, i.e. $\|Ax - b\|_{A^{-1}}^2$.

8.3 Convergence behavior

Let's consider the Krylov subspaces $\mathcal{K}_m(A, b)$ and the problem of approximating $x = A^{-1}b$ by some $\hat{x} \in \mathcal{K}_m(A, b)$. How good can the best approximation

from a given Krylov subspace be? Note that any element of $\mathcal{K}_m(A, b)$ can be associated with a polynomial of degree at most $m - 1$, i.e.

$$\hat{x} = p(A)b, \quad p \in \mathcal{P}_{m-1}.$$

The difference between $\hat{x}$ and x is

$$\hat{x} - x = (p(A) - A^{-1})b,$$

or, assuming A is diagonalizable,

$$\hat{x} - x = V(p(\Lambda) - \Lambda^{-1})V^{-1}b.$$

Taking norms, we have

$$\|\hat{x} - x\| \leq \kappa(V) \max_{\lambda} |p(\lambda) - \lambda^{-1}| \, \|b\|.$$

That is, apart from the annoying issue of the conditioning of the eigenvectors, we can reduce the problem of bounding the error of the best approximation to the problem of finding a polynomial $p(z)$ that best approximates z^{-1} on the set of eigenvalues.

For the SPD case, one can bound the best-estimate behavior using a polynomial approximation to z^{-1} on an interval $[\lambda_{\min}, \lambda_{\max}]$. The argument involves a pretty use of Chebyshev polynomials – see, for example, the theorem on p. 187 of the textbook – but this bound is often quite pessimistic in practice. The actual convergence depends on how clustered the eigenvalues are, and also on the nature of the right hand side vector b.

8.4 Preconditioning

Krylov subspace methods are typically *preconditioned*; that is, rather than solving

$$Ax = b$$

one solves (at least notionally)

$$M^{-1}Ax = M^{-1}b$$

where applying M^{-1} (the *preconditioner solve*) is assumed to be relatively inexpensive. Typical choices for the preconditioner include the M matrix from

a stationary iteration, solves with approximate factorizations, and methods that take advantage of the physical meaning of A (e.g. multigrid methods). A good preconditioner tends to cluster the eigenvalues of $M^{-1}A$. For CG, both the preconditioner and the matrix A must be SPD. Choosing a good preconditioner tends to be as much an art as a science, with the best preconditioners often depending on an understanding of the particular application at hand.

8.5 Krylov subspaces for eigenvalue problems

If $AQ_k = Q_k H_k + h_{k,k+1} q_{k+1}$ is an Arnoldi decomposition, the eigenvalues of H_k are often used to estimate eigenvalues of A. The columns of Q_k span a Krylov subspace that may be generated using A or using some transformed matrix (e.g. $(A-\sigma I)^{-1}$ where σ is some shift of interest). Hence, the Krylov subspace of A contains k steps of a power iteration, possibly with a shift-invert transformation, and should have at least the approximating power of that iteration. Note that if $H_k v = v\lambda$, then $\hat{x} = Q_k v$ is an approximate eigenvector with

$$A\hat{x} = \hat{x}\lambda + h_{k,k+1} q_{k+1} e_k^T v,$$

i.e.

$$\|A\hat{x} - \hat{x}\lambda\| \leq |h_{k,k+1}||v_k|.$$

The MATLAB command `eigs` computes a few of the largest eigenvalues, smallest eigenvalues, or eigenvalues near some shift via Arnoldi (or Lanczos in the case of symmetric problems).

8.6 Problems

1. Suppose A is symmetric positive definite and $\phi(x) = x^T Ax/2 - x^T b$. Show that over all approximations of the form $\hat{x} = Uz$, the one that minimizes ϕ satisfies $(U^T AU)z = U^T b$.

2. Suppose A is SPD and ϕ is defined as in the previous question. If $\hat{x} = Uz$ minimizes the energy of $\phi(\hat{x})$, show that $\|Uz - \hat{x}\|_A^2$ is also minimal.

3. Suppose A is nonsingular and has k distinct eigenvalues. Argue that $\mathcal{K}_k(A, b)$ contains $A^{-1}b$.

4. Argue that the residual after k steps of GMRES with a Jacobi preconditioner is no larger than the residual after k steps of Jacobi iteration.

5. If A is symmetric, the largest eigenvalue is the maximum value of the Rayleigh quotient $\rho_A(x)$. Show that computing the largest eigenvalue of $\rho_T(z)$ where $T = Q^T AQ$ is equivalent to maximizing $\rho_A(x)$ over x s.t. $x = Qz$. The largest eigenvalue of T is always less than or equal to the largest eigenvalue of A; why?

9 Iterations in 1D

We started the class with a discussion of equation solving in one variable. The goal was to get you accustomed to thinking about certain ideas (fixed point iteration, Newton iteration) in a less complicated setting before moving on to the more general setting of systems of equations.

9.1 Fixed point iteration and convergence

Fixed point iteration A *fixed point* of a function $g : \mathbb{R} \to \mathbb{R}$ is a solution to the equation

$$x = g(x).$$

A one-dimensional *fixed point iteration* is an iteration of the form

$$x_{k+1} = g(x_k).$$

Convergence analysis Our standard recipe for analyzing the convergence of a fixed point iteration is

1. Subtract the fixed point equation from the iteration equation to get an iteration for the error.
2. Linearize the error iteration to obtain a tractable problem that describes the behavior of the error for starting points "close enough" to the initial point.

More concretely, if we write the error at step k as $e_k = x_k - x_*$, then subtracting the fixed point equation from the iteration equation gives

$$e_{k+1} = g(x_* + e_k) - g(x_*).$$

Assuming g is differentiable, we have

$$e_{k+1} = g'(x_*)e_k + O(|e_k|^2).$$

If $|g'(x_*)| < 1$, then the fixed point is *attractive*: that is, the iteration will converge to x_* for starting points close enough to x_*.

Plotting convergence When g is differentiable and $0 < |g'(x_*)| < 1$, fixed point iteration is *linearly convergent*. That is, we have

$$|e_k| \approx |e_0||g'(x_*)|^k,$$

and so when we plot the error on a semi-logarithmic scale, we see

$$\log |e_k| \approx k \log |g'(x_*)| + \log |e_0|,$$

i.e. the (log scale) errors fall approximately on a straight line. Of course, this convergence behavior only holds until rounding error starts to dominate! When $g'(x) = 0$, we have *superlinear* convergence.

9.2 Newton's method

Newton's method Newton's method for solving $f(x) = 0$ is:

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k).$$

We derive the method by taking the linear approximation

$$f(x_k + p) \approx f(x_k) + f'(x_k)p$$

and choosing the update p such that the approximation is zero; that is, we solve the linear equation

$$f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0.$$

Local convergence Assuming f is twice differentiable, the *true* solution x_* satisfies

$$f(x_k) + f'(x_k)(x_* - x_k) = O(|x_k - x_*|^2)$$

Writing $e_k = x_k - x_*$ and subtracting the true solution equation from the iteration equation gives us

$$f'(x_k)e_{k+1} = O(|e_k|^2).$$

If $f'(x_k)$ is bounded away from zero, we then have that $|e_{k+1}| = O(|e_k|^2)$, or *quadratic convergence*. Plotting quadratic convergence on a semi-logarithmic plot gives us a shape that looks like a downward-facing parabola, up to the point where roundoff errors begin to dominate (which often only takes a few steps).

Initial guesses Newton's iteration is *locally convergent* – it is only guaranteed to converge from starting points that are sufficiently close to the solution. Hence, a good initial guess can be critically important. Getting a good initial guess frequently involves reasoning about the problem in some application-specific way, approximating the original equations in a way that yields something analytically tractable.

Secant iteration One of the annoying features of Newton's iteration is that it requires that we compute derivatives. Of course, we can always replace the derivatives by a *finite difference* approximation:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

This leads to the *secant iteration*

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

The convergence analysis for secant iteration is slightly more complicated than that for Newton iteration, but the iteration is certainly superlinear.

9.3 Bisection

Newton's iteration – and most other fixed point iterations – generally only converge if the initial guess is good enough. An alternate approach of *bisection* converges slowly but consistently to a solution of $f(x) = 0$ in an interval $[a, b]$ assuming that $f(a)f(b) < 0$ and f is continuous on $[a, b]$. Bisection relies on the idea that if f changes sign between the endpoints of an interval, then there must be a zero somewhere in the interval. If there is a sign change between $f(a)$ and $f(b)$ and $c = (a + b)/2$, then there are three possibilities:

- $f(c) = 0$ (in which case we're done).
- $f(c)$ has the same sign as $f(a)$, in which case $[c, b]$ contains a zero of f.
- $f(c)$ has the same sign as $f(b)$, in which case $[a, c]$ contains a zero of f.

Thus, we have an interval half the size of $[a, b]$ that again contains a solution to the problem.

Bisection produces a sequence of ever-smaller intervals, each guaranteed to contain a solution. If we know there is a solution in the interval $[a, b]$, we usually take $x = (a + b)/2$ as the approximation; barring any additional information about the solution, this is the approximation in the interval that minimizes the worst-case error. Hence, if $[a, b]$ is the initial interval and $x_0 = (a+b)/2$ is the initial guess, then the initial error bound is $|x_0 - x_*| \leq |b-a|/2$. For successive iterations, the error bound is $|x_k - x_*| \leq |b - a|/2^{k+1}$.

9.4 Combined strategies

Newton and secant iterations are fast but dangerous. Bisection is slow but steady. We would like the best of both worlds[2]: superlinear convergence close to the solution, with steady progress even far from the solution. *Brent's algorithm* (the algorithm used in MATLAB's `fzero`) does this. Of course, Brent's algorithm still requires an initial bracketing interval, but it is otherwise about as bulletproof as these things can possibly be.

9.5 Sensitivity analysis

Suppose $\hat{x}$ is an approximation of x_* such that $f(x_*) = 0$, where f is at least continuously differentiable. How can we evaluate the quality of the estimate? The simplest thing is to check the *residual* error $|f(\hat{x})|$. In some cases, this is enough – we really care about making $|f|$ small, and any point that satisfies this goal will suffice. In other cases, though, we care about the *forward* error $|\hat{x} - x_*|$. Of course, if we have an estimate of the derivative $f'(x_*)$, then we can use a Taylor expansion to estimate

$$|\hat{x} - x_*| \approx |f(\hat{x})|/|f'(x_*)| \approx |f(\hat{x})|/|f'(\hat{x})|.$$

You should recognize this as saying that the Newton correction starting from $\hat{x}$ is a good estimate of the error. Of course, if we are able to compute both $f(\hat{x})$ and $f'(\hat{x})$ accurately, it may make sense to compute a Newton update directly! One of the standard termination criteria for Newton iteration involves using the size of the last correction as a (very conservative) estimate of the error at the current step.

There are two caveats here. First, there is often some rounding error in our computation of f, and we may need to take this into account. Assuming

[2]Does this sound like a blurb for a bad romance novel?

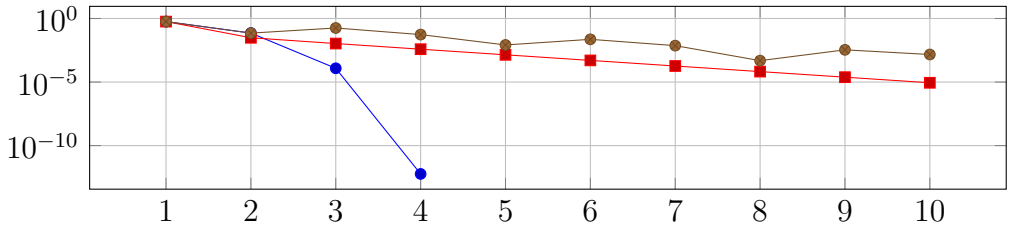

Figure 1: Convergence of Newton iteration, a fixed point iteration, and bisection for $\cos(x) = 0$.

we can compute $|f'(\hat{x})|$ reasonably accurately, and $|\hat{f}(\hat{x}) - f(\hat{x})| < \delta$ where $\hat{f}$ is the value of $f(\hat{x})$ computed with roundoff, then we have

$$|\hat{x} - x_*| \lesssim (|\hat{f}(\hat{x})| + \delta)/|f'(\hat{x})|.$$

Thus, $\delta/|f'(\hat{(}x)|$ estimates the best error we could reasonably expect.

The second caveat is that sometimes $f'(x_*) = 0$, or is incredibly close to zero. In this case, we can still pursue the same type of analysis, but we need to take additional terms in the Taylor expansion. Or, if we know in advance that $f'(x_*) = 0$, we may choose to find a root of f' rather than finding a root of f.

9.6 Problems

1. Consider the fixed point iteration $x_{k+1} = g(x_k)$ and assume x_* is an attractive point. Also assume $|g''(x)| < M$ everywhere. We know that the iteration converges to x_* from "close enough" starting points; show that a sufficient condition for convergence is

$$|x_0 - x_*| < \frac{2(1 - g'(x_*))}{M}.$$

2. What is Newton's iteration for finding $\sqrt{a}$?

3. Consider the fixed-point iteration $x_{k+1} = x_k + \cos(x_k)$. Show that for x_0 near enough to $x_* = \pi/2$, the iteration converges, and describe the convergence behavior.

4. The graphs shown in Figure 1 show the convergence of Newton's iteration starting from $x_0 = 1$, the fixed point iteration $x_{k+1} = x_k + \cos(x_k)/x_k$ starting from $x_0 = 1$ and bisection starting from $[0, 2]$ to the solution of $\cos(x) = 0$. Which plot corresponds to which method? How can you tell?

5. Find an example of a function with a unique zero and a starting value such that Newton's iteration does not converge.

6. Suppose f has a sign change for between $a = 1000$ and $b = 1001$. How many steps of bisection are required to obtain a *relative* error of 10^{-6}?

10 Multivariate nonlinear problems

In the last part of the class, we moved from problems in numerical linear algebra (simple for one reason) and nonlinear equations in one variable (simple for another reason) to problems involving nonlinear equation solving and optimization with many variables. The picture is similar to the one we saw in 1D, but more complicated both due to the fact that we're now dealing with several dimension and due to the fact that our safe fallback method in 1D (bisection) does not generalize nicely to higher-dimensional problems.

10.1 Nonlinear equations and optimization

We are interested in two basic problems:

1. Given $F : \mathbb{R}^n \to \mathbb{R}^n$ a twice-differentiable function, solve $F(x_*) = 0$.
2. Find a local minimum of $\phi : \mathbb{R}^n \to \mathbb{R}$ a differentiable function with three derivatives.

The two problems are not so far apart: finding a zero of F is equivalent to minimizing $\|F\|^2$, while a local minimum of ϕ occurs at a stationary point, i.e. x_* satisfying the nonlinear equation $\nabla\phi(x_*) = 0$. The optimization perspective is particularly useful for analyzing "globalized" iterations (trust region methods and line search methods).

10.2 Fixed point iterations

As in one space dimension, our basic tool is a fixed point iteration:

$$x_{k+1} = G(x_k)$$

converging to a stationary point

$$x_* = G(x_*).$$

Letting $e_k = x_k - x_*$, we have

$$e_{k+1} = G'(x_*)e_k + O(\|e_k\|^2),$$

and so we have convergence for small enough e_0 when $\rho(G'(x_*)) < 1$. If $G'(x_*) = 0$, we have superlinear convergence.

10.3 Newton's method for systems

Newton's method for solving nonlinear systems is $x^{(k+1)} = x^{(k)} + p^{(k)}$ where

$$F(x^{(k)}) + F'(x^{(k)})p^{(k)} = 0.$$

Put differently,

$$x^{(k+1)} = x^{(k)} - F'(x^{(k)})^{-1}F(x^{(k)}).$$

The iteration is quadratically convergent if $F'(x^*)$ is nonsingular.

As an example, consider the problem of finding the intersection between the unit circle and the unit hyperbola, i.e., finding a zero of

$$F(x, y) = \begin{bmatrix} x^2 + y^2 - 1 \\ xy - 1 \end{bmatrix}.$$

The Jacobian of F is

$$F' = \begin{bmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

Note that the Jacobian is singular when $x = y$; that is, not only is the Newton iteration only locally convergent, but the Newton step may be impossible to solve at some points.

10.4 Newton's method for optimization

For optimization problems, Newton's method involves finding a stationary point, i.e. a point at which the gradient $\nabla\phi$ is zero. However, a stationary point could also be a local maximum or a saddle point, so for optimization we typically only use Newton steps if we can guarantee that they will decrease the objective function. Otherwise we modify the Newton iteration to guarantee that we choose a *descent direction* for our step.

The Jacobian of the gradient is the matrix of second derivatives H_ϕ, also known as the Hessian matrix. At a local minimum, the Hessian must at least be positive semidefinite; at and near a *strong* local minimum, the Hessian must be positive definite. A pure Newton step for optimization is

$$p_k = -H_\phi(x_k)^{-1}\nabla\phi(x_k)$$

We can think of this as running Newton on the gradient system or as finding a stationary point of the local quadratic approximation

$$\phi(x_k + p) \approx \phi(x_k) + \nabla\phi(x_k)^T p + \frac{1}{2} p^T H_\phi(x_k)^{-1} p.$$

We would like to guarantee that steps move "downhill", i.e. that p_k is a descent direction:

$$\nabla\phi(x_k)^T p_k < 0.$$

When $H_\phi(x_k)$ is positive definite, so is $H_\phi(x_k)^{-1}$, and so in this case $\nabla\phi(x_k)^T p_k = -\nabla\phi(x_k)^T p_k H_\phi(x_k)^{-1} \nabla\phi(x_k) < 0$ and we do have a descent direction. When H_ϕ is indefinite, we typically modify our step to guarantee a descent direction. That is, we consider an iteration with steps

$$p_k = -H_k^{-1} \nabla\phi(x_k)$$

where H_k is a positive definite scaling matrix, chosen to be the Hessian when that is positive definite and something close to the Hessian (e.g. $H_k = H_\phi(x_k) + \eta I$, or something based on a modified factorization of $H_\phi(x_k)$).

10.5 Gauss-Newton and nonlinear least squares

The *nonlinear least squares* problem is to minimize

$$\phi(x) = \frac{1}{2}\|F(x)\|^2 = \frac{1}{2} F(x)^T F(x), \quad F : \mathbb{R}^n \to \mathbb{R}^m.$$

The gradient of ϕ is

$$\nabla\phi(x) = J(x)^T F(x), \quad J(x) = F'(x),$$

and the Hessian of ϕ is the matrix with entries

$$H_{\phi,ij} = (J^T J)_{ij} + \sum_k \frac{\partial^2 F_j}{\partial x_i \partial x_k} F_k(x).$$

The latter term is often a pain to compute, and when the least squares problem can be solved so that F has a small residual (i.e. $F(x_*) \approx 0$), we might want to discard it. This leads us to the *Gauss-Newton* iteration

$$x_{k+1} = x_k + p_k, \quad p_k = -(J^T J)^{-1}(J^T F) = -J^\dagger F.$$

Alternately, we can think of the Gauss-Newton iteration as minimizing the linearized residual approximation

$$F(x_k + p) \approx F(x_k) + J(x_k)p.$$

When $n = m$ and the Jacobian is nonsingular, Gauss-Newton iteration is the same as Newton iteration on the equation $F(x) = 0$. Otherwise, Gauss-Newton is *not* the same as Newton iteration for the least squares optimization problem, and it does not converge quadratically unless the solution satisfies $F(x_*) = 0$. On the other hand, the linear convergence is often more than adequate (and it can be accelerated if needed).

10.6 Problems with Newton

There are a few drawbacks to pure Newton (and Gauss-Newton) iterations.

- The iterations are only locally convergent. We therefore want either good initial guesses (application specific) or approaches that *globalize* the iterations, expanding the region in which they converge. Often, we need to use both strategies.

- Newton iteration involves computing first and second derivatives. This is fine for simple functions of modest size, and in principle automated differentiation tools can compute the relevant derivatives for us if we have access to the source code for a program that computes the function. On the other hand, we don't always have that luxury – someone may hand us a "black box" function for which we lack source code, for example – and so sometimes it is difficult to get the relevant derivatives. Even if computing the derivatives is not difficult, it may be unappealingly expensive.

- Even when we can get all the relevant derivatives, Newton iteration requires factoring a new matrix (Jacobian or Hessian) at every step. The linear algebra costs may again be expensive.

For all these reasons, Newton iteration is not the ending point of linear solvers, but a starting point.

10.7 Approximating Newton

There are two main approaches to approximating Newton steps. First, one can use an *inexact* Newton approach, solving the Newton linear systems approximately. For example, in *Newton-Krylov* methods, we would apply a (preconditioned) Krylov subspace solver to the linear systems, and we might choose to terminate the solver while the residual for the linear system was not completely zero. That is, there is a tradeoff between how many linear iteration steps we take and how many nonlinear iteration steps we take. We did some analysis on a homework problem to show that for an optimization problem in which we solve the linear system with residual r, if $\kappa(H)\|r\| < \|\nabla\phi\|$ then we are at least guaranteed a descent direction.

Inside a Newton-Krylov solver, one repeatedly computes matrix vector products with the Jacobian matrix J. It is not always necessary to compute the Jacobian explicitly to form these matrix-vector products. Indeed, it may not even be necessary to compute the Jacobian analytically; note that

$$Jv = F'(x)v = \frac{\partial F}{\partial v}(x) = h^{-1}(F(x+hv) - F(x)) + O(h).$$

Of course, this still leaves the question of how to choose the finite difference step size h!

The second family of methods are *quasi* Newton methods, in which we use an approximation of the Jacobian or Hessian. The most popular quasi-Newton approach is the BFGS algorithm (and the L-BFGS variant), which builds up a Hessian approximation by updating an initial approximation with information obtained by looking at successive iterates. We mentioned these methods briefly in lecture, and the book mentions them as well, but did not go into detail.

10.8 Other first-order methods

In addition to inexact Newton methods and quasi-Newton methods, there are a plethora of first-order methods that don't look particularly Newton like, at least at first glance. For optimization, for example, classic steepest descent methods are usually introduced before Newton methods (though steepest descent is often very slow). There are also methods related to classical stationary iterations for linear systems. For example, the cyclic coordinate descent method for optimization considers each variable in turn and adjusts it to

reduce the objective function value. When applied to a quadratic objective function, cyclic coordinate descent becomes Gauss-Seidel iteration.

10.9 Globalization: line search

So far, we have only discussed how to choose an update p that "looks promising" (i.e. is a descent direction for an optimization problem). This may involve an expensive subcomputation as in Newton's method, or a simple-minded choice like $p = \pm e_j$ as in cyclic coordinate descent. But just because an update p_k looks promising based on a simplified linear or quadratic model of the objective function does not mean that $x_{k+1} = x_k + p_k$ will actually be better than x_k; indeed, the objective function at x_{k+1} (or the norm of the residual in the case of equation solving) may be *worse* than it was at x_k.

A *line search* strategy uses the update

$$x_{k+1} = x_k + \alpha_k p_k$$

for some $0 < \alpha_k < 1$ chosen to guarantee a reduction in the objective function value. A typical strategy is to start by trying $\alpha_k = 1$, then cut α_k in half if the step does not look "good enough" according to some criterion. A common criterion is the *Armijo* condition, which says that we need to make at least some fixed fraction of the progress predicted by the linear model; that is, we require

$$\phi(x_{k+1}) - \phi(x_k) < \eta \alpha_k \nabla \phi(x_k)^T p$$

for some constant $\eta < 1$. A simpler criterion (and one that can cause non-convergence, at least in principle) is to simply insist

$$\phi(x_{k+1}) < \phi(x_k).$$

With an appropriate line search strategy and a condition on the search direction, we can obtain optimization strategies that always converge to a local minimum, unless the objective function is asymptotically flat or decreasing in some direction so that the iteration can "escape to infinity." The appropriate *Wolfe conditions* are spelled out in detail in optimization classes, but we pass over them here.

10.10 Globalization: trust regions

In line search, we first pick a direction and then decide how far to go in that direction to guarantee progress. But if we proposed a Newton step and line

search told us we couldn't take it, that means that the step fell outside the range where we could really trust the quadratic approximation on which the step was based. The idea of *trust region* methods is to try to choose a step that minimizes the function within some region in which we trust the model. If we fail to make adequate progress, we reduce the size of our trust region; if the model proves highly accurate, we might expand the trust region.

The trust region subproblem is to minimize a quadratic approximation

$$\psi(x_k + p) = \phi(x_k) + \nabla\phi(x_k)^T p + \frac{1}{2} p^T H p$$

subject to the constraint that $\|p\| \leq \rho$ for some given ρ. At the solution to this problem, we satisfy the critical point equation

$$(H + \lambda I)p = -\nabla\phi(x)$$

for some $\lambda \geq 0$. If the ordinary Newton step falls inside the trust region, then $\lambda = 0$ and the constraint is said to be *inactive*. Otherwise, we choose λ so that $\|p\| = \rho^2$. Alternately, we may focus on λ, leaving the trust region radius ρ implicit.

Applied to the Gauss-Newton iteration, the trust region approach yields subproblems of the form

$$(J^T J + \lambda I)p = -J^T F.$$

You may recognize this as equivalent to solving the least-squares problem for the Gauss-Newton update with Tikhonov regularization. This *Levenberg-Marquardt* update strategy actually pre-dates the development of the trust region framework.

Because it potentially involves searching for an appropriate λ, the trust region subproblem is more expensive than an ordinary Newton subproblem, which may already be rather expensive. Because of this, the trust region subproblem is usually only solved approximately (using the colorfully-named dogleg strategy, for example).

While trust region methods are somewhat more complicated to implement than line search methods, they frequently are able to solve problems with fewer function evaluations.

10.11 Globalization: continuation methods

Even with globalization, Newton and Newton-like iterations will not necessarily converge quickly without a good initial guess. Moreover, if there are

multiple solutions to a problem, Newton may converge to the "wrong" (in the light of the application) solution without a good initial guess. Most of the time, finding a good initial guess is a matter of manipulating application-specific approximations. There is, however, one general-purpose strategy that often works: continuation.

The idea of continuation methods is to study not one function, but a parametric family. The parameter may be a physically meaningful quantity (e.g. magnitude of a force, displacement, voltage, etc); or it may be a purely artificial construct. By gradually changing the parameter, one can move from an easy problem instance to a hard problem instance in a controlled way. Continuation methods follow a *predictor-corrector* pattern: given a solution at parameter value s_k, one first *predicts* the solution at parameter value s_{k+1} and then *corrects* the prediction using some locally-convergent iteration. For example, the predictor might be the trivial predictor (i.e. use as an initial guess at s_{k+1} the converged solution at s_k) and a Newton corrector. If the iteration diverges, one can always try again with a shorter step size.

The other advantage provided by continuation methods is that we are not forced to use just one parameter. We can choose between parameters, or even make up a new parameterization (this is the basis for *pseudo-arclength* strategies, which I mentioned in a throw-away sentence one lecture). Often problems that seem difficult when using one parameter are trivial with respect to a different parameter.

10.12 Problems

1. Write a MATLAB code to estimate α and x such that $y = \alpha x$ is tangent to $y = \cos(x)$ near $x_0 = n\pi$ for $n > 0$. I recommend writing two equations (matching function values and matching derivatives) in two unknowns (the intersection x and α) and applying Newton. What is a good initial guess?

2. Write a MATLAB code to find a critical point of $\phi(x, y) = -\exp(x^2 + y^2)(x^2 + y^2 - 2(ax + by) + c)$ using Newton's iteration.

3. Write a MATLAB fragment to minimize $\sum_j \exp(r_j^2) - 1$ where $r = Ax - b$. Use a Gauss-Newton strategy (no need to bother with safeguards like line search).

4. Consider the fixed point iteration

$$x_{k+1} = x_k - A^{-1}F(x_k)$$

where F has two continuous derivatives and A is some (possibly crude) approximation to the Jacobian of F at the solution x_*. Under what conditions does the iteration converge?

5. Suppose x_* is a strong local minimum for ϕ, i.e. $\nabla\phi(x_*) = 0$ and $H_\phi(x_*)$ is positive definite. For starting points x_0 close enough to x_*, Newton with line search based on the Armijo condition behaves identically to an unguarded Newton iteration with no line search. Why?

6. Argue that for large enough λ, $p = -(H_\phi(x) + \lambda I)^{-1}\nabla\phi(x)$ is guaranteed to be a descent direction, assuming x is not a stationary point.

7. Suppose $F : \mathbb{R}^n \times \mathbb{R} \mapsto \mathbb{R}^n$ (i.e. $F = F(x, s)$). If we solve $F(x(s), s) = 0$ for a given s using Newton's iteration and we are able to compute $\partial F/\partial s$ in at most $O(n^2)$ time, we can compute dx/ds in $O(n^2)$ time. How?

8. Describe a fast algorithm to solve

$$Ax = b(x_n)$$

where $A \in \mathbb{R}^{n\times n}$ is a fixed matrix and $b : \mathbb{R} \to \mathbb{R}^n$ is twice differentiable. Your algorithm should cost $O(n^2)$ per step and converge quadratically.

11 Constrained problems

Most of our discussion of optimization involved *unconstrained* optimization, but we did spend two lectures talking about the constrained case (and the overview section in the book is pretty reasonable). The constrained problem is

$$\text{minimize } \phi(x) \text{ s.t. } x \in \Omega$$

where $\Omega \subset \mathbb{R}^n$ is usually defined in terms of a collection of equations and inequalities

$$\Omega = \{x \in \mathbb{R}^n : g(x) = 0 \text{ and } h(x) \leq 0\}.$$

We discussed three basic approaches to constraints: elimination, barriers and penalties, and Lagrange multipliers. Each can be used for both theory and as the basis for algorithms.

11.1 Constraint elimination

The idea behind *constraint elimination* is that a set of equality constraints $g(x) = 0$ implicitly define a lower-dimensional surface in $\mathbb{R}^n$, and we can write the surface as a parametric function $x = F(y)$ for $y \in \mathbb{R}^p$ for $p < n$. Then the constrained problem in x is an unconstrained problem in y:

$$\text{minimize } \phi(x) \text{ s.t. } g(x) = 0 \quad \equiv \quad \text{minimize } \phi(F(y)).$$

The chief difficulty with constraint elimination is that we have to find a parameterization of the solutions to the constraint equations. This is hard in general, but is straightforward when the constraints are linear: $g(x) = A^T x - b$. In that case we can use a full (not economy) QR decomposition of A to parameterize all feasible x as

$$x = F(y) = Q_1 R_1^{-T} b + Q_2 y.$$

In terms of the linear algebra, linear constraint elimination has some attractive features: if ϕ is convex, then so is $\phi \circ F$; and the Hessians of $\phi \circ F$ are better conditioned than those of ϕ. On the other hand, even linear constraint elimination will generally destroy sparsity of the problem.

Constraint elimination is also an attractive option for some classes of problems involving linear *inequality* constraints, particularly if those constraints are simple (e.g. elementwise non-negativity of the solution vector).

One can either solve the inequality-constrained problem by an iteration that incrementally improves estimates of the active set and solves equality-constrained subproblems. Alternately, one might use a parameterization that removes the need for the inequality constraint; for example, we can parameterize $\{x \in \mathbb{R} : x \geq 0\}$ as $x = y^2$ where y is unconstrained. However, while this is simple to think about, it may not be the best approach numerically. For example, an objective that is convex in x may no longer by convex in y.

Ideally, you should understand constraint elimination well enough to implement it for linear constraints and simple objectives (e.g. quadratic objectives and linear least squares).

11.2 Penalties and barriers

The idea behind penalties and barriers is that we add a term to the objective function to (approximately) enforce the constraint. For penalty methods, we add a term that is positive if the constraint is violated, and grows quickly as the level of violation becomes worse. For barrier methods, we add a term that is positive near the boundary and blows up to infinity at (and outside) the boundary. Often the penalty or barrier depends on some *penalty parameter* μ, and the exact solution is recovered in the limit as $\mu \to 0$. A common example is to enforce an equality constraint by a quadratic penalty:

$$x(\mu) = \text{argmin } \phi(x) + \frac{1}{2\mu} g(x)^2.$$

As $\mu \to 0$, $x(\mu)$ converges to x_*, a constrained minimizer of ϕ subject to $g(x) = 0$.

Unfortunately the conditioning of the Hessian scales like $O(\mu^{-1})$, so the problems become increasingly numerically sensitive with larger μ. One way of dealing with this sensitivity is to get increasingly better initial guesses by tracing $x(\mu)$ through a sequence of ever smaller μ values. One can do the same thing for inequality constraints; this idea, together with a logarithmic barrier, is the main tool in *interior point* methods.

In the case of *exact* penalties, the exact solution may be recovered for nonzero μ; but exact penalty methods often result in a non-differentiable objective.

Ideally, you should understand the basic idea behind penalties at the level where you could implement a simple penalty method (or barrier method). You should also understand enough about sensitivity and conditioning of

linear systems to understand how well a quadratic penalty with a quadratic objective does at approximating the solution to an equality-constrained least squares problem.

11.3 Lagrange multipliers

Just as solutions to unconstrained optimization problems occur at stationary points, so do the solutions to constrained problems; we just have to work with a slightly different functional. To minimize $\phi(x)$ subject to $g(x) = 0$ and $h(x) \leq 0$, we form the *Lagrangian*

$$L(x, \lambda, \mu) = \phi(x) + \lambda^T g(x) + \mu^T h(x).$$

where λ and μ are vectors of *Lagrange multipliers*. In class, we described these multipliers by a physical analogy as forces that enforce the constraint. At a critical point, we have the KKT conditions:

$$\begin{aligned} \frac{\partial L}{\partial x} &= 0 && \text{(stationarity)} \\ \mu^T h(x) &= 0 && \text{(complementary slackness)} \\ g = 0 \text{ and } h &\geq 0 && \text{(primal feasibility)} \\ \mu &\geq 0 && \text{(dual feasibility)} \end{aligned}$$

If $\mu_i > 0$, we say the ith inequality constraint is *active*.

There are two major ways of dealing with constraints in solvers. *Active set* methods guess which constraints are active, and then solve an equality constrained sub-problem. If the guess was wrong, one adjusts the choice of active constraints and solves another equality constrained sub-problem. In contrast, methods based on *penalties* or *barriers* deal with inequality constraints by adding a cost to ϕ that gets big as one gets close to the boundary of the feasible region, either from the inside (barriers) or from the outside (penalties). *Interior point methods*, which are among the most popular and successful constrained optimization solvers, use a parameterized barrier function together with a continuation strategy to drive the barrier parameter toward a small value that yields hard (ill-conditioned) subproblems but with accurate results.

You should ideally understand the special case of setting and running Newton iteration for a Lagrangian function associated with an equality-constrained optimization problem. Determining the active set for the inequality-

constrained case is trickier, and probably would not be a great source of questions for an exam.

11.4 Problems

1. Describe three algorithms to minimize $\|Ax - b\|^2$ subject to $Cx = d$, where $A \in \mathbb{R}^{m \times n}$, $m > n$ and $C \in \mathbb{R}^{p \times m}$, $p < m$: one based on constraint elimination, one a quadratic penalty formulation, and one via Lagrange multipliers.

2. Write a system of equations to characterize the minimum of a linear function $\phi(x) = v^T x$ on the ball $x^T M x = 1$.

3. Suppose A is symmetric and positive definite and consider the quadratic objective
$$\phi(x) = \frac{1}{2} x^T A x - x^T b.$$
Suppose the global optimum has some negative components. We can find the constrained optimum be solving a sequence of logarithmic barrier problems
$$\text{minimize } \phi(x) - \mu \sum_i \log(x_i).$$
Write a Newton iteration to solve this problem.