Background Plus a Bit

For this class, I assume you know some linear algebra and multivariable calculus. You should also know how enough programming to *pick up* how to write and debug simple Julia scripts (the language syntax is similar to MATLAB). But there are some things you may never have forgotten that you will need for this class, and there are other things that you might not have learned. This set of notes will describe some of these things. It is fine if you do *not* know all this material! Ask questions if you see things you do not know or understand, and do not feel bad about asking the same question more than once if you get confused or forget during class.

1 Linear algebra background

In what follows, I will mostly consider 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 map between the abstract and concrete pictures of vector spaces 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 $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$. 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 $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots)$ is the matrix of *singular values*. The singular values are non-negative; by convention, they should be in ascending order.

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

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

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 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.

4 Julia background

Julia is a relatively young language initially released in 2012; the first releases of MATLAB and Python were 1984 and 1991, respectively. It has become increasingly popular for scientific computing and data science types of problems for its speed, simple MATLAB-like array syntax, and support for a variety of programming paradigms. We will provide pointers to some resources for getting started with Julia (or going further with Julia), but here we summarize some useful things to remember for writing concise codes for this class.

Note: These notes were adapted from a similar set of notes for MATLAB once upon a time. They will almost surely be refined and extended, perhaps even over the course of this semester.

Building matrices and vectors Julia supports general multi-dimensional arrays. Though the behavior can be changed, by default, these use one-based indexing (like MATLAB or Fortran, unlike Python or C/C++). Indexing uses square brackets (unlike MATLAB), e.g.

```
x = v[1]
y = A[1,1]
```

By default, we think of a one-dimensional array as a column vector, and a two-dimensional array as a matrix. We can do standard linear algebra operations like scaling (`2*A`), summing like types of objects `v1+v2`), and matrix multiplication `A*v`.

The expression

```
w = v'
```

represents the adjoint of the vector `v` (i.e. the conjugate transpose). The tick operator also gives the transpose of a matrix. We note that the tick operator in Julia does not actually copy any storage; it just gives us a re-interpretation of the argument. This shows up, for example, if we write

```
v = [1, 2] # v is a 2-element Vector{Int64} containing [1, 2]
w = v'     # w is a 1-2 adjoint(::Vector{Int64}) with eltype Int64
v[2] = 3   # Now v contains [1, 3] and w is the adjoint [1, 3]'
```

Julia gives us several standard matrix and vector construction functions.

```
Z = zeros(n) # Length n vector of zeros
Z = zeros(n,n) # n-by-n matrix of zeros
b = rand(n)  # Length n random vector of U[0,1] entries
e = ones(n)  # Length n vector of ones
D = diagm(e) # Construct a diagonal matrix
e2 = diag(D) # Extract a matrix diagonal
```

The identity matrix in Julia is simply `I`. This is an abstract matrix with a size that can usually be inferred from context. In the rare cases when you need a *concrete* instantiation of an identity matrix, you can use `Matrix(I, n, n)`.

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

```
y = [1; 2]      # Length-2 vector
y = [1 2]       # 1-by-2 matrix
M = [1 2; 3 4]  # 2-by-2 matrix
M = [I A]       # Horizontal matrix concatenation
M = [I; A]      # Vertical matrix concatenation
```

Julia uses commas to separate elements of a list-like data type or an array. So `[1, 2]` and `[1; 2]` give us the same thing (a length 2 vector), but `[I, A]` gives us a list consisting of a uniform scaling object and a matrix — not quite the same as horizontal matrix concatenation.

Transpose and rearrangemenent Julia 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: Julia is column-major
avec = reshape(A, prod(size(A)));
A = reshape(avec, n, n)

idx = randperm(n) # Random permutation of indices (need to use Random)
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 Julia 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 Julia 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
```

```
5
6   y = A*x   # Matrix times vector
7   y = x'*A  # Vector times matrix
8   C = A*B   # Matrix times matrix
9
10  # Don't use inv!
11  x = A\b   # Solve Ax = b *or* least squares
12  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. Julia does provide these operations. But if you find yourself typing `inv` or `det` in Julia, think long and hard. Is there an alternate formulation? Could you use the forward slash or backslash operations for solving a linear system?

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 Julia statement

```
1  % Compute the sum of x and y (assuming they are exact)
2  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 Julia; 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.

6 Sensitivity, conditioning, and types of error

In almost every sort of numerical computation, we need to think about errors. Errors in numerical computations can come from many different sources, including:

- *Roundoff error* from inexact computer arithmetic.
- *Truncation error* from approximate formulas.
- *Termination of iterations.*
- *Statistical error.*

There are also *model errors* that are related not to how accurately we solve a problem on the computer, but to how accurately the problem we solve models the state of the world.

There are also several different ways we can think about errors. 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 often concerned with *first-order* error analysis, i.e. error analysis based on a linearized approximation to the true problem. First-order analysis is often adequate to understand the effect of roundoff error or truncation of certain approximations. It may not always be enough to understand the effect of large statistical fluctuations.

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?