Notes for 2016-08-26

1 Logistics

1. Our enrollment is at 50, and there are still a few students who want to get in. We only have 50 seats in the room, and I cannot increase the cap further. So if you are not planning to stick with the class, please don't wait to formally drop!

2. I got off to a slow start, but I will typically try to post notes for each lecture before the lecture itself (though maybe only the morning of the lecture). The notes are on GitHub, so if you find errors, please feel free to help me correct them.

3. The notes for the first lecture include some material that I did not talk about explicitly: recommendations for reading in linear algebra and numerical linear algebra to supplement what I cover in class, and a long list of notational conventions that I plan to follow in class and in the notes.

2 Matrix shapes and structures

In linear algebra, we talk about different matrix structures. For example:

- $A \in \mathbb{R}^{n\times n}$ is *nonsingular* if the inverse exists; otherwise it is *singular*.
- $Q \in \mathbb{R}^{n\times n}$ is *orthogonal* if $Q^TQ = I$.
- $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A = A^T$.
- $S \in \mathbb{R}^{n\times n}$ is *skew-symmetric* if $S = -S^T$.
- $L \in \mathbb{R}^{n\times m}$ is *low rank* if $L = UV^T$ for $U \in \mathbb{R}^{n\times k}$ and $V \in \mathbb{R}^{m\times k}$ where $k \ll \min(m, n)$.

These are properties of an underlying linear map or quadratic form; if we write a different matrix associated with an (appropriately restricted) change of basis, it will also have the same properties.

In matrix computations, we also talk about the *shape* (nonzero structure) of a matrix. For example:

- D is *diagonal* if $d_{ij} = 0$ for $i \neq j$.
- T is *tridiagonal* if $t_{ij} = 0$ for $i \notin \{j-1, j, j+1\}$.
- U is *upper triangular* if $u_{ij} = 0$ for $i > j$ and *strictly upper triangular* if $u_{ij} = 0$ for $i \geq j$ (lower triangular and strictly lower triangular are similarly defined).
- H is *upper Hessenberg* if $h_{ij} = 0$ for $i > j + 1$.
- B is *banded* if $b_{ij} = 0$ for $|i - j| > \beta$.
- S is *sparse* if most of the entries are zero. The position of the nonzero entries in the matrix is called the *sparsity structure*.

We often represent the shape of a matrix by marking where the nonzero elements are (usually leaving empty space for the zero elements); for example:

$$\text{Diagonal} \begin{bmatrix} \times & & & & \\ & \times & & & \\ & & \times & & \\ & & & \times & \\ & & & & \times \end{bmatrix} \qquad \text{Tridiagonal} \begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

$$\text{Triangular} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix} \qquad \text{Hessenberg} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

We also sometimes talk about the *graph* of a (square) matrix $A \in \mathbb{R}^{n \times n}$: if we assign a node to each index $\{1, \ldots, n\}$, an edge (i, j) in the graph corresponds to $a_{ij} \neq 0$. There is a close connection between certain classes of graph algorithms and algorithms for factoring sparse matrices or working with different matrix shapes. For example, the matrix A can be permuted so that PAP^T is upper triangular iff the associated directed graph is acyclic.

The shape of a matrix (or graph of a matrix) is not intrinsically associated with a more abstract linear algebra concept; apart from permutations, sometimes, almost any change of basis will completely destroy the shape.

3 Sparse matrices

We say a matrix is *sparse* if the vast majority of the entries are zero. Because we only need to explicitly keep track of the nonzero elements, sparse matrices require less than $O(n^2)$ storage, and we can perform many operations more cheaply with sparse matrices than with dense matrices. In general, the cost to store a sparse matrix, and to multiply a sparse matrix by a vector, is $O(\text{nnz}(A))$, where $\text{nnz}(A)$ is the *number of nonzeros* in A.

Two specific classes of sparse matrices are such ubiquitous building blocks that it is worth pulling them out for special attention. These are diagonal matrices and permutation matrices. Many linear algebra libraries also have support for *banded* matrices (and sometimes for generalizations such as *skyline* matrices). MATLAB also provides explicit support for general sparse matrices in which the nonzeros can appear in any position.

3.1 Diagonal matrices

A diagonal matrix is zero except for the entries on the diagonal. We often associate a diagonal matrix with the vector of these entries, and we will adopt in class the notational convention used in MATLAB: the operator diag maps a vector to the corresponding diagonal matrix, and maps a matrix to the vector of diagonal entries. For example, for the vector and matrix

$$d = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \quad D = \begin{bmatrix} d_1 & & \\ & d_2 & \\ & & d_3 \end{bmatrix}$$

we would write $D = \text{diag}(d)$ and $d = \text{diag}(D)$.

The MATLAB routine `diag` forms a dense representation of a diagonal matrix. Next to `inv`, it is one of the MATLAB commands that is most commonly poorly used. The primary *good* use of `diag` is as the first term in a sum that builds a more complicated matrix. But *multiplication* by a diagonal matrix should never go through the `diag` routine. To multiple a diagonal matrix by a vector, the preferred idiom is to use the elementwise multiplication operation, i.e.

```
y = diag(d) * x; % Bad -- O(n^2) time and intermediate storage
y = d .* x; % Good -- equivalent, but O(n) time and space
```

To multiply a diagonal matrix by a vector in MATLAB, use the `bsxfun` command, e.g.

```
B = diag(d) * A; % Bad -- left scaling in O(n^3) time
C = A * diag(d); % Bad -- right scaling in O(n^3) time
B = bsxfun(@times, d, A); % Good -- O(n^2) time
C = bsxfun(@times, A, d.'); % Ditto
```

3.2 Permutations

A permutation matrix is a 0-1 matrix in which one appears exactly once in each row and column. We typically use P or Π to denote permutation matrices; if there are two permutations in a single expression, we might use P and Q.

A permutation matrix is so named because it permutes the entries of a vector. As with diagonal matrices, it is usually best to work with permutations implicitly in computational practice. For any given permutation vector P, we can define an associated mapping vector p such that $p(i) = j$ iff $P_{ij} = 1$. We can then apply the permutation to a vector or matrix using MATLAB's indexing operations:

```
B = P*A; % Straightforward, but slow if P is a dense rep'n
C = A*P';
B = A(p,:); % Better
C = A(:,p);
```

To apply a transpose permutation, we would usually use the permuted indexing on the destination rather than the source:

```
y = P'*x; % Implies that P*y = x
y(p) = x; % Apply the transposed permutation via indexing
```

3.3 Narrowly banded matrices

If a matrix A has zero entries outside a narrow band near the diagonal, we say that A is a *banded* matrix. More precisely, if $a_{ij} = 0$ for $j < i - k_1$ or $j > i + k_2$, we say that A has *lower bandwidth* k_1 and *upper bandwidth* k_2. The most common narrowly-banded matrices in matrix computations (other than diagonal matrices) are *tridiagonal* matrices in which $k_1 = k_2 = 1$.

In the conventional storage layout for band matrices (used by LAPACK) the nonzero entries for a band matrix A are stored in a packed storage matrix B such that each column of B corresponds to a column of A and each row

of B corresponds to a nonzero (off-)diagonal of A. For example,

$$\begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{bmatrix} \mapsto \begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{bmatrix}$$

MATLAB does not provide easy specialized support for band matrices (though it is possible to access the band matrix routines if you are tricky). Instead, the simplest way to work with narrowly banded matrices in MATLAB is to use a general sparse representation.

3.4 General sparse matrices

For diagonal and band matrices, we are able to store nonzero matrix entries explicitly, but (as with the dense matrix format) the locations of those nonzero entries in the matrix are implicit. For permutation matrices, the values of the nonzero entries are implicit (they are always one), but we must store their positions explicitly. In a *general* sparse matrix format, we store both the positions and the values of nonzero entries explicitly.

For input and output, MATLAB uses a *coordinate* format for sparse matrices consisting of three parallel arrays (`i`, `j`, and `aij`). Each entry in the parallel arrays represents a nonzero in the matrix with value `aij(k)` at row `i(k)` and column `j(k)`. For input, repeated entries with the same row and column are allowed; in this case, all the entries for a given location are summed together in the final matrix. For example,

```
i = [1, 2, 3, 4, 4]; % Row indices
j = [2, 3, 4, 5, 5]; % Col indices
v = [5, 8, 13, 21, 34]; % Entry values/contributions
A = sparse(i,j,v,5,5); % 5-by-5 sparse matrix
full(A) % Convert to dense format and display

% Output:
% ans =
%
%  0   5   0   0   0
%  0   0   8   0   0
%  0   0   0  13   0
%  0   0   0   0  55
%  0   0   0   0   0
```

This functionality is useful in some applications (e.g. for assembling finite element matrices).

Internally, MATLAB uses a *compressed sparse column* format for sparse matrices. In this format, the row position and value for each nonzero are stored in parallel arrays, in column-major order (i.e. all the elements of column k appear before elements of column $k+1$). The column positions are not stored explicitly for every element; instead, a *pointer array* indicates the offset in the row and entry arrays of the start of the data for each column; a pointer array entry at position $n+1$ indicates the total number of nonzeros in the data structure.

The compressed sparse column format has some features that may not be obvious at first:

- For very sparse matrices, multiplying a sparse format matrix by a vector is much faster than multiplying a dense format matrix by a vector — but this is not true if a significant fraction of the matrix is nonzeros. The tradeoff depends on the matrix size and machine details, but sparse matvecs will often have the same speed as — or even be slower than — dense matvecs when the sparsity is above a few percent.

- Adding contributions into a sparse matrix is relatively slow, as each sum requires recomputing the sparse indexing data structure and reallocating memory. To build up a sparse matrix as the sum of many components, it is usually best to use the coordinate form first.

In general, though, the sparse matrix format has a great deal to recommend it for genuinely sparse matrices. MATLAB uses the sparse matrix format not only for general sparse matrices, but also for the special case of banded matrices.

4 Data-sparse matrices

A *sparse* matrix has mostly zero entries; this lets us design compact storage formats with space proportional to the number of nonzeros, and fast matrix-vector multiplication with time proportional to the number of nonzeros. A *data-sparse* matrix can be described with far fewer than n^2 parameters, even if it is not sparse. Such matrices usually also admit compact storage schemes and fast matrix-vector products. This is significant because many of the

iterative algorithms we describe later in the semester do not require any particular representation of the matrix; they only require that we be able to multiply by a vector quickly.

The study of various data sparse representations has blossomed into a major field of study within matrix computations; in this section we give a taste of a few of the most common types of data sparsity. We will see several of these structures in model problems used over the course of the class.

4.1 (Nearly) low-rank matrices

The simplest and most common data-sparse matrices are *low-rank* matrices. If $A \in \mathbb{R}^{m\times n}$ has rank k, it can be written in outer product form as

$$A = UW^T, \quad , U \in \mathbb{R}^{m\times k}, W \in \mathbb{R}^{n\times k}.$$

This factored form has a storage cost of $(n+m)k$, a significant savings over the mn cost of the dense representation in the case $k \ll \max(m,n)$. To multiply a low-rank matrix by a vector fast, we need only to use associativity of matrix operations

```
y = (U*V')*x; % O(mn) storage, O(mnk) flops
y = U*(V'*x); % O((m+n) k) storage and flops
```

4.2 Circulant, Toeplitz, and Hankel structure

A *Toeplitz* matrix is a matrix in which each (off)-diagonal is constant, e.g.

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_{-1} & a_0 & a_1 & a_2 \\ a_{-2} & a_{-1} & a_0 & a_1 \\ a_{-3} & a_{-2} & a_{-1} & a_0 \end{bmatrix}.$$

Toeplitz matrices play a central role in the theory of constant-coefficient finite difference equations and in many applications in signal processing.

Multiplication of a Toeplitz matrix by a vector represents (part of) a *convolution*; and afficionados of Fourier analysis and signal processing may already know that this implies that matrix multiplication can be done in $O(n \log n)$ time using a discrete Fourier transforms. The trick to this is to

view the Toeplitz matrix as a block in a larger *circulant* matrix

$$C = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_{-3} & a_{-2} & a_{-1} \\ a_{-1} & a_0 & a_1 & a_2 & a_3 & a_{-3} & a_{-2} \\ a_{-2} & a_{-1} & a_0 & a_1 & a_2 & a_3 & a_{-3} \\ a_{-3} & a_{-2} & a_{-1} & a_0 & a_1 & a_2 & a_3 \\ a_3 & a_{-3} & a_{-2} & a_{-1} & a_0 & a_1 & a_2 \\ a_2 & a_3 & a_{-3} & a_{-2} & a_{-1} & a_0 & a_1 \\ a_1 & a_2 & a_3 & a_{-3} & a_{-2} & a_{-1} & a_0 \end{bmatrix} = \sum_{k=-3}^{3} a_{-k} P^k,$$

where P is the cyclic permutation matrix

$$P = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 1 & & & & 0 \\ & 1 & & & 0 \\ & & \ddots & & \vdots \\ & & & 1 & 0 \end{bmatrix}.$$

As we will see later in the course, the discrete Fourier transform matrix is the eigenvector matrix for this cyclic permutation, and this is a gateway to fast matrix-vector multiplication algorithms.

Closely-related to Toeplitz matrices are *Hankel* matrices, which are constant on skew-diagonals (that is, they are Toeplitz matrices flipped upside down). Hankel matrices appear in numerous applications in control theory.

4.3 Separability and Kronecker product structure

The *Kronecker product* $A \otimes B \in \mathbb{R}^{(mp)\times(nq)}$ of matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$ is the (gigantic) matrix

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}.$$

Multiplication of a vector by a Kronecker product represents a matrix triple product:

$$(A \otimes B)\,\mathrm{vec}(X) = \mathrm{vec}(BXA^T)$$

where $\mathrm{vec}(X)$ represents the vector formed by listing the elements of a matrix in column major order, e.g.

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

Kronecker product structure appears often in control theory applications and in problems that arise from difference or differential equations posed on regular grids — you should expect to see it for regular discretizations of differential equations where separation of variables works well. There is also a small industry of people working on *tensor decompositions*, which feature sums of Kronecker products.

4.4 Low-rank block structure

In problems that come from certain areas of mathematical physics, integral equations, and PDE theory, one encounters matrices that are not low rank, but have *low-rank submatrices*. The *fast multipole method* computes a matrix-vector product for one such class of matrices; and again, there is a cottage industry of related methods, including the $\mathcal{H}$ matrices studied by Hackbush and colleagues, the sequentially semi-separable (SSS) and heirarchically semi-separable (HSS) matrices, quasi-separable matrices, and a horde of others. A good reference is the pair of books by Vandebril, Van Barel and Mastronardi [1, 2].

References

[1] Raf Vandebril, Marc Van Barel, and Nicola Mastonardi. *Matrix Computations and Semiseparable Matrices: Eigenvalue and Singular Value Methods*. John Hopkins University Press, 2010.

[2] Raf Vandebril, Marc Van Barel, and Nicola Mastonardi. *Matrix Computations and Semiseparable Matrices: Linear Systems*. John Hopkins University Press, 2010.