2022-08-30

1 Matrix algebra versus linear algebra

> We share a philosophy about linear algebra: we think basis-free, we write basis-free, but when the chips are down we close the office door and compute with matrices like fury.
>
> — Irving Kaplansky on the late Paul Halmos

Linear algebra is fundamentally about the structure of vector spaces and linear maps between them. A matrix represents a linear map with respect to some bases. Properties of the underlying linear map may be more or less obvious via the matrix representation associated with a particular basis, and much of matrix computations is about finding the right basis (or bases) to make the properties of some linear map obvious. We also care about finding changes of basis that are "nice" for numerical work.

In some cases, we care not only about the linear map a matrix represents, but about the matrix itself. For example, the *graph* associated with a matrix $A \in \mathbb{R}^{n \times n}$ has vertices $\{1, \ldots, n\}$ and an edge (i, j) if $a_{ij} \neq 0$. Many of the matrices we encounter in this class are special because of the structure of the associated graph, which we usually interpret as the "shape" of a matrix (diagonal, tridiagonal, upper triangular, etc). This structure is a property of the matrix, and not the underlying linear transformation; change the bases in an arbitrary way, and the graph changes completely. But identifying and using special graph structures or matrix shapes is key to building efficient numerical methods for all the major problems in numerical linear algebra.

In writing, we represent a matrix concretely as an array of numbers. Inside the computer, a *dense* matrix representation is a two-dimensional array data structure, usually ordered row-by-row or column-by-column in order to accomodate the one-dimensional structure of computer memory address spaces. While much of our work in the class will involve dense matrix layouts, it is important to realize that there are other data structures! The "best" representation for a matrix depends on the structure of the matrix and on what we want to do with it. For example, many of the algorithms we will discuss later in the course only require a black box function to multiply an (abstract) matrix by a vector.

2 Dense matrix basics

There is one common data structure for dense vectors: we store the vector as a sequential array of memory cells. In contrast, there are *two* common data structures for general dense matrices. In MATLAB (and Fortran), matrices are stored in *column-major* form. For example, an array of the first four positive integers interpreted as a two-by-two column major matrix represents the matrix

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

The same array, when interpreted as a *row-major* matrix, represents

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

Unless otherwise stated, we will assume all dense matrices are represented in column-major form for this class. As we will see, this has some concrete effects on the efficiency of different types of algorithms.

2.1 The BLAS

The *Basic Linear Algebra Subroutines* (BLAS) are a standard library interface for manipulating dense vectors and matrices. There are three *levels* of BLAS routines:

- **Level 1**: These routines act on vectors, and include operations such scaling and dot products. For vectors of length n, they take $O(n^1)$ time.
- **Level 2:** These routines act on a matrix and a vector, and include operations such as matrix-vector multiplication and solution of triangular systems of equations by back-substitution. For $n \times n$ matrices and length n vectors, they take $O(n^2)$ time.
- **Level 3:** These routines act on pairs of matrices, and include operations such as matrix-matrix multiplication. For $n \times n$ matrices, they take $O(n^3)$ time.

All of the BLAS routines are superficially equivalent to algorithms that can be written with a few lines of code involving one, two, or three nested loops

(depending on the level of the routine). Indeed, except for some refinements involving error checking and scaling for numerical stability, the reference BLAS implementations involve nothing more than these basic loop nests. But this simplicity is deceptive — a surprising amount of work goes into producing high performance implementations.

2.2 Locality and memory

When we analyze algorithms, we often reason about their complexity abstractly, in terms of the scaling of the number of operations required as a function of problem size. In numerical algorithms, we typically measure *flops* (short for floating point operations). For example, consider the loop to compute the dot product of two vectors:

```
function mydot(x, y)
    n = length(x)
    result = 0.0
    for i = 1:n
        result += x[i]*y[i] # Two flops/iteration
    end
    result
end
```

Because it takes n additions and n multiplications, we say this code takes $2n$ flops, or (a little more crudely) $O(n)$ flops.

On modern machines, though, counting flops is at best a crude way to reason about how run times scale with problem size. This is because in many computations, the time to do arithmetic is dominated by the time to fetch the data into the processor! A detailed discussion of modern memory architectures is beyond the scope of these notes, but there are at least two basic facts that everyone working with matrix computations should know:

- Memories are optimized for access patterns with *spatial locality*: it is faster to access entries of memory that are close to each other (ideally in sequential order) than to access memory entries that are far apart. Beyond the memory system, sequential access patterns are good for *vectorization*, i.e. for scheduling work to be done in parallel on the vector arithmetic units that are present on essentially all modern processors.

- Memories are optimized for access patterns with *temporal locality*; that is, it is much faster to access a small amount of data repeatedly than to access large amounts of data.

The main mechanism for optimizing access patterns with temporal locality is a system of *caches*, fast and (relatively) small memories that can be accessed more quickly (i.e. with lower latency) than the main memory. To effectively use the cache, it is helpful if the *working set* (memory that is repeatedly accessed) is smaller than the cache size. For level 1 and 2 BLAS routines, the amount of work is proportional to the amount of memory used, and so it is difficult to take advantage of the cache. On the other hand, level 3 BLAS routines do $O(n^3)$ work with $O(n^2)$ data, and so it is possible for a clever level 3 BLAS implementation to effectively use the cache.

2.3 Matrix-vector multiply

Let us start with a very simple Julia function for matrix-vector multiplication:

```
function matvec1_row(A, x)
  m, n = size(A)
  y = zeros(m)
  for i = 1:m
    for j = 1:n
      y[i] += A[i,j]*x[j]
    end
  end
  y
end
```

We could just as well have switched the order of the i and j loops to give us a column-oriented rather than row-oriented version of the algorithm:

```
function matvec1_col(A, x)
  m, n = size(A)
  y = zeros(m)
  for j = 1:n
    for i = 1:m
      y[i] += A[i,j]*x[j]
    end
  end
  y
end
```

It's not too surprising that the builtin matrix-vector multiply routine in Julia runs faster than either of our `matvec` variants, but there are some other surprises lurking. The Pluto notebook accompanying this lecture goes into more detail.

2.4 Matrix-matrix multiply

The classic algorithm to compute $C := C + AB$ involves three nested loops

```
function matmul!(A, B, C)
  m, n = size(A)
  n, p = size(B)
  for i = 1:m
    for j = 1:n
      for k = 1:p
        C[i,j] += A[i,k]*B[k,j]
      end
    end
  end
end
```

This is sometimes called an *inner product* variant of the algorithm, because the innermost loop is computing a dot product between a row of A and a column of B. But addition is commutative and associative, so we can sum the terms in a matrix-matrix product in any order and get the same result. And we can interpret the orders! A non-exhaustive list is:

- `ij(k)` or `ji(k)`: Compute entry c_{ij} as a product of row i from A and column j from B (the *inner product* formulation)
- `k(ij)`: C is a sum of outer products of column k of A and row k of B for k from 1 to n (the *outer product* formulation)
- `i(jk)` or `i(kj)`: Each row of C is a row of A multiplied by B
- `j(ik)` or `j(ki)`: Each column of C is A multiplied by a column of B

At this point, we could write down all possible loop orderings and run a timing experiment, similar to what we did with matrix-vector multiplication. But the truth is that high-performance matrix-matrix multiplication routines use another access pattern altogether, involving more than three nested loops, and we will describe this now.

2.5 Blocking and performance

The basic matrix multiply outlined in the previous section will usually be at least an order of magnitude slower than a well-tuned matrix multiplication routine. There are several reasons for this lack of performance, but one of

the most important is that the basic algorithm makes poor use of the *cache*. Modern chips can perform floating point arithmetic operations much more quickly than they can fetch data from memory; and the way that the basic algorithm is organized, we spend most of our time reading from memory rather than actually doing useful computations. Caches are organized to take advantage of *spatial locality*, or use of adjacent memory locations in a short period of program execution; and *temporal locality*, or re-use of the same memory location in a short period of program execution. The basic matrix multiply organizations don't do well with either of these. A better organization would let us move some data into the cache and then do a lot of arithmetic with that data. The key idea behind this better organization is *blocking*.

When we looked at the inner product and outer product organizations in the previous sections, we really were thinking about partitioning A and B into rows and columns, respectively. For the inner product algorithm, we wrote A in terms of rows and B in terms of columns

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

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

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

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

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

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

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

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

2.6 The lazy man's approach to performance

An algorithm like matrix multiplication seems simple, but there is a lot under the hood of a tuned implementation, much of which has to do with the organization of memory. We often get the best "bang for our buck" by taking the time to formulate our algorithms in block terms, so that we can spend most of our computation inside someone else's well-tuned matrix multiply routine (or something similar). There are several implementations of the Basic Linear Algebra Subroutines (BLAS), including some implementations provided by hardware vendors and some automatically generated by tools like ATLAS. The best BLAS library varies from platform to platform, but by using a good BLAS library and writing routines that spend a lot of time in *level 3* BLAS operations (operations that perform $O(n^3)$ computation on $O(n^2)$ data and can thus potentially get good cache re-use), we can hope to build linear algebra codes that get good performance across many platforms.

This is also a good reason to use systems like Julia, MATLAB, or NumPy (built appropriately): they uses pretty good BLAS libraries, and so you can often get surprisingly good performance from it for the types of linear algebraic computations we will pursue.

3 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 $Q^T Q = I$.
- $A \in \mathbb{R}^{n \times n}$ 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:

Diagonal
$$\begin{bmatrix} \times & & & & \\ & \times & & & \\ & & \times & & \\ & & & \times & \\ & & & & \times \end{bmatrix}$$

Tridiagonal
$$\begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

Triangular
$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}$$

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.

4 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.

4.1 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 Julia `Diagonal` type represents diagonal matrices.

4.2 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
```

4.3 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,

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

Julia 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 Julia is to use a general sparse representation.

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

Internally, Julia's sparse package 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.

5 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.

5.1 (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
```

5.2 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.

5.3 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.

5.4 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 [2, 1].

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.