CHAPTER 1

Matrix factorizations and low rank approximation

We will start with a review of basic concepts from linear algebra that will be used frequently. Note that the pace is fast here, and assumes that the reader has some level of familiarity with the material. The one concept that is perhaps less well known is the material on the *interpolative decomposition (ID)* in Section 1.5.

1.1. Notation, etc

1.1.1. Norms. Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ denote a vector in $\mathbb{R}^n$ or $\mathbb{C}^n$. Our default norm for vectors is the Euclidean norm

$$\|\mathbf{x}\| = \left(\sum_{j=1}^{n} |x_j|^2 \right)^{1/2}.$$

We will at times also use ℓ^p norms

$$\|\mathbf{x}\|_p = \left(\sum_{j=1}^{n} |x_j|^p \right)^{1/p}.$$

Let $\mathbf{A}$ denote an $m \times n$ matrix. For the most part, we allow $\mathbf{A}$ to have complex entries. We define the *spectral norm* of $\mathbf{A}$ via

$$\|\mathbf{A}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \sup_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

We define the *Frobenius norm* of $\mathbf{A}$ via

$$\|\mathbf{A}\|_{\mathrm{F}} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |\mathbf{A}(i,j)|^2 \right)^{1/2}.$$

Observe that the spectral norm and the Frobenius norms satisfy the inequalities

$$\|\mathbf{A}\| \leq \|\mathbf{A}\|_{\mathrm{F}} \leq \sqrt{\min(m,n)}\, \|\mathbf{A}\|.$$

1.1.2. Transpose and adjoint. Given an $m \times n$ matrix $\mathbf{A}$, the *transpose* $\mathbf{A}^{\mathrm{t}}$ is the $n \times m$ matrix $\mathbf{B}$ with entries

$$\mathbf{B}(i,j) = \mathbf{A}(j,i).$$

The transpose is most commonly used for *real* matrices. It can also be used for a *complex* matrix, but more typically, we then use the *adjoint*, which is the complex conjugate of the transpose

$$\mathbf{A}^* = \overline{\mathbf{A}^{\mathrm{t}}}.$$

1.1.3. Subspaces. Let $\mathbf{A}$ be an $m \times n$ matrix.

- The *row space* of $\mathbf{A}$ is denoted $\text{row}(\mathbf{A})$ and is defined as the subspace of $\mathbb{R}^n$ spanned by the rows of $\mathbf{A}$.
- The *column space* of $\mathbf{A}$ is denoted $\text{col}(\mathbf{A})$ and is defined as the subspace of $\mathbb{R}^m$ spanned by the columns of $\mathbf{A}$. The column space equals the *range* of $\mathbf{A}$, so $\text{col}(\mathbf{A}) = \text{ran}(\mathbf{A}) = \{\mathbf{A}\mathbf{x} : \mathbf{x} \in \mathbb{R}^n\}$.
- The *nullspace* or *kernel* of $\mathbf{A}$ is the subspace $\ker(\mathbf{A}) = \text{null}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{0}\}$.

1.1.4. Special classes of matrices. We use the following terminology to classify matrices:

- An $m \times n$ matrix $\mathbf{A}$ is *orthonormal* if its columns form an orthonormal basis, i.e. $\mathbf{A}^*\mathbf{A} = \mathbf{I}$.
- An $n \times n$ matrix $\mathbf{A}$ is *normal* if $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A}$.
- An $n \times n$ real matrix $\mathbf{A}$ is *symmetric* if $\mathbf{A}^{\text{t}} = \mathbf{A}$.
- An $n \times n$ matrix $\mathbf{A}$ is *self-adjoint* if $\mathbf{A}^* = \mathbf{A}$.
- An $n \times n$ matrix $\mathbf{A}$ is *skew-adjoint* if $\mathbf{A}^* = -\mathbf{A}$.
- An $n \times n$ matrix $\mathbf{A}$ is *unitary* if it is invertible and $\mathbf{A}^* = \mathbf{A}^{-1}$.

Suppose that $\mathbf{A}$ is an $n \times n$ self-adjoint matrix. Then for any $x \in \mathbb{C}^n$, one can easily verify that $\mathbf{x}^*\mathbf{A}\mathbf{x}$ is a *real* scalar number. We then say that $\mathbf{A}$ is *non-negative* if $\mathbf{x}^*\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{C}^n$, and *positive* if $\mathbf{x}^*\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbb{C}^n$. The properties *non-positive* and *negative* are defined analogously.

1.2. Low rank approximation

1.2.1. Exact rank deficiency. Let $\mathbf{A}$ be an $m \times n$ matrix. Let k denote an integer between 1 and $\min(m, n)$. Then the following conditions are equivalent:

- The columns of $\mathbf{A}$ span a subspace of $\mathbb{R}^m$ of dimension k.
- The rows of $\mathbf{A}$ span a subspace of $\mathbb{R}^n$ of dimension k.
- The nullspace of $\mathbf{A}$ has dimension $n - k$.
- The nullspace of $\mathbf{A}^*$ has dimension $m - k$.

If $\mathbf{A}$ satisfies any of these criteria, then we say that $\mathbf{A}$ *has rank* k. When $\mathbf{A}$ has rank k, it is possible to find matrices $\mathbf{E}$ and $\mathbf{F}$ such that

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times k}{\mathbf{E}} \underset{k \times n}{\mathbf{F}}.$$

The columns of $\mathbf{E}$ span the column space of $\mathbf{A}$, and the rows of $\mathbf{F}$ span the row space of $\mathbf{A}$. Having access to such factors $\mathbf{E}$ and $\mathbf{F}$ can be very helpful:

- Storing $\mathbf{A}$ requires mn words of storage.
  Storing $\mathbf{E}$ and $\mathbf{F}$ requires $km + kn$ words of storage.
- Given a vector $\mathbf{x}$, computing $\mathbf{A}\mathbf{x}$ requires mn flops.
  Given a vector $\mathbf{x}$, computing $\mathbf{A}\mathbf{x} = \mathbf{E}(\mathbf{F}\mathbf{x})$ requires $km + kn$ flops.
- The factors $\mathbf{E}$ and $\mathbf{F}$ are often useful for *data interpretation.*

In practice, we often impose conditions on the factors. For instance, in the well known QR decomposition, the columns of $\mathbf{E}$ are orthonormal, and $\mathbf{F}$ is upper triangular (up to permutations of the columns).

1.2.2. Approximate rank deficiency. The condition that $\mathbf{A}$ has *precisely* rank k is of high theoretical interest, but is not realistic in practical computations. Frequently, the numbers we use have been measured by some device with finite precision, or they may have been computed via a simulation with some approximation errors (e.g. by solving a PDE numerically). In any case, we almost always work with data that is stored in some finite precision format (typically about 10^{-15}). For all these reasons, it is very useful to define the concept of *approximate rank*. In this course, we will typically use the following definition:

DEFINITION 1. Let $\mathbf{A}$ be an $m \times n$ matrix, and let ε be a positive real number. We then define the *ε-rank of $\mathbf{A}$* as the unique integer k such that both the following two conditions hold:

(a) There exists a matrix $\mathbf{B}$ of precise rank k such that $\|\mathbf{A} - \mathbf{B}\| \leq \varepsilon$.
(b) There does not exist any matrix $\mathbf{B}$ of rank less than k such that $\|\mathbf{A} - \mathbf{B}\| \leq \varepsilon$.

The term ε-rank is sometimes used without enforcing condition (b): We sometimes say that $\mathbf{A}$ has ε-rank k if

$$\inf\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } k\} \leq \varepsilon,$$

without worrying about whether the rank could actually be smaller. In other words, we sometimes say "$\mathbf{A}$ has ε-rank k" when we really mean "$\mathbf{A}$ has ε-rank at most k."

1.3. The Singular Value Decomposition

1.3.1. Definition of full SVD. Let $\mathbf{A}$ be an $m \times n$ matrix (*any* matrix, it can be rectangular, complex or real valued, etc). Set $p = \min(m, n)$. Then $\mathbf{A}$ admits a factorization

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times p}{\mathbf{U}} \quad \underset{p \times p}{\mathbf{D}} \quad \underset{p \times n}{\mathbf{V}^*}, \tag{1.1}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and where $\mathbf{D}$ is diagonal. We write these out as

$$\mathbf{U} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_p \end{bmatrix},$$
$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_p \end{bmatrix},$$
$$\mathbf{D} = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \sigma_p \end{bmatrix}.$$

The vectors $\{\mathbf{u}_j\}_{j=1}^p$ are the *left singular vectors* and the vectors $\{\mathbf{v}_j\}_{j=1}^p$ are the *right singular vectors*. These form orthonormal bases of the ranges of $\mathbf{A}$ and $\mathbf{A}^*$, respectively. The values $\{\sigma_j\}_{j=1}^p$ are the *singular values* of $\mathbf{A}$. These are customarily ordered so that

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_p \geq 0.$$

The SVD (1.1) can alternatively be written as a decomposition of $\mathbf{A}$ as a sum of p "outer products" of vectors

$$\mathbf{A} = \sum_{j=1}^{p} \sigma_j \, \mathbf{u}_j \, \mathbf{v}_j^*.$$

1.3.2. Low rank approximation via SVD. For purposes of approximating a given matrix by a matrix of low rank, the SVD is in a certain sense *optimal*. To be precise, suppose that we are given a matrix $\mathbf{A}$, and have computed its SVD (1.1). Then for an integer $k \in \{1, 2, \ldots, p\}$, we define

$$\mathbf{A}_k = \sum_{j=1}^{k} \sigma_j \, \mathbf{u}_j \, \mathbf{v}_j^*.$$

Clearly $\mathbf{A}_k$ is a matrix of rank k. It is in fact the particular rank-k matrix that best approximates $\mathbf{A}$. The classical Eckart-Young theorem [**11, 36**] states that

$$\|\mathbf{A} - \mathbf{A}_k\| = \inf\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } k\},$$
$$\|\mathbf{A} - \mathbf{A}_k\|_{\mathrm{F}} = \inf\{\|\mathbf{A} - \mathbf{B}\|_{\mathrm{F}} : \mathbf{B} \text{ has rank } k\}.$$

If $k < p$, then it is easily verified that the minimal residuals evaluate to

$$\|\mathbf{A} - \mathbf{A}_k\| = \sigma_{k+1},$$
$$\|\mathbf{A} - \mathbf{A}_k\|_{\mathrm{F}} = \left(\sum_{j=k+1}^{p} \sigma_j^2 \right)^{1/2}.$$

REMARK 1.1. *For normal matrices, a truncated eigenvalue decomposition attains exactly the same approximation error as a truncated SVD, so for this particular class of matrices, either decomposition can be used. (But note that the EVD could be complex even for a real matrix.)*

1.3.3. Connection between the SVD and the eigenvalue decomposition. Let $\mathbf{A}$ be a general matrix of size $m \times n$. Then form the $m \times m$ matrix

$$\mathbf{B} = \mathbf{A}\mathbf{A}^*.$$

Observe that $\mathbf{B}$ is self-adjoint and non-negative, so there exist m orthonormal eigenvector $\{\mathbf{u}_j\}_{j=1}^m$ with associated *real* eigenvalues $\{\lambda_j\}_{j=1}^m$. Then $\{\mathbf{u}_j\}_{j=1}^m$ are left singular vectors of $\mathbf{A}$ associated with singular values $\sigma_j = \sqrt{\lambda_j}$. Analogously, if we form the $n \times n$ matrix

$$\mathbf{C} = \mathbf{A}^*\mathbf{A},$$

then the eigenvectors of $\mathbf{C}$ are right singular vectors of $\mathbf{A}$.

1.3.4. Computing the SVD. One can prove that in general computing the singular values (or eigenvalues) of an $n \times n$ matrix *exactly* is an equivalent problem to solving an n'th order polynomial (the characteristic equation). Since this problem is known to not, in general, have an algebraic solution for $n \geq 5$, it is not surprising that algorithms for computing singular values are iterative in nature. However, they typically converge very fast, so for practical purposes, algorithms for computing a full SVD perform quite similarly to algorithms for computing full QR decompositions. For details, we refer to standard textbooks [**14, 38**], but some facts about these algorithms are relevant to what follows:

- Standard algorithms for computing the SVD of a dense $m \times n$ matrix (as found in Matlab, LAPACK, etc), have practical asymptotic speed of $O(mn \min(m, n))$.
- While the asymptotic complexity of algorithms for computing full factorizations tend to be $O(mn \min(m, n))$ regardless of the choice of factorization (LU, QR, SVD, etc), the scaling constants are different. In particular, column pivoted QR is slower than unpivoted QR, and the SVD is slower still.
- Standard library functions for computing the SVD almost always produce results that are accurate to full double precision accuracy. They can fail to converge for certain matrices, but in high-quality software, this happens very rarely.
- Standard algorithms are challenging to parallelize well. For a small number of cores on a modern CPU they work well, but performance deteriorates as the number of cores increase, or if the computation is to be carried out on a GPU or a distributed memory machine.

(1) $\mathbf{Q}_0 = [\,]$; $\mathbf{R}_0 = [\,]$; $\mathbf{A}_0 = \mathbf{A}$; $p = \min(m, n)$;
(2) **for** $j = 1 : p$
(3) $\quad i_j = \mathsf{argmin}\{\|\mathbf{A}(:, \ell)\| : \ \ell = 1, 2, \ldots, n\}$
(4) $\quad \mathbf{q} = \mathbf{A}(:, i)/\|\mathbf{A}(:, i)\|$.
(5) $\quad \mathbf{r} = \mathbf{q}^*\mathbf{A}_{j-1}$
(6) $\quad \mathbf{Q}_j = [\mathbf{Q}_{j-1}\ \mathbf{q}]$
(7) $\quad \mathbf{R}_j = \begin{bmatrix} \mathbf{R}_{j-1} \\ \mathbf{r} \end{bmatrix}$
(8) $\quad \mathbf{A}_j = \mathbf{A}_{j-1} - \mathbf{q}\mathbf{r}$
(9) **end for**
(10) $\mathbf{Q} = \mathbf{Q}_p$; $\mathbf{R} = \mathbf{R}_p$;

FIGURE 1.1. The basic Gram Schmidt process. Given an input matrix $\mathbf{A}$, the algorithm computes an ON matrix $\mathbf{Q}$ and a "morally" upper triangular matrix $\mathbf{R}$ such that $\mathbf{A} = \mathbf{Q}\mathbf{R}$. At the intermediate steps, we have $\mathbf{A} = \mathbf{A}_j + \mathbf{Q}_j\mathbf{R}_j$. Moreover at any step k, the columns of $\mathbf{Q}(:, 1 \ : \ k)$ form an ON basis for the space spanned by the pivot vectors $\mathbf{A}(:, [i_1, i_2, \ldots, i_k])$.

1.4. The QR decomposition

1.4.1. Definition of the column pivoted QR (CPQR) decomposition. Let $\mathbf{A}$ be an $m \times n$ matrix. Set $p = \min(m, n)$. Then $\mathbf{A}$ admits a factorization

$$\underset{m\times n}{\mathbf{A}} \quad \underset{n\times n}{\mathbf{P}} \quad = \quad \underset{m\times p}{\mathbf{Q}} \quad \underset{p\times n}{\mathbf{R}}, \tag{1.2}$$

where $\mathbf{P}$ is a permutation matrix, $\mathbf{Q}$ is orthonormal, and $\mathbf{R}$ is upper triangular. The action of the matrix $\mathbf{P}$ is to reorder the columns of $\mathbf{A}$. To be precise, if we let $\{\mathbf{a}_j\}_{j=1}^n$ denote the columns of $\mathbf{A}$, so that $\mathbf{A} = \begin{bmatrix}\mathbf{a}_1\ \mathbf{a}_2\ \cdots\ \mathbf{a}_n\end{bmatrix}$, then there is an index vector $I = [i_1,\ i_2,\ \ldots,\ i_n]$ such that

$$\mathbf{A}\mathbf{P} = \mathbf{A}(:, I) = \begin{bmatrix}\mathbf{a}_{i_1}\ \mathbf{a}_{i_2}\ \cdots\ \mathbf{a}_{i_n}\end{bmatrix}.$$

1.4.2. Computing the CPQR. The Gram-Schmidt process with column pivoting is a standard technique for computing the CPQR, see [**14**, Sec. 5.4] or [**38**, Lecture 10]. Let us recapitulate the main steps: Start by finding the largest column of $\mathbf{A}$, and i_1 denote the index of this column. Swap the columns 1 and i_1 to put the pivot column first. Normalize $\mathbf{a}_{i_1}$ to form the first column $\mathbf{q}_1$ of $\mathbf{Q}$, and project the remaining columns of $\mathbf{A}$ onto the orthogonal complement of $\mathbf{q}_1$. Then find the largest of the remaining columns, move it to the second slot, etc. Figure 1.1 summarizes the procedure. This strategy is in standard software packages typically implemented using so called Householder reflectors, which provide superior numerical stability.

Standard algorithms for computing the CPQR are fairly efficient, but suffer from the fact that they necessarily proceed via a sequence of matrix-vector operations, as we will discuss in further detail in Section 1.4.5.

1.4.3. Low rank approximation via the QR factorization. The algorithm for computing the QR factorization can trivially be modified to compute a low rank approximation to a matrix. Using the notation defined in Figure 1.1, we find that after k steps of the algorithm, the matrices $\mathbf{A}_k$, $\mathbf{Q}_k$, and $\mathbf{R}_k$ are related via

$$\underset{m\times n}{\mathbf{A}} \quad = \quad \underset{m\times p}{\mathbf{Q}_j} \quad \underset{p\times n}{\mathbf{R}_j} \quad + \quad \underset{m\times n}{\mathbf{A}_j}.$$

The first term has rank k and the second term is the "remainder."

Now suppose that we are interested in computing a low rank factorization of $\mathbf{A}$ that is accurate to some precision ε, using the Frobenius norm. Then after the k'th step, we can simply evaluate $\|\mathbf{A}_k\|_{\mathrm{F}}$ and stop when this quantity drops below ε.

1.4.4. Computing a partial SVD via a partial QR decomposition. The technique described in Section 1.4.3 results in a factorization of the form

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times k}{\mathbf{Q}} \quad \underset{k\times n}{\mathbf{R}\mathbf{P}^*} + \underset{m\times n}{\mathbf{E}} \tag{1.3}$$

where $\mathbf{Q}$ is orthonormal, $\mathbf{R}$ is upper triangular, and the "error" or "remainder" matrix $\mathbf{E}$ satisfies

$$\|\mathbf{E}\|_{\mathrm{F}} \leq \varepsilon.$$

(We dropped the subscripts k here.) Suppose now that we seek a partial SVD. It turns out that this can be accomplished through two simple steps:

(1) Compute a full SVD of the matrix $\mathbf{R}\mathbf{P}^*$, which is cheap since $\mathbf{R}$ is small (it has only k rows)

$$\mathbf{R}\mathbf{P}^* = \hat{\mathbf{U}}\,\mathbf{D}\,\mathbf{V}^*.$$

(2) Multiply $\mathbf{Q}$ and $\hat{\mathbf{U}}$ together

$$\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}.$$

Observe that now $\mathbf{U}$ and $\mathbf{V}$ are both orthonormal, $\mathbf{D}$ is diagonal, and

$$\mathbf{A} = \mathbf{Q}\,\underbrace{\mathbf{R}\mathbf{P}^*}_{=\hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*} + \mathbf{E} = \underbrace{\mathbf{Q}\hat{\mathbf{U}}}_{=\mathbf{U}}\mathbf{D}\mathbf{V}^* + \mathbf{E} = \mathbf{U}\mathbf{D}\mathbf{V}^* + \mathbf{E}. \tag{1.4}$$

We have obtained a partial SVD. Observe in particular that the error term $\mathbf{E}$ is exactly the same in both (1.3) and (1.4).

1.4.5. Blocking of algorithms and execution speed. The various QR factorization algorithms described in this section are both powerful and useful. They have been developed over decades, and current implementations are highly accurate, entirely robust, and fairly fast. They suffer from one serious short coming, however, which is that they inherently are formed as a sequence of n low-rank updates to a matrix. The reason this is bad is that on modern computers, the cost of moving data (from RAM to cache, between levels of cache, etc) often exceeds the time to execute flops. As an illustration of this phenomenon, suppose that we are given an $m \times n$ matrix $\mathbf{A}$ and a set of vectors $\{\mathbf{x}_i\}_{i=1}^{p}$, and that we seek to evaluate the vectors

$$\mathbf{y}_i = \mathbf{A}\,\mathbf{x}_i, \qquad i = 1, 2, \ldots, p.$$

One could either do this via a simple loop:

for $i = 1 : p$
  $\mathbf{y}_i = \mathbf{A}\mathbf{x}_i$
end for

Or, one could put all the vectors in a matrix and simply evaluate a matrix-matrix product:

$$\big[\mathbf{y}_1\ \mathbf{y}_2\ \cdots\ \mathbf{y}_p\big] = \mathbf{A}\big[\mathbf{x}_1\ \mathbf{x}_2\ \cdots\ \mathbf{x}_p\big].$$

The two options are mathematically equivalent, and they both require precisely mnp flops. But, executing the computation as a matrix-matrix multiplication is *much* faster. To simplify slightly, the reason is that when you execute the loop, the matrix $\mathbf{A}$ has to be read from memory p times. (Real life is more complicated since the compiler might be smart and optimize the loop, etc.) In general, any linear algebraic operation that can be coded using matrix-matrix operations tends to be much faster than a corresponding operation

coded as a sequence of matrix-vector operators. Technically, we sometimes refer to "BLAS3" operations (matrix-matrix) versus "BLAS2" operations (martix-vector) [**8, 5**].

The problem with the column pivoted QR factorization is that it inherently consists of a sequence of BLAS2 operations. This makes it hard to get good performance on multicore CPUs and GPUs (and in fact, even on singlecore CPUs, due to the multiple levels of cache on modern processors). This leaves us in an uncomfortable spot when it comes to low rank approximation. To explain, suppose that $\mathbf{A}$ is an $n \times n$ matrix, and let us consider three different matrix factorization algorithms:

(1) QR factorization without pivoting ("QR").
(2) QR factorization with column pivoting ("CPQR").
(3) Singular value decomposition ("SVD").

All three algorithms have asymptotic complexity of $O(n^3)$, meaning that there are constants such that

$$T_{\text{QR}} \sim C_{\text{QR}}\, n^3, \qquad T_{\text{CPQR}} \sim C_{\text{CPQR}}\, n^3, \qquad T_{\text{SVD}} \sim C_{\text{SVD}}\, n^3.$$

On most computer architectures we have $C_{\text{QR}} < C_{\text{CPQR}} < C_{\text{SVD}}$, and the differences typically are not small. (See Exercise ??). Comparing these three algorithms, we find that:

Algorithm:	QR	CPQR	SVD
Speed:	Fast.	Slow.	Slowest.
Ease of parallelization:	Fairly easy.	Very hard.	Very hard.
Useful for low-rank approximation:	No.	Yes.	Excellent.
Partial factorization possible?	Yes, but not useful.	Yes.	Not easily.

What we would want is an algorithm for low rank approximation that can be *blocked* so that it can be implemented using BLAS3 operations rather than BLAS2 operations. It turns out that randomized sampling provides an excellent path for this, as we will see in Section 2.

1.5. The Interpolatory Decomposition (ID)

1.5.1. Structure preserving factorizations. Any matrix $\mathbf{A}$ of size $m \times n$ and rank k, where $k < \min(m, n)$, admits a so called "interpolative decomposition (ID)" which takes the form

$$\underset{m \times n}{\mathbf{A}} \quad = \quad \underset{m \times k}{\mathbf{C}} \quad \underset{k \times n}{\mathbf{Z}}, \tag{1.5}$$

where the matrix $\mathbf{C}$ is given by a subset of the columns of $\mathbf{A}$ and where $\mathbf{Z}$ is well-conditioned in a sense that we will make precise shortly. The ID has several advantages, as compared to, e.g., the QR or SVD factorizations:

- If $\mathbf{A}$ is sparse or non-negative, then $\mathbf{C}$ shares these properties.
- The ID requires less memory to store than either the QR or the singular value decomposition.
- Finding the indices associated with the spanning columns is often helpful in *data interpretation.*
- In the context of numerical algorithms for discretizing PDEs and integral equations, the ID often preserves "the physics" of a problem in a way that the QR or SVD do not.

One shortcoming of the ID is that when $\mathbf{A}$ is not of precisely rank k, then the approximation error by the best possible rank-k ID can be substantially larger than the theoretically minimal error. (In fact, the ID and the column pivoted QR factorizations are closely related, and they attain *exactly* the same minimal error.)

For future reference, let J_{s} be an index vector in $\{1, 2, \ldots, n\}$ that identifies the k columns in $\mathbf{C}$ so that

$$\mathbf{C} = \mathbf{A}(:, J_{\text{s}}).$$

One can easily show (see, e.g., [**25**, Thm. 9]) that any matrix of rank k admits a factorization (1.5) that is well-conditioned in the sense that each entry of $\mathbf{Z}$ is bounded in modulus by one. However, any algorithm that is guaranteed to find such an optimally conditioned factorization must have combinatorial complexity. Polynomial time algorithms with high practical efficiency are discussed in [**17, 7**]. Randomized algorithms are described in [**21, 39**].

1.5.2. Three flavors of ID: The row, column, and double-sided ID. Section 1.5.1 describes a factorization where we use a subset of the *columns* of $\mathbf{A}$ to span its *column space*. Naturally, this factorization has a sibling which uses the *rows* of $\mathbf{A}$ to span its *row space.* In other words $\mathbf{A}$ also admits the factorization

$$\underset{m\times n}{\mathbf{A}} = \underset{m\times k}{\mathbf{X}}\ \underset{k\times n}{\mathbf{R}}, \tag{1.6}$$

where $\mathbf{R}$ is a matrix consisting of k rows of $\mathbf{A}$, and where $\mathbf{X}$ is a matrix that contains the $k \times k$ identity matrix. We let I_{s} denote the index vector of length k that marks the "skeleton" rows so that $\mathbf{R} = \mathbf{A}(I_{\mathrm{s}}, :)$.

Finally, there exists a so called *double-sided ID* which takes the form

$$\underset{m\times n}{\mathbf{A}} = \underset{m\times k}{\mathbf{X}}\ \underset{k\times k}{\mathbf{A}_{\mathrm{s}}}\ \underset{k\times n}{\mathbf{Z}}, \tag{1.7}$$

where $\mathbf{X}$ and $\mathbf{Z}$ are the same matrices as those that appear in (1.5) and (1.6), and where $\mathbf{A}_{\mathrm{s}}$ is the $k \times k$ submatrix of $\mathbf{A}$ given by

$$\mathbf{A}_{\mathrm{s}} = \mathbf{A}(I_{\mathrm{s}}, J_{\mathrm{s}}).$$

1.5.3. Computing the ID. In this section we demonstrate that there is a close connection between the column ID and the classical column pivoted QR factorization (CPQR). The end result is that standard software used to compute the CPQR can with some light post-processing be used to compute the column ID.

As a starting point, recall that for a given $m \times n$ matrix $\mathbf{A}$, with $m \geq n$, the QR factorization can be written as

$$\underset{m\times n}{\mathbf{A}}\ \underset{n\times n}{\mathbf{P}} = \underset{m\times n}{\mathbf{Q}}\ \underset{n\times n}{\mathbf{S}}, \tag{1.8}$$

where $\mathbf{P}$ is a permutation matrix, where $\mathbf{Q}$ has orthonormal columns and where $\mathbf{S}$ is upper triangular. (We use the letter $\mathbf{S}$ instead of the traditional $\mathbf{R}$ to avoid confusion with the "R"-factor in the row ID in (1.6).) Since our objective here is to construct a rank-k approximation to $\mathbf{A}$, we split off the leading k columns from $\mathbf{Q}$ and $\mathbf{S}$ to obtain partitions

$$\mathbf{Q} = \begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \left[\begin{array}{cc} \mathbf{Q}_1 & \mathbf{Q}_2 \end{array}\right] \end{array}, \qquad \text{and} \qquad \mathbf{S} = \begin{array}{c} \\ \begin{array}{c} k \\ m-k \end{array} \end{array} \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \left[\begin{array}{cc} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{0} & \mathbf{S}_{22} \end{array}\right] \end{array}. \tag{1.9}$$

Combining (1.8) and (1.9), we then find that

$$\mathbf{AP} = [\mathbf{Q}_1 \mid \mathbf{Q}_2] \begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{0} & \mathbf{S}_{22} \end{bmatrix} = [\mathbf{Q}_1\mathbf{S}_{11} \mid \mathbf{Q}_1\mathbf{S}_{12} + \mathbf{Q}_2\mathbf{S}_{22}]. \tag{1.10}$$

Equation (1.10) tells us that the $m \times k$ matrix $\mathbf{Q}_1\mathbf{S}_{11}$ consists precisely of first k columns of $\mathbf{AP}$. These columns were the first k columns that were chosen as "pivots" in the QR-factorization procedure. They typically form a good (approximate) basis for the column space of $\mathbf{A}$. We consequently define our $m \times k$ matrix $\mathbf{C}$ as this matrix holding the first k pivot columns. Letting J denote the permutation vector associated with the permutation matrix $\mathbf{P}$, so that

$$\mathbf{AP} = \mathbf{A}(:, J),$$

we define $J_{\mathrm{s}} = J(1 : k)$ as the index vector identifying the first k pivots, and set

$$\mathbf{C} = \mathbf{A}(:, J_{\mathrm{s}}) = \mathbf{Q}_1\mathbf{S}_{11}. \tag{1.11}$$

Now let us rewrite (1.10) by extracting the product $\mathbf{Q}_1\mathbf{S}_{11}$

$$\mathbf{AP} = \mathbf{Q}_1[\mathbf{S}_{11} \mid \mathbf{S}_{12}] + \mathbf{Q}_2[\mathbf{0} \mid \mathbf{S}_{22}] = \mathbf{Q}_1\mathbf{S}_{11}[\mathbf{I}_k \mid \mathbf{S}_{11}^{-1}\mathbf{S}_{12}] + \mathbf{Q}_2[\mathbf{0} \mid \mathbf{S}_{22}]. \tag{1.12}$$

(Remark 1.2 discusses why $\mathbf{S}_{11}$ must be invertible.) We define

$$\mathbf{T} = \mathbf{S}_{11}^{-1}\mathbf{S}_{12}, \qquad \text{and} \qquad \mathbf{Z} = [\mathbf{I}_k \mid \mathbf{T}]\mathbf{P}^* \tag{1.13}$$

so that (1.12) can be rewritten (upon right-multiplication by $\mathbf{P}^*$, which equals $\mathbf{P}^{-1}$ since $\mathbf{P}$ is unitary) as

$$\mathbf{A} = \mathbf{C}[\mathbf{I}_k \mid \mathbf{T}]\mathbf{P}^* + \mathbf{Q}_2[\mathbf{0} \mid \mathbf{S}_{22}]\mathbf{P}^* = \mathbf{CZ} + \mathbf{Q}_2[\mathbf{0} \mid \mathbf{S}_{22}]\mathbf{P}^*. \tag{1.14}$$

Equation (1.14) is precisely the column ID we sought, with the additional bonus that the remainder term is explicitly identified. Observe that when the spectral or Frobenius norms are used, the error term is of *exactly* the same size as the error term obtained from a truncated QR factorization:

$$\|\mathbf{A} - \mathbf{CZ}\| = \|\mathbf{Q}_2[\mathbf{0} \mid \mathbf{S}_{22}]\mathbf{P}^*\| = \|\mathbf{S}_{22}\| = \|\mathbf{A} - \mathbf{Q}_1[\mathbf{S}_{11} \mid \mathbf{S}_{12}]\mathbf{P}^*\|.$$

REMARK 1.2 (Conditioning). *Equation (1.12) involves the quantity $\mathbf{S}_{11}^{-1}$ which prompts the question of whether $\mathbf{S}_{11}$ is necessarily invertible, and what its condition number might be. It is easy to show that whenever the rank of $\mathbf{A}$ is at least k, the CPQR algorithm is guaranteed to result in a matrix $\mathbf{S}_{11}$ that is non-singular. (If the rank of $\mathbf{A}$ is j, where $j < k$, then the QR factorization process can detect this and halt the factorization after j steps.) Unfortunately, $\mathbf{S}_{11}$ is typically quite ill-conditioned. The saving grace is that even though one should expect $\mathbf{S}_{11}$ to be poorly conditioned, it is often the case that the linear system*

$$\mathbf{S}_{11}\mathbf{T} = \mathbf{S}_{12} \tag{1.15}$$

still has a solution $\mathbf{T}$ whose entries are of moderate size. Informally, one could say that the directions where $\mathbf{S}_{11}$ and $\mathbf{S}_{12}$ are small "line up." For standard column pivoted QR, the system (1.15) will in practice be observed to almost always have a solution $\mathbf{T}$ of small size [**7**]*, but counter-examples can be constructed. More sophisticated pivot selection procedures have been proposed that are guaranteed to result in matrices $\mathbf{S}_{11}$ and $\mathbf{S}_{12}$ such that (1.15) has a good solution; but these are harder to code and take longer to execute* [**17**].

Of course, the row ID can be computed via an entirely analogous process that starts with a CPQR of the *transpose* of $\mathbf{A}$. In other words, we execute a pivoted Gram-Schmidt orthonormalization process on the rows of $\mathbf{A}$.

Finally, to obtain the double-sided ID, we start with using the CPQR-based process to build the column ID (1.5). Then compute the row ID by performing Gram-Schmidt on the rows of the tall thin matrix $\mathbf{C}$.

The three deterministic algorithms described for computing the three flavors of ID are summarized in Figure 1.2.

REMARK 1.3 (Partial factorization). *The algorithms for computing interpolatory decompositions shown in Figure 1.2 are wasteful when $k \ll \min(m, n)$ since they involve a full QR factorization, which has complexity $O(mn \min(m, n))$. This problem is very easily remedied by replacing the full QR factorization by a partial QR factorization, which has cost $O(mnk)$. Such a partial factorization could take as input either a preset rank k, or a tolerance ε. In the latter case, the factorization would stop once the residual error $\|\mathbf{A} - \mathbf{Q}(:, 1:k)\mathbf{R}(1:k, :)\| = \|\mathbf{S}_{22}\| \leq \varepsilon$. When the QR factorization is interrupted after k steps, the output would still be a factorization of the form (1.8), but in this case, $\mathbf{S}_{22}$ would not be upper triangular. This is immaterial since $\mathbf{S}_{22}$ is never used. To further accelerate the computation, one can advantageously use a randomized algorithm, cf. Section 2.8 or or* [**30, 32**].

Compute a column ID so that $\mathbf{A} \approx \mathbf{A}(:, J_{\mathrm{s}})\,\mathbf{Z}$.
function $[J_{\mathrm{s}},\ \mathbf{Z}] = \texttt{ID\_col}(\mathbf{A}, k)$
$[\mathbf{Q}, \mathbf{S}, J] = \texttt{qr}(\mathbf{A}, 0)$;
$\mathbf{T} = (\mathbf{S}(1:k, 1:k))^{-1}\mathbf{S}(1:k, (k+1):n)$;
$\mathbf{Z} = \texttt{zeros}(k, n)$
$\mathbf{Z}(:, J) = [\mathbf{I}_k\ \mathbf{T}]$;
$J_{\mathrm{s}} = J(1:k)$;

Compute a row ID so that $\mathbf{A} \approx \mathbf{X}\,\mathbf{A}(I_{\mathrm{s}}, :)$.
function $[I_{\mathrm{s}},\ \mathbf{X}] = \texttt{ID\_row}(\mathbf{A}, k)$
$[\mathbf{Q}, \mathbf{S},\ J] = \texttt{qr}(\mathbf{A}^*, 0)$;
$\mathbf{T} = (\mathbf{S}(1:k, 1:k))^{-1}\mathbf{S}(1:k, (k+1):m)$;
$\mathbf{X} = \texttt{zeros}(m, k)$
$\mathbf{X}(J, :) = [\mathbf{I}_k\ \mathbf{T}]^*$;
$I_{\mathrm{s}} = J(1:k)$;

Compute a double-sided ID so that $\mathbf{A} \approx \mathbf{X}\,\mathbf{A}(I_{\mathrm{s}}, J_{\mathrm{s}})\,\mathbf{Z}$.
function $[I_{\mathrm{s}}, J_{\mathrm{s}}, \mathbf{X}, \mathbf{Z}] = \texttt{ID\_double}(\mathbf{A}, k)$
$[J_{\mathrm{s}},\ \mathbf{Z}] = \texttt{ID\_col}(\mathbf{A}, k)$;
$[I_{\mathrm{s}},\ \mathbf{X}] = \texttt{ID\_row}(\mathbf{A}(:, J_{\mathrm{s}}), k)$;

FIGURE 1.2. Deterministic algorithms for computing the column, row, and double-sided ID via the column pivoted QR factorization. The input is in every case an $m \times n$ matrix $\mathbf{A}$ and a target rank k. Since the algorithms are based on the CPQR, it is elementary to modify them to the situation where a tolerance rather than a rank is given. (Recall that the errors resulting from these ID algorithms are *identical* to the error in the first CPQR factorization executed.)

1.6. The Moore-Penrose pseudoinverse

The Moore-Penrose pseudoinverse is a generalization of the concept of an inverse for a non-singular square matrix. To define it, let $\mathbf{A}$ be a given $m \times n$ matrix. Let k denote its actual rank, so that its singular value decomposition (SVD) takes the form

$$\mathbf{A} = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^* = \mathbf{U}_k \mathbf{D}_k \mathbf{V}_k^*,$$

where $\sigma_1 \geq \sigma_2 \geq \sigma_k > 0$. Then the pseudoinverse of $\mathbf{A}$ is the $n \times m$ matrix defined via

$$\mathbf{A}^\dagger = \sum_{j=1}^{k} \frac{1}{\sigma_j} \mathbf{v}_j \mathbf{u}_j^* = \mathbf{V}_k \mathbf{D}_k^{-1} \mathbf{U}_k^*.$$

For any matrix $\mathbf{A}$, the matrices

$$\mathbf{A}^\dagger \mathbf{A} = \mathbf{V}_k \mathbf{V}_k^*, \qquad \text{and} \qquad \mathbf{A}\mathbf{A}^\dagger = \mathbf{U}_k \mathbf{U}_k^*,$$

are the orthogonal projections onto the row and column spaces of $\mathbf{A}$, respectively. If $\mathbf{A}$ is square and non-singular, then $\mathbf{A}^\dagger = \mathbf{A}^{-1}$. If $\mathbf{b} \in \mathbb{R}^m$ is any vector, then the vector $\mathbf{x} = \mathbf{A}^\dagger \mathbf{b}$ is the least-squares solution to the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$.

CHAPTER 2

Randomized methods for low rank approximation

2.1. Introduction

This chapter describes randomized techniques for computing approximate low-rank factorizations to matrices. To quickly introduce the key ideas, let us describe a simple prototypical randomized algorithm: Let $\mathbf{A}$ be a matrix of size $m \times n$ that is approximately of low rank. In other words, we assume that for some integer $k < \min(m, n)$, and some tolerance $\varepsilon > 0$, there exists a matrix $\mathbf{A}_k$ of rank k such that

$$\|\mathbf{A} - \mathbf{A}_k\| \leq \varepsilon.$$

Then a natural question is how do you in a computationally efficient manner construct such a rank-k approximating matrix $\mathbf{A}_k$? It was observed in [**27**] (which was inspired by [**12**], and later led to [**23, 28, 21**]) that random matrix theory provides a simple and elegant solution: Draw a *Gaussian random matrix* $\mathbf{G}$ of size $n \times k$ and form a *sampling matrix* $\mathbf{Y} = \mathbf{AG}$. Then in many important situations, the matrix

$$\underset{m \times n}{\mathbf{A}_k} := \underset{m \times k}{\mathbf{Y}} \underset{k \times n}{\left(\mathbf{Y}^{\dagger}\mathbf{A}\right)}, \tag{2.1}$$

where $\mathbf{Y}^{\dagger}$ is the Moore-Penrose pseudo-inverse of $\mathbf{Y}$, is close to optimal. With this observation as a starting point, one can construct highly efficient algorithms for computing approximate spectral decompositions of $\mathbf{A}$, for solving certain least-squares problems, for doing principal component analysis of large data sets, etc.

Many of the randomized sampling techniques we will describe are supported by rigorous mathematical analysis. For instance, it has been proved that the matrix $\mathbf{A}_k$ defined by (2.1) provides almost as good of an approximation to $\mathbf{A}$ as the best possible approximant of rank, say, $k - 5$, with probability almost 1. (The number "5" in "$k - 5$" results from a particular choice of a tuning parameter.) See [**21**, Sec. 10] and Section 2.5.

The algorithms that result from using randomized sampling techniques are computationally efficient, and are simple to implement as they rely on standard building blocks such as matrix-matrix multiplication, unpivoted QR factorization, etc, that are available for most computing environments (multicore CPU, GPU, distributed memory machines, etc). As an illustration, we invite the reader to peek ahead at Figure 2.1, which provides a complete Matlab code for a randomized algorithm that computes an approximate singular value decomposition of a matrix. Examples of improvements enabled by these randomized algorithms include:

- Given an $m \times n$ matrix $\mathbf{A}$, the cost of computing a rank-k approximant using classical methods is $O(mnk)$. Randomized algorithms attain complexity $O(mn \log k + k^2(m + n))$ [**21**, Sec. 6.1], and Section 2.4.
- Techniques for performing principal component analysis (PCA) of large data sets have been greatly accelerated, in particular when the data is stored out-of-core [**20**].
- Randomized methods tend to require less communication than traditional methods, and can be efficiently implemented on severely communication constrained environments such as GPUs [**32**] and even distributed computing platforms such as the Amazon EC2 Cloud computer [**19**, Ch. 4].
- Randomized algorithms enable *single-pass* matrix factorization in which the matrix is streamed and never stored, cf. [**21**, Sec. 6.3].

2.2. A two-stage approach

The problem of computing an approximate low-rank factorization to a given matrix can conveniently be split into two distinct stages. For concreteness, we describe the split for the specific task of computing an approximate singular value decomposition. To be precise, given an $m \times n$ matrix $\mathbf{A}$ and a target rank k, we seek to compute factors $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ such that

$$\underset{m\times n}{\mathbf{A}} \quad \approx \quad \underset{m\times k}{\mathbf{U}} \quad \underset{k\times k}{\mathbf{D}} \quad \underset{k\times n}{\mathbf{V}^*}.$$

The factors $\mathbf{U}$ and $\mathbf{V}$ should be orthonormal, and $\mathbf{D}$ should be diagonal. (For now, we assume that the rank k is known in advance, techniques for relaxing the assumption are described in Section 2.7.) Following [**21**], we split this task into two computational stages:

Stage A — find an approximate range: Build an $m \times k$ matrix $\mathbf{Q}$ with orthonormal columns such that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A}$. This step will be executed via a randomized process described in Section 2.3.

Stage B — form a specific factorization: Given the matrix $\mathbf{Q}$ from Stage A, form factors $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$, via classical deterministic techniques. For instance, proceed as follows:
(1) Form the $k \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.
(2) Decompose the matrix $\mathbf{B}$ in a singular value decomposition $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$.
(3) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

The point here is that in a situation where $k \ll \min(m, n)$, the difficult part of the computation is concentrated to Stage A. Once that is finished, the post-processing in Stage B is easy.

REMARK 2.1. *Stage B is exact up to floating point arithmetic so all errors in the factorization process are incurred at Stage A. In other words, if the factor* $\mathbf{Q}$ *satisfies*

$$||\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}|| \leq \varepsilon,$$

then the full factorization satisfies

$$||\mathbf{A} - \mathbf{U}\mathbf{D}\mathbf{V}^*|| \leq \varepsilon \tag{2.2}$$

unless ε *is close to the machine precision. Note that (2.2) does not in general guarantee that the computed singular vectors in the matrices* $\mathbf{U}$ *and* $\mathbf{V}$ *are within distance* ε *of the exact leading* k *singular vectors. In many (but not all) contexts, this is not a problem since only the error in the product* $\mathbf{U}\mathbf{D}\mathbf{V}^*$ *matters. (The computed singular values in* $\mathbf{D}$ *are within distance* ε *of the exact singular values, but note that for small singular values the relative precision may be poor.)*

2.3. A randomized algorithm for "Stage A" — the range finding problem

This section describes a randomized technique for solving the range finding problem introduced as "Stage A" in Section 2.2. As a preparation for this discussion, let us recall that an "ideal" basis matrix $\mathbf{Q}$ for the range of a given matrix $\mathbf{A}$ is the matrix $\mathbf{U}_k$ formed by the k leading left singular vectors of $\mathbf{A}$. Letting $\sigma_j(\mathbf{A})$ denote the j'th singular value of $\mathbf{B}$, the Eckart-Young theorem [**11, 36**], cf. Section 1.3.2, states that

$$\inf\{||\mathbf{A} - \mathbf{C}|| : \mathbf{C} \text{ has rank } k\} = ||\mathbf{A} - \mathbf{U}_k\mathbf{U}_k^*\mathbf{A}|| = \sigma_{k+1}(\mathbf{A}).$$

Now consider a simplistic randomized method for constructing a spanning set with k vectors for the range of a matrix $\mathbf{A}$: Draw k random vectors $\{\mathbf{g}_j\}_{j=1}^k$ from a Gaussian distribution, map these to vectors $\mathbf{y}_j = \mathbf{A}\mathbf{g}_j$ in the range of $\mathbf{A}$, and then use the resulting set $\{\mathbf{y}_j\}_{j=1}^k$ as a basis. Upon orthonormalization via, e.g., Gram-Schmidt, an orthonormal basis $\{\mathbf{q}_j\}_{j=1}^k$ would be obtained. If the matrix $\mathbf{A}$ has *exact* rank k, then the vectors $\{\mathbf{A}\mathbf{g}_j\}_{j=1}^k$ would with probability 1 be linearly independent, and the resulting ON-basis

ALGORITHM: RSVD — BASIC RANDOMIZED SVD

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 10$).

Outputs: Matrices $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate rank-$(k+p)$ SVD of $\mathbf{A}$. In other words, $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, $\mathbf{D}$ is diagonal, and $\mathbf{A} \approx \mathbf{U}\mathbf{D}\mathbf{V}^*$.

Stage A:

(1) Form an $n \times (k+p)$ Gaussian random matrix $\mathbf{G}$. `G = randn(n,k+p)`

(2) Form the sample matrix $\mathbf{Y} = \mathbf{A}\mathbf{G}$. `Y = A*G`

(3) Orthonormalize the columns of the sample matrix $\mathbf{Q} = \mathtt{orth}(\mathbf{Y})$. `[Q,~] = qr(Y,0)`

Stage B:

(4) Form the $(k+p) \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$. `B = Q'*A`

(5) Form the SVD of the small matrix $\mathbf{B}$: $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$. `[Uhat,D,V] = svd(B,'econ')`

(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$. `U = Q*Uhat`

FIGURE 2.1. A basic randomized algorithm. If a factorization of precisely rank k is desired, the factorization in Step 5 can be truncated to the k leading terms.

$\{\mathbf{q}_j\}_{j=1}^k$ would exactly span the range of $\mathbf{A}$. This would in a sense be an ideal algorithm. The problem is that in practice, there are almost always many non-zero singular values beyond the first k ones. These modes will shift the sample vectors $\mathbf{A}\mathbf{g}_j$ out of the space spanned by the k leading singular vectors of $\mathbf{A}$ and the process described can (and frequently does) produce a poor basis. Luckily, there is a fix: Simply take a few extra samples. It turns out that if we take, say, $k + 10$ samples instead of k, then the process will with probability almost 1 produce a basis that is comparable to the best possible basis.

To summarize the discussion in the previous paragraph, the randomized sampling algorithm for constructing an approximate rank k basis for the range of a given $m \times n$ matrix $\mathbf{A}$ proceeds as follows: First pick a small integer p representing how much "over-sampling" we do. (The choice $p = 10$ is often good.) Then execute the following steps:

(1) Form a set of $k + p$ random Gaussian vectors $\{\mathbf{g}_j\}_{j=1}^{k+p}$.

(2) Form a set $\{\mathbf{y}_j\}_{j=1}^{k+p}$ of samples from the range where $\mathbf{y}_j = \mathbf{A}\mathbf{g}_j$.

(3) Perform Gram-Schmidt on the set $\{\mathbf{y}_j\}_{j=1}^{k+p}$ to form the ON-set $\{\mathbf{q}_j\}_{j=1}^{k+p}$.

Now observe that the $k + p$ matrix-vector products are independent and can advantageously be executed in parallel. A full algorithm for computing an approximate SVD using this simplistic sampling technique for executing "Stage A" is summarized in Figure 2.1.

The error incurred by the randomized range finding method described in this section is a random variable. There exist rigorous bounds for both the expectation of this error, and for the likelihood of a large deviation from the expectation. These bounds demonstrate that when the singular values of $\mathbf{A}$ decay "reasonably fast," the error incurred is close to the theoretically optimal one. We provide more details in Section 2.5.

2.4. A method with complexity $O(mn \log k)$ for dense matrices that fit in RAM

The RSVD algorithm described in Figure 2.1 for constructing an approximate basis for the range of a given matrix $\mathbf{A}$ is highly efficient when we have access to fast algorithms for evaluating matrix-vector products $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$. For the case where $\mathbf{A}$ is a general $m \times n$ matrix given simply as an array or real numbers, the cost of evaluating the sample matrix $\mathbf{Y} = \mathbf{A}\mathbf{G}$ (in Step (2) of the algorithm in Figure 2.1) is $O(mnk)$. The algorithm is still often faster than classical methods since the matrix-matrix multiply can be highly

optimized, but it does not have an edge in terms of asymptotic complexity. However, it turns out to be possible to modify the algorithm by replacing the Gaussian random matrix $\mathbf{G}$ with a different random matrix $\mathbf{\Omega}$ that has two seemingly contradictory properties:

(1) $\mathbf{\Omega}$ is sufficiently *structured* that the product $\mathbf{A\Omega}$ can be evaluated in $O(mn\log(k))$ flops.
(2) $\mathbf{\Omega}$ is sufficiently *random* that the columns of $\mathbf{A\Omega}$ accurately span the range of $\mathbf{A}$.

For instance, a good choice $\mathbf{\Omega}$ is [**1, 23, 41**]

$$\underset{n\times\ell}{\mathbf{\Omega}} = \underset{n\times n}{\mathbf{D}}\ \underset{n\times n}{\mathbf{F}}\ \underset{n\times\ell}{\mathbf{S}}, \tag{2.3}$$

where $\mathbf{D}$ is a diagonal matrix whose diagonal entries are complex numbers of modulus one drawn from a uniform distribution on the unit circle in the complex plane, where $\mathbf{F}$ is the discrete Fourier transform,

$$\mathbf{F}(p,q) = n^{-1/2}\, e^{-2\pi \mathrm{i}(p-1)(q-1)/n}, \qquad p,q \in \{1,\, 2,\, 3,\, \ldots,\, n\},$$

and where $\mathbf{S}$ is matrix consisting of a random subset of ℓ columns from the $n\times n$ unit matrix (drawn without replacement). In other words, given an arbitrary matrix $\mathbf{X}$ of size $m\times n$, the matrix $\mathbf{XS}$ consists of a randomly drawn subset of ℓ columns of $\mathbf{X}$. For the matrix $\mathbf{\Omega}$ specified by (2.3), the product $\mathbf{X\Omega}$ can be evaluated via a subsampled FFT in $O(mn\log(\ell))$ operations. The parameter ℓ should be chosen slightly larger than the target rank k; the choice $\ell = 2k$ is often good.

By using the structured random matrix described in this section, we can reduce the complexity of "Stage A" in the RSVD from $O(mnk)$ to $O(mn\log k)$. We next need to modify "Stage B" to eliminate the need to compute $\mathbf{Q}^*\mathbf{A}$. One option is to use a "single pass" algorithm, as described in [**21**, Sec. 6.3], using the structured random matrix to approximate both the row and the column spaces of $\mathbf{A}$. A second, and typically better, option is to use a so called *row-extraction* technique for Stage B, we describe the details in Section 2.8.

The current error analysis for the accelerated range finder is less satisfactory than the one for Gaussian random matrices. In the general case, only very weak results can be proven. In practice, the accelerated scheme is often as accurate as the Gaussian one, but we do not currently have good theory to predict precisely when this happens, see [**21**, Sec. 11].

2.5. Theoretical performance bounds

In this section, we will briefly summarize some proven results concerning the error in the output of the basic RSVD algorithm in Figure 2.1. Observe that the factors $\mathbf{U}$, $\mathbf{D}$, $\mathbf{V}$ depend not only on $\mathbf{A}$, but also on the draw of the random matrix $\mathbf{G}$. This means that the error that we try to bound is a random variable. It is therefore natural to seek bounds on first the *expectation* of the error, and then on the likelihood of *large deviations* from the expectation.

Before we start, let us recall that Stage B is exact up to floating point arithmetic. This means that the entire error in the randomized SVD in Figure 2.1 is incurred in Stage A. To be precise:

$$\mathbf{A} - \mathbf{Q}\underbrace{\mathbf{Q}^*\mathbf{A}}_{=\mathbf{B}} = \mathbf{A} - \mathbf{Q}\ \underbrace{\mathbf{B}}_{=\hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*} = \mathbf{A} - \underbrace{\mathbf{Q}\hat{\mathbf{U}}}_{=\mathbf{U}}\mathbf{D}\mathbf{V}^* = \mathbf{A} - \mathbf{U}\mathbf{D}\mathbf{V}^*.$$

Consequently, we can (and will) restrict ourselves to providing bounds on $\|\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}\|$.

2.5.1. Bounds on the expectation of the error. For instance, Theorem 10.6 of [**21**] states:

THEOREM 2.1. *Let $\mathbf{A}$ be an $m \times n$ matrix with singular values $\{\sigma_j\}_{j=1}^{\min(m,n)}$. Let k be a target rank, and let p be an over-sampling parameter such that $p \geq 2$ and $k + p \leq \min(m, n)$. Let $\mathbf{G}$ be a Gaussian random matrix of size $n \times (k + p)$ and let $\mathbf{Q}$ denote an orthonormal matrix of size $m \times (k + p)$ whose columns form an orthonormal basis for the column space of the matrix $\mathbf{Y} = \mathbf{AG}$. (The matrix $\mathbf{Q}$ could for instance result from an unpivoted QR factorization, so that $[\mathbf{Q}, \sim] = qr(\mathbf{A}, 0)$.) Then the average error, as measured in the Frobenius norm, satisfies*

$$\mathbb{E}\big[\|\mathbf{A} - \mathbf{QQ}^*\mathbf{A}\|_{\mathrm{F}}\big] \leq \left(1 + \frac{k}{p-1}\right)^{1/2} \left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2}. \tag{2.4}$$

The corresponding result for the spectral norm reads

$$\mathbb{R}\big[\|\mathbf{A} - \mathbf{QQ}^*\mathbf{A}\|\big] \leq \left(1 + \sqrt{\frac{k}{p-1}}\right) \sigma_{k+1} + \frac{e\sqrt{k+p}}{p} \left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2}. \tag{2.5}$$

When errors are measured in the Frobenius norm, Theorem 2.1 is very gratifying. For our standard recommendation of $p = 10$, we are basically within a factor of $\sqrt{k/9}$ of the theoretically minimal error. (Recall that the Eckart-Young theorem states that $\left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2}$ is a lower bound on the residual for any rank-k approximant.) If you over-sample a little more aggressively and set $p = k$, then we are within a distance of $\sqrt{2}$ of the theoretically minimal error.

When errors are measured in the *spectral norm,* the situation is much less rosy. The first term in the bound in (2.5) is perfectly acceptable, but the second term is unfortunate in that it involves the minimal error in the Frobenius norm, which can be a much bigger factor. The theorem is quite sharp, as it turns out, so this disparity reflects a true problem for the basic randomized scheme.

The extent to which the suboptimality in (2.5) is problematic depends on how rapidly the "tail" singular values $\{\sigma_j\}_{j>k}$ decay. If they decay fast, then the spectral norm error and the Frobenius norm error are similar, and the RSVD works well. If they decay slowly, then the RSVD performs fine when errors are measured in the Frobenius norm, but not very well when the spectral norm is the one of interest. To illustrate the difference, let us consider two situations:

Case 1 — fast decay: Suppose that the tail singular values decay exponentially fast, so that for some $\beta \in (0, 1)$ we have $\sigma_j \approx \sigma_{k+1}\, \beta^{j-k-1}$ for $j > k$. Then $\left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2} \approx \sigma_{k+1} \left(\sum_{j=k+1}^{\min(m,n)} \beta^2\right)^{1/2} \leq \sigma_{k+1}(1 - \beta^2)^{-1/2}$. As long as β is not very close to 1, we see that the contribution from the tail singular values is modest in this case.

Case 2 — no decay: Suppose that the tail singular values exhibit *no* decay, so that $\sigma_j = \sigma_{k+1}$ for $j > k$. This represents the worst case scenario, and now $\left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2} = \sigma_{k+1}\sqrt{n-k}$. Since we want to allow for n to be very large (say $n = 10^6$), this represents a huge degree of suboptimality.

Fortunately, it is possible to modify the RSVD in such a way that the errors produced are close to optimal in both the spectral and the Frobenius norms. This is achieved by modestly increasing the computational cost. See Section 2.6 and [**21**, Sec. 4.5].

2.5.2. Bounds on the likelihood of large deviations. One can prove that (perhaps surprisingly) the likelihood of a large deviation from the mean depends only on the over-sampling parameter p, and decays extra-ordinarily fast. For instance, one can prove that if $p \geq 4$, then

$$||\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}|| \leq \left(1 + 17\sqrt{1+k/p}\right)\sigma_{k+1} + \frac{8\sqrt{k+p}}{p+1}\left(\sum\nolimits_{j>k}\sigma_j^2\right)^{1/2}, \tag{2.6}$$

with failure probability at most $3\,e^{-p}$, see [**21**, Cor. 10.9].

2.6. An accuracy enhanced randomized scheme

2.6.1. The key idea — power iteration. We mentioned earlier that the basic randomized scheme, as described in Figure 2.1, gives accurate results for matrices whose singular values decay rapidly, but tends to produce suboptimal results when they do not. The theoretical results summarized in Section 2.5 make this claim precise. To recapitulate the argument, suppose that we compute a rank-k approximation to an $m \times n$ matrix $\mathbf{A}$ with singular values $\{\sigma_j\}_j$. The theory shows that the error measured in the spectral norm behaves like $\left(\sum_{j>k}\sigma_j^2\right)^{1/2}$. When the singular values decay slowly, this quantity can be much larger than the theoretically minimal approximation error (which is σ_{k+1}).

Recall that the objective of the randomized sampling is to construct a set of ON vectors $\{\mathbf{q}_j\}_{j=1}^{\ell}$ that capture to high accuracy the space spanned by the k dominant left singular vectors $\{\mathbf{u}_j\}_{j=1}^{k}$ of $\mathbf{A}$. The idea is now to sample not $\mathbf{A}$, but the matrix

$$\mathbf{A}^{(q)} := \left(\mathbf{A}\mathbf{A}^*\right)^q\mathbf{A},$$

where q is a small positive integer (typically, $q = 1$ or $q = 2$). A simple calculation shows that if $\mathbf{A}$ has singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^*$, then the SVD of $\mathbf{A}^{(q)}$ is

$$\mathbf{A}^{(q)} = \mathbf{U}\,\mathbf{D}^{2q+1}\,\mathbf{V}^*.$$

In other words, $\mathbf{A}^{(q)}$ has the same left singular values as $\mathbf{A}$, while its singular values are $\{\sigma_j^{2q+1}\}_{j=1}^{\min(m,n)}$. Even when the singular values of $\mathbf{A}$ decay slowly, the singular values of $\mathbf{A}^{(q)}$ tend to decay fast enough for our purposes.

The accuracy enhanced scheme now consists of drawing a Gaussian matrix $\mathbf{G}$ and then forming a sample matrix

$$\mathbf{Y} = \left(\mathbf{A}\mathbf{A}^*\right)^q\mathbf{A}\mathbf{G}.$$

Then orthonormalize the columns of $\mathbf{Y}$ to obtain $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$, and proceed as before. The resulting scheme is shown in Figure 2.2.

REMARK 2.2. *The scheme described in Figure 2.2 can lose accuracy due to round-off errors. The problem is that as q increases, all columns in the sample matrix $\mathbf{Y} = \left(\mathbf{A}\mathbf{A}^*\right)^q\mathbf{A}\mathbf{G}$ tend to align closer and closer to the dominant left singular vector. This means that we lose almost all accuracy in regards to the directions of singular values associated with smaller singular vectors. Roughly speaking, if*

$$\frac{\sigma_j}{\sigma_1} \leq \epsilon_{\text{mach}}^{1/(2q+1)},$$

then all information associated with the j'th singular mode is lost. A standard technique for ameliorating this problem is to modify the scheme in Figure 2.2 by explicitly orthonormalizing the sample vectors between each iteration:

ALGORITHM: ACCURACY ENHANCED RANDOMIZED SVD

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, an over-sampling parameter p (say $p = 10$), and a small integer q denoting the number of steps in the power iteration.

Outputs: Matrices $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate rank-$(k+p)$ SVD of $\mathbf{A}$. (I.e. $\mathbf{U}$ and $\mathbf{V}$ are orthonormal and $\mathbf{D}$ is diagonal.)

(1) $\mathbf{G} = \texttt{randn}(n, k+p)$;
(2) $\mathbf{Y} = \mathbf{AG}$;
(3) **for** $j = 1 : q$
(4) $\quad \mathbf{Z} = \mathbf{A}^*\mathbf{Y}$;
(5) $\quad \mathbf{Y} = \mathbf{AZ}$;
(6) **end for**
(7) $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$;
(8) $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$;
(9) $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd}(\mathbf{B}, \texttt{'econ'})$;
(10) $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$;

FIGURE 2.2. The accuracy enhanced randomized SVD. If a factorization of precisely rank k is desired, the factorization in Step 5 can be truncated to the k leading terms.

(1) $\mathbf{G} = \texttt{randn}(n, k+p)$;
(2) $\mathbf{Q} = \texttt{orth}(\mathbf{AG})$;
(3) **for** $j = 1 : q$
(4) $\quad \mathbf{W} = \texttt{orth}(\mathbf{A}^*\mathbf{Q})$;
(5) $\quad \mathbf{Q} = \texttt{orth}(\mathbf{AW})$;
(6) **end for**
(7) $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$;
(8) $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd}(\mathbf{B}, \texttt{'econ'})$;
(9) $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$;

This scheme is more costly due to the calls to `orth`*. However, this orthonormalization can be executed using unpivoted Gram-Schmidt, which is quite fast.*

2.6.2. Theoretical results. A detailed error analysis of the scheme described in Figure 2.2 is provided in [**21**, Sec. 10.4]. In particular, the key theorem states the following:

THEOREM 2.1. *Let $\mathbf{A}$ denote an $m \times n$ matrix, let $p \geq 2$ be an over-sampling parameter, and let q denote a small integer. Draw a Gaussian matrix $\mathbf{G}$ of size $n \times (k+p)$, set $\mathbf{Y} = (\mathbf{AA}^*)^q\mathbf{AG}$, and let $\mathbf{Q}$ denote an $m \times (k+p)$ ON matrix resulting from orthonormalizing the columns of $\mathbf{Y}$. Then*

$$\mathbb{E}\big[\|\mathbf{A} - \mathbf{QQ}^*\mathbf{A}\|\big] \leq \left[\left(1 + \sqrt{\frac{k}{p-1}}\right)\sigma_{k+1}^{2q+1} + \frac{e\sqrt{k+p}}{p}\left(\sum\nolimits_{j>k}\sigma_j^{2(2q+1)}\right)^{1/2}\right]^{1/(2q+1)}. \tag{2.7}$$

The bound in (2.7) is slightly hard to decipher. To simplify it, let us consider the worst case scenario where there is no decay in the singular values beyond the truncation point, so that $\sigma_{k+1} = \sigma_{k+2} = \cdots =$

$\sigma_{\min(m,n)}$. Then (2.7) simplifies to

$$\mathbb{E}\big[\|\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}\|\big] \leq \left[1 + \sqrt{\frac{k}{p-1}} + \frac{e\sqrt{k+p}}{p} \cdot \sqrt{\min\{m,n\} - k}\right]^{1/(2q+1)} \sigma_{k+1}.$$

In other words, as we increase the exponent q, the power scheme drives factor that multiplies $\sigma_{k=1}$ to one exponentially fast. This factor represents the degree of "sub-optimality" you can expect to see.

2.6.3. Extended sampling matrix. The scheme described in Section 2.6.1 is slightly wasteful in that it does not directly use all the sampling vectors computed. To further improve accuracy, let us form an "extended" sampling matrix

$$\mathbf{Y} = \big[\mathbf{A}\mathbf{G},\ (\mathbf{A}\mathbf{A}^*)\mathbf{A}\mathbf{G},\ (\mathbf{A}\mathbf{A}^*)^2\mathbf{A}\mathbf{G},\ \ldots,\ (\mathbf{A}\mathbf{A}^*)^q\mathbf{A}\mathbf{G}\big].$$

Observe that this new sampling matrix $\mathbf{Y}$ has $q\ell$ columns. Then proceed as before:

$$\mathbf{Q} = \mathtt{qr}(\mathbf{Y}), \qquad \mathbf{B} = \mathbf{Q}^*\mathbf{A}, \qquad [\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \mathtt{svd}(\mathbf{B}, \mathtt{'econ'}), \qquad \mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}.$$

Note that these computations are all much more expensive than those in Section 2.6.1 since we now work with matrices with $q\ell$ columns, as opposed to ℓ columns earlier. Since the cost of QR factorization, etc, grows quadratically with the number of columns, the difference is very substantial — the cost of dense linear algebra increases by a factor of $O(q^2)$. Consequently, the scheme described here is primarily useful in situations where the computational cost is dominated by applications of $\mathbf{A}$ and $\mathbf{A}^*$, and we want to maximally leverage all interactions with $\mathbf{A}$.

2.7. Adaptive rank determination with updating of the matrix

2.7.1. Problem formulation. Up to this point, we have assumed that the rank k is given as an input variable to the factorization algorithm. In practical usage, it is common that we are given instead a matrix $\mathbf{A}$ and a computational tolerance ε, and our task is then to determine a matrix $\mathbf{B}_k$ of rank k such that $\|\mathbf{A}-\mathbf{B}_k\| \leq \varepsilon$.

The techniques described in this section are designed for dense matrices stored in RAM. They directly update the matrix, and come with a firm guarantee that the computed low rank approximation is within distance ε of the original matrix. There are many situations where direct updating is not feasible and we can in practice only interact with the matrix via the matrix-vector multiplication (e.g., very large matrices stored out-of-core, sparse matrices, matrices that are defined implicitly). In this environment, one can advantageously use randomized sampling techniques to *estimate* to approximation error, cf. [**26**, Sec. 12.4].

Recall that for the case where a computational tolerance is given (rather than a rank), the optimal solution is given by the SVD. Specifically, let $\{\sigma_j\}_{k=1}^{\min(m,n)}$ be the singular values of $\mathbf{A}$. Then the minimal rank k for which

$$\inf\{\|\mathbf{A} - \mathbf{B}\| :\ \mathbf{B} \text{ has rank } k\} \leq \varepsilon$$

is the smallest integer k such that $\sigma_{k+1} \leq \varepsilon$. The algorithms described here will determine a k that is not necessarily optimal, but is typically fairly close.

2.7.2. A basic updating algorithm. Let us start by describing a very general algorithmic template for how to compute an approximate rank-k approximate factorization of a matrix. To be precise, suppose that we are given an $m \times n$ matrix $\mathbf{A}$, and a computational tolerance ε. Our objective is then to determine an integer $k \in \{1, 2, \ldots, \min(m,n)\}$, an $m \times k$ ON matrix $\mathbf{Q}_k$, and a $k \times n$ matrix $\mathbf{B}_k$ such that

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times k}{\mathbf{Q}_k}\ \underset{k\times n}{\mathbf{B}_k}.$$

(1) $\mathbf{Q}_0 = [\,]$; $\mathbf{B}_0 = [\,]$; $\mathbf{A}_0 = \mathbf{A}$; $k = 0$;
(2) **while** $\|\mathbf{A}_k\| > \varepsilon$
(3) $\quad k = k + 1$
(4) $\quad$ Pick a vector $\mathbf{y} \in \mathrm{Ran}(\mathbf{A}_{k-1})$
(5) $\quad \mathbf{q} = \mathbf{y}/\|\mathbf{y}\|$.
(6) $\quad \mathbf{b} = \mathbf{q}^*\mathbf{A}_{k-1}$
(7) $\quad \mathbf{Q}_k = [\mathbf{Q}_{k-1}\ \mathbf{q}]$
(8) $\quad \mathbf{B}_k = \begin{bmatrix} \mathbf{B}_{k-1} \\ \mathbf{b} \end{bmatrix}$
(9) $\quad \mathbf{A}_k = \mathbf{A}_{k-1} - \mathbf{q}\mathbf{b}$
(10) **end for**

FIGURE 2.3. A greedy algorithm for building a low-rank approximation to a given $m \times n$ matrix $\mathbf{A}$ that is accurate to within a given precision ε. To be precise, the algorithm determines an integer k, an $m \times k$ ON matrix $\mathbf{Q}_k$ and a $k \times n$ matrix $\mathbf{B}_k = \mathbf{Q}_k^*\mathbf{A}$ such that $\|\mathbf{A} - \mathbf{Q}_k\mathbf{B}_k\| \leq \varepsilon$. One can easily verify that after the algorithm finishes, we have $\mathbf{A} = \mathbf{Q}_j\mathbf{B}_j + \mathbf{A}_j$ for any $j = 0, 1, 2, \ldots, k$.

This problem can be solved using the greedy algorithm shown in Figure 2.3 which builds $\mathbf{Q}_k$ and $\mathbf{B}_k$ one column and row at a time.

The algorithm described in Figure 2.3 is a generalization of the basic Gram-Schmidt procedure described in Figure 1.1. The key to understanding how the algorithm works is provided by the identity

$$\mathbf{A} = \mathbf{Q}_j\mathbf{B}_j + \mathbf{A}_j, \qquad j = 0, 1, 2, \ldots, k.$$

The computational efficiency and accuracy of the algorithm depends crucially on how the vector $\mathbf{y}$ is picked on line (4). Let us consider three possible selection strategies:

Pick the largest remaining column. Suppose we instantiate line (4) by letting $\mathbf{y}$ be simply the largest column of the remainder matrix $\mathbf{A}_{k-1}$.

$$\text{(4)} \qquad \text{Set } j_k = \mathrm{argmax}\{\|\mathbf{A}_{k-1}(:, j)\| :\ j = 1, 2, \ldots, n\} \text{ and then } \mathbf{y} = \mathbf{A}_{k-1}(:, j_k).$$

With this choice, the algorithm in Figure 2.3 is *precisely* column pivoted Gram-Schmidt (CPQR). This algorithm is reasonably efficient, and often leads to fairly close to optimal low-rank approximation. For instance, when the singular values of $\mathbf{A}$ decay rapidly, CPQR typically determines a numerical rank k that is typically reasonably close to the theoretically exact ε-rank. However, this is not always the case even when the singular values decay rapidly, and the results can be quite poor when the singular values decay slowly.

Pick the locally optimal vector. A choice that is natural, and is conceptually very simple is to pick the vector $\mathbf{y}$ by solving the obvious minimization problem:

$$\text{(4)} \qquad \mathbf{y} = \mathrm{argmin}\{\|\mathbf{A}_{k-1} - \mathbf{y}\mathbf{y}^*\mathbf{A}_{k-1}\| :\ \|\mathbf{y}\| = 1\}.$$

With this choice, the algorithm will produce matrices that attain the theoretically optimal precision

$$\|\mathbf{A} - \mathbf{Q}_j\mathbf{B}_j\| = \sigma_{j+1}.$$

This tells us that the greediness of the algorithm is not a problem. However, solving the local minimization problem is sufficiently computationally hard that this algorithm is not particularly practical.

(1) $\mathbf{Q} = [\,]; \mathbf{B} = [\,];$
(2) **while** $\|\mathbf{A}\| > \varepsilon$
(3) Draw an $n \times \ell$ random matrix $\mathbf{R}$.
(4) Compute the $m \times \ell$ matrix $\mathbf{Q}_{\text{new}} = \texttt{qr}(\mathbf{AR}, 0)$.
(5) $\mathbf{B}_{\text{new}} = \mathbf{Q}_{\text{new}}^* \mathbf{A}$
(6) $\mathbf{Q} = [\mathbf{Q}\ \mathbf{Q}_{\text{new}}]$
(7) $\mathbf{B} = \begin{bmatrix} \mathbf{B} \\ \mathbf{B}_{\text{new}} \end{bmatrix}$
(8) $\mathbf{A} = \mathbf{A} - \mathbf{Q}_{\text{new}} \mathbf{B}_{\text{new}}$
(9) **end while**

FIGURE 2.4. A greedy algorithm for building a low-rank approximation to a given $m \times n$ matrix $\mathbf{A}$ that is accurate to within a given precision ε. This algorithm is a blocked analogue of the method described in Figure 2.3 and takes as input a block size ℓ. Its output is an ON matrix $\mathbf{Q}$ of size $m \times k$ (where k is a multiple of ℓ) and a $k \times n$ matrix $\mathbf{B}$ such that $\|\mathbf{A} - \mathbf{QB}\| \leq \varepsilon$. For higher accuracy, one can incorporate a couple of steps of power iteration and set $\mathbf{Q}_{\text{new}} = \texttt{qr}((\mathbf{AA}^*)^q \mathbf{AR}, 0)$ on Line (4).

A randomized selection strategy. Suppose now that we pick $\mathbf{y}$ by forming a linear combination of the columns of $\mathbf{A}_{k-1}$ with the expansion weights drawn from a normalized Gaussian distribution:

(4) Draw a Gaussian random vector $\mathbf{g} \in \mathbb{R}^n$ and set $\mathbf{y} = \mathbf{A}_{k-1}\mathbf{g}$.

With this choice, the algorithm becomes logically equivalent to the basic randomized SVD given in Figure 2.1. This means that this choice often leads to a factorization that is close to optimally accurate, and is also computationally efficient. One can attain higher accuracy by trading away some computational efficiency by incorporating a couple of steps of power iteration, and choosing $\mathbf{y} = \left(\mathbf{A}_{k-1}\mathbf{A}_{k-1}^*\right)^q \mathbf{A}_{k-1}\mathbf{g}$ for some small integer q.

2.7.3. A blocked updating algorithm. A key benefit of the randomized greedy algorithm described in Section 2.7.2 is that it can easily be *blocked*. In other words, given a block size ℓ, we can at each step of the iteration draw a set of ℓ Gaussian random vectors, compute the corresponding sample vectors, and then extend the factors $\mathbf{Q}$ and $\mathbf{B}$ by adding ℓ columns and ℓ rows at a time, respectively. The resulting algorithm is shown in Figure 2.4.

2.8. Computing interpolative decompositions via randomized sampling

The randomized range finder described in Section 2.3 is particularly effective when used in conjunction with the interpolative decomposition. To illustrate the process, consider for a moment the task of factorizing an $m \times n$ matrix $\mathbf{A}$ of *exact* rank k (we will discuss the case of approximate rank shortly). Using the randomized sampling technique, we would then draw an $n \times k$ random matrix $\mathbf{G}$, form a sample matrix $\mathbf{Y} = \mathbf{AG}$, orthonormalize its columns to form the matrix $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$, and finally form the rank-k factorization

$$\mathbf{A} \approx \mathbf{Q}\left(\mathbf{Q}^*\mathbf{A}\right).$$

This process requires the evaluation of the matrix-matrix product $\mathbf{Q}^*\mathbf{A}$. It turns out that by using the ID instead, we can skip this step. Instead of applying Gram-Schmidt to the *columns* of $\mathbf{Y}$, we apply Gram-Schmidt to the *rows* of $\mathbf{Y}$ to find a good spanning set of rows. To be precise, we execute the command

$$[\mathbf{X}, I_{\text{s}}] = \texttt{ID\_row}(\mathbf{Y}, k),$$

ALGORITHM: RANDOMIZED ID

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, an over-sampling parameter p (say $p = 10$), and a small integer q denoting the number of power iterations taken.

Outputs: An $m \times k$ interpolation matrix $\mathbf{X}$ and an index vector $I_{\rm s} \in \mathbb{N}^k$ such that $\mathbf{A} \approx \mathbf{X}\mathbf{A}(I_{\rm s}, :)$.

(1) $\mathbf{G} = \texttt{randn}(n, k+p)$;
(2) $\mathbf{Y} = \mathbf{A}\mathbf{G}$;
(3) **for** $j = 1 : q$
(4) $\quad \mathbf{Z} = \mathbf{A}^*\mathbf{Y}$;
(5) $\quad \mathbf{Y} = \mathbf{A}\mathbf{Z}$;
(6) **end for**
(7) Form an ID of the $n \times (k+p)$ sample matrix: $[\mathbf{X}, I_{\rm s}] = \texttt{ID\_row}(\mathbf{Y}, k)$.

FIGURE 2.5. An $O(mnk)$ algorithm for computing an interpolative decomposition of $\mathbf{A}$ via randomized sampling. For $q = 0$, the scheme is fast (but not quite as fast as the scheme in Figure 2.6) and accurate for matrices whose singular values decay rapidly. For matrices whose singular values decay slowly, one should pick a larger q (say $q = 1$ or 2) to improve accuracy at the cost of longer execution time. If accuracy better than $\epsilon_{\rm mach}^{1/(2q+1)}$ is desired, then the scheme should be modified to incorporate orthonormalization as described in Remark 2.2.

where `ID_row` is as defined in Figure 1.2. Then

$$\mathbf{Y} = \mathbf{X}\mathbf{Y}(I_{\rm s}, :). \tag{2.8}$$

Now, automatically (!), the couple $\{\mathbf{X}, I_{\rm s}\}$ also forms an ID for $\mathbf{A}$,

$$\mathbf{A} = \mathbf{X}\mathbf{A}(I_{\rm s}, :). \tag{2.9}$$

(See Remark 2.3 for details.) The beauty here is that once we have the sample matrix $\mathbf{Y}$, we do not need to revisit all of $\mathbf{A}$ to construct a rank-k factorization — all we need to do is to extract the k rows of $\mathbf{A}$ indicated by the index vector $I_{\rm s}$.

By combining the idea of using an SRFT for "Stage A" (at complexity $O(mn\log(k))$) with an ID for "Stage B" (at complexity $O((m+n)k^2)$), we get the accelerated algorithm for constructing an ID described in Figure 2.6. With a slight modification to Stage B, we get the $O(mn\log(k))$ algorithm for computing an SVD described in Figure 2.7. More details (including an error analysis for the case when $\mathbf{A}$ has only approximate rank k) can be found in [**21**, Sec. 5.2].

REMARK 2.3. *It is perhaps not immediately clear why the ID $\{\mathbf{X}, I_{\rm s}\}$ computed for $\mathbf{Y}$ should automatically also be an ID for $\mathbf{A}$. To see why this is the case, let us first observe that since the columns of $\mathbf{Y}$ form a basis for the columns of $\mathbf{A}$, there must exist a $k \times n$ matrix $\mathbf{F}$ such that*

$$\mathbf{A} = \mathbf{Y}\mathbf{F}. \tag{2.10}$$

Now insert the relation (2.8) into (2.10):

$$\mathbf{A} = \mathbf{X}\mathbf{Y}(I_{\rm s}, :)\mathbf{F}. \tag{2.11}$$

Restrict (2.11) to the rows in $I_{\rm s}$, and exploit that $\mathbf{X}(I_{\rm s}, :) = \mathbf{I}_k$ to find

$$\mathbf{A}(I_{\rm s}, :) = \mathbf{X}(I_{\rm s}, :)\mathbf{Y}(I_{\rm s}, :)\mathbf{F} = \mathbf{Y}(I_{\rm s}, :)\mathbf{F}. \tag{2.12}$$

Finally, insert (2.12) in (2.11) to attain (2.9).

ALGORITHM: FAST RANDOMIZED ID

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = k$).

Outputs: An $m \times k$ interpolation matrix $\mathbf{X}$ and an index vector $I_{\mathrm{s}} \in \mathbb{N}^k$ such that $\mathbf{A} \approx \mathbf{X}\mathbf{A}(I_{\mathrm{s}}, :)$.

Stage A:

(1) Form an $n \times (k + p)$ SRFT $\mathbf{\Omega}$.
(2) Form the sample matrix $\mathbf{Y} = \mathbf{A}\,\mathbf{\Omega}$.

Stage B:

(3) Form an ID of the $n \times (k + p)$ sample matrix: $[\mathbf{X}, I_{\mathrm{s}}] = \texttt{ID\_row}(\mathbf{Y}, k)$.

FIGURE 2.6. An $O(mn \log(k))$ algorithm for computing an interpolative decomposition of $\mathbf{A}$.

ALGORITHM: FAST RANDOMIZED SVD

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = k$).

Outputs: Matrices $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ in an approximate rank-$(k+p)$ SVD of $\mathbf{A}$. (I.e. $\mathbf{U}$ and $\mathbf{V}$ are orthonormal and $\mathbf{\Sigma}$ is diagonal.)

Stage A:

(1) Form an $n \times (k + p)$ SRFT $\mathbf{\Omega}$.
(2) Form the sample matrix $\mathbf{Y} = \mathbf{B}\,\mathbf{\Omega}$.

Stage B:

(3) Form an ID of the sample matrix $\mathbf{Y}$: $[\mathbf{X}, I_{\mathrm{s}}] = \texttt{ID\_row}(\mathbf{Y}, k)$.
(4) Compute the QR decomposition of the interpolation matrix $[\mathbf{Q}, \mathbf{R}] = \texttt{qr}(\mathbf{X})$.
(5) Extract $k + p$ rows of $\mathbf{A}$: $\mathbf{A}^{\mathrm{rows}} = \mathbf{A}(I_{\mathrm{s}}, :)$.
(6) Multiply $\mathbf{R}$ and $\mathbf{A}^{\mathrm{rows}}$ to form the $(k + p) \times n$ matrix $\mathbf{F} = \mathbf{R}\mathbf{A}^{\mathrm{rows}}$.
(7) Decompose the matrix $\mathbf{F}$ in a singular value decomposition $[\hat{\mathbf{U}}, \mathbf{\Sigma}, \mathbf{V}] = \texttt{svd}(\mathbf{F})$.
(8) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

FIGURE 2.7. An $O(mn \log(k))$ algorithm for computing a partial SVD. If an SVD of exactly rank k is desired, then truncate the factors computed in Step (7).

2.9. References

Our objective in this chapter has been to describe randomized methods that attain high practical computational efficiency. In particular, we have used randomization mostly as a tool for minimizing *communication*, rather than minimizing the flop count (although as we saw in Section 2.4, remarkably reductions in the asymptotic flop count can be attained as well). The methods described were first published in [**27**] (which was inspired by [**12**], and later led to [**23, 28**]; see also [**34**]). Our presentation largely follows that in the 2011 survey [**21**], but with a focus more on practical usage, rather than theoretical analysis. We have also included material from more recent work, including [**39**] on factorizations that allow for better data interpretation, [**32**] on blocking and adaptive error estimation, and [**29, 30**] on full factorizations.

The idea of using randomization to improve algorithms for low-rank approximation of matrices has been extensively investigated within the theoretical computer science community, with early work including [**10, 12, 34**]. The focus of these texts has been to develop algorithms with optimal or close to optimal theoretical performance guarantees in terms of asymptotic flop counts and error bounds. The surveys [**24, 9**] and [**40**] provide introductions to this literature. A highly accessible introduction to the topic that covers several viewpoints can be found in [**37**]. The text in this chapter is in part adapted from [**26**].

CHAPTER 3

Fast algorithms for rank structured matrices

3.1. Introduction

All direct solvers described in this text make frequent use of matrix operations such as matrix-matrix multiplications, matrix inversions, LU factorizations, etc. The matrices that are manipulated are often dense, but fortunately, most of these dense matrices will be either of small size, or will have internal structure that allows the required operations to be performed rapidly even though the matrices are dense. To be precise, the "internal structure" that is exploited is that off-diagonal blocks of the matrices can be well approximated by matrices of low rank. The chapter will illustrate the key ideas by introducing a very simple family of "compressible" matrices, and then showing how to rapidly perform algebraic operations on matrices in this family.

There is an extensive literature on "structured matrix computations." The type of structure we discuss in this chapter is closely related to the well-established $\mathcal{H}$ and $\mathcal{H}^2$ matrices of Hackbusch and co-workers [**4, 6, 15, 18**]. The format we consider in this initial discussion is a particularly simple special case of an $\mathcal{H}$-matrix. In the literature on $\mathcal{H}$-matrices, it would be referred as an "$\mathcal{H}$-matrix with a weak admissibility condition." Matrices in this family have also been referred to as "Hierarchically Off-Diagonal Low Rank (HODLR)" [**3**, Sec. 3.1], which is the term we will use in this chapter. Section 3.9 provides additional pointers to the literature.

3.2. Inversion of a 2×2 block matrix

As a warm-up, suppose that we seek to invert a 2×2 block matrix $\mathbf{A}$ of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix},$$

where the off-diagonal blocks $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ have low rank. (In applications, a typical matrix under consideration have off-diagonal blocks of low *numerical* rank, but to keep the discussion accessible we for simplicity assume low exact rank.) To be specific, let us assume that each block is of size $n \times n$, and that the off-diagonal blocks have rank k and admit factorizations

$$\underset{n\times n}{\mathbf{A}_{12}} = \underset{n\times k}{\mathbf{U}_1}\ \underset{k\times k}{\tilde{\mathbf{A}}_{12}}\ \underset{k\times n}{\mathbf{V}_2^*}, \quad \text{and} \quad \underset{n\times n}{\mathbf{A}_{21}} = \underset{n\times k}{\mathbf{U}_2}\ \underset{k\times k}{\tilde{\mathbf{A}}_{21}}\ \underset{k\times n}{\mathbf{V}_1^*}.$$

First observe that we can write $\mathbf{A}$ as a sum of one matrix that is block-diagonal, and one matrix that has overall low rank,

$$\mathbf{A} = \underbrace{\begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix}}_{\textit{block diagonal}} + \underbrace{\begin{bmatrix} \mathbf{U}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 \end{bmatrix} \begin{bmatrix} \mathbf{0} & \tilde{\mathbf{A}}_{12} \\ \tilde{\mathbf{A}}_{21} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^* & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^* \end{bmatrix}}_{\textit{rank}=2k}. \tag{3.1}$$

Now let us recall that the so called "Woodbury formula" provides a handy tool for inverting a matrix that is of the form "easy to invert" plus "low rank". One instantiation of the Woodbury formula that is particularly convenient in this context is given in the following theorem:

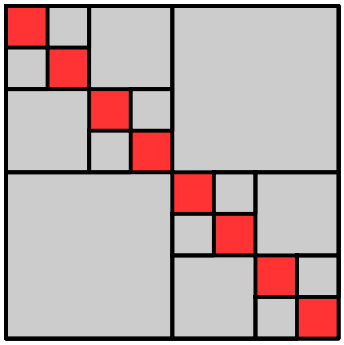

FIGURE 3.1. A rank structured matrix. The diagonal blocks (red) are dense general matrices, while the off-diagonal blocks have low (numerical) rank.

THEOREM 3.1. *Suppose* $\mathbf{X}$ *is a non-singular matrix that can be written in the form*

$$\underset{m\times m}{\mathbf{X}} = \underset{m\times m}{\mathbf{D}} + \underset{m\times k}{\mathbf{U}}\ \underset{k\times k}{\tilde{\mathbf{A}}}\ \underset{k\times m}{\mathbf{V}^*}, \tag{3.2}$$

where ℓ *is a positive integer such that* $\ell < m$*. Suppose further that the matrices* $\mathbf{D}$ *and* $\tilde{\mathbf{A}} + \mathbf{V}^*\mathbf{D}^{-1}\mathbf{U}$ *are invertible. Then*

$$\underset{m\times m}{\mathbf{X}^{-1}} = \underset{m\times m}{\mathbf{D}^{-1}} - \underset{m\times \ell}{\mathbf{D}^{-1}\mathbf{U}}\ \underset{\ell\times\ell}{\left(\tilde{\mathbf{A}} + \mathbf{V}^*\mathbf{D}^{-1}\mathbf{U}\right)^{-1}}\ \underset{\ell\times m}{\mathbf{V}^*\mathbf{D}^{-1}}. \tag{3.3}$$

Applying formula (3.3) to (3.1), we find that

$$\underset{2n\times 2n}{\mathbf{A}^{-1}} = \underset{2n\times 2n}{\begin{bmatrix} \mathbf{A}_{11}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22}^{-1} \end{bmatrix}} - \underset{2n\times 2k}{\begin{bmatrix} \mathbf{A}_{11}^{-1}\mathbf{U}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22}^{-1}\mathbf{U}_2 \end{bmatrix}} \underset{2k\times 2k}{\begin{bmatrix} \mathbf{S}_1 & \tilde{\mathbf{A}}_{12} \\ \tilde{\mathbf{A}}_{21} & \mathbf{S}_2 \end{bmatrix}^{-1}} \underset{2k\times 2n}{\begin{bmatrix} \mathbf{V}_1^*\mathbf{A}_{11}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^*\mathbf{A}_{22}^{-1} \end{bmatrix}}, \tag{3.4}$$

where

$$\mathbf{S}_1 = \mathbf{V}_1^*\,\mathbf{A}_{11}^{-1}\,\mathbf{U}_1, \qquad \text{and} \qquad \mathbf{S}_2 = \mathbf{V}_2^*\,\mathbf{A}_{22}^{-1}\,\mathbf{U}_2.$$

In terms of computational effort, we have reduced the task of inverting a matrix of size $2n \times 2n$ to the following tasks:

(a) Inverting two matrices, each of size $n \times n$.
(b) Inverting one matrix of size $2k \times 2k$.
(c) Matrix-matrix multiplications with at least one dimension at most k.
(d) Adding two matrices of size $2n \times 2n$.

Assuming that k is much smaller than n, the cost of task (a) is the dominant one. In effect, we have reduced the task of inverting one matrix of size $2n \times 2n$ to the task of inverting two matrices of size $n \times n$. In general, the cost of matrix inversion scales cubically. Let c denote the scaling constant so that the cost of inverting a matrix of size $m \times m$ is asymptotically cm^3. Then we have reduced the cost from $c(2n)^3 = 8cn^3$ to $2cn^3$, which is a savings by a factor of 4. To attain further savings, we will next apply the formula recursively.

3.3. Hierarchically off-diagonal low rank (HODLR) matrices

In this section, we build on the inversion algorithm of a 2×2 matrix described in Section 3.2, but further assume that the diagonal blocks $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ have the same block structure as $\mathbf{A}$ itself, and apply the idea recursively. Informally, this amounts to assuming that the overall matrix $\mathbf{A}$ allows a tessellation into blocks

Given a vector $\mathbf{q}$ and a HODLR matrix $\mathbf{A}$, form $\mathbf{u} = \mathbf{A}\mathbf{q}$.

(1) **function u =** `matvec`(**A**,**q**)
(2) **if** ($\dim(\mathbf{A}) < 2k$) **then**
(3) Evaluate by brute force: $\mathbf{u} = \mathbf{A}\mathbf{q}$.
(4) **else**
(5) Split $\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$ and $\mathbf{q} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \end{bmatrix}$.
(6) $\mathbf{u}_1 = \texttt{matvec}(\mathbf{A}_{11}, \mathbf{q}_1) + \mathbf{A}_{12}\mathbf{q}_2$.
(7) $\mathbf{u}_2 = \texttt{matvec}(\mathbf{A}_{22}, \mathbf{q}_2) + \mathbf{A}_{21}\mathbf{q}_1$.
(8) $\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}$.
(9) **end if**
(10) **end function**

FIGURE 3.2. Matrix-vector multiplication for a HODLR matrix.

like the one shown in Figure 3.1. Here, each diagonal block has been recursively split into a 2×2 block matrix until the diagonal blocks are themselves of size at most $2k \times 2k$.

DEFINITION 2. Let $\mathbf{A}$ be a matrix of size $N \times N$, and let k be an integer such that $k < N$. We then say that $\mathbf{A}$ is a *hierarchically off-diagonal low rank (HODLR)* matrix with rank k, if either of the following two conditions hold:

- $\mathbf{A}$ is itself of size at most $2k \times 2k$.
- If $\mathbf{A}$ is partitioned into four equi-sized blocks,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix},$$

  then $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ have rank at most k, and $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ are HODLR matrices of rank k.

This recursive definition is intuitively simple. An algorithm for executing a matrix-vector multiplication following the same notation is given in Figure 3.2. We will in Section 3.6 prove that the asymptotic cost of executing the matrix-vector multiplication is $O(kN \log N)$, and that this is also how much storage is required for a HODLR matrix.

REMARK 3.1. *For notational simplicity, we assume in this chapter that the ranks of all off-diagonal blocks are the same. It is a simple matter to use adaptively tuned ranks when implementing the algorithms.*

3.4. Inversion of compressible matrices

A HODLR matrix can be inverted by recursive application of the formula (3.4) for inversion of a 2×2 block matrix. There is one catch, however, in that the formula involves a low rank update that could potentially increase the ranks of the off-diagonal blocks. To illustrate, suppose that $\mathbf{A}_{11}^{-1}$ and $\mathbf{A}_{22}^{-1}$ are each HODLR matrices of rank k. The potential snag is that when we add the second term, we add a term of rank k to each off-diagonal block, potentially raising the rank from k to $2k$. For a general matrix, this is a genuine issue since there is no guarantee that the inverse of a rank-k HODLR matrix is necessarily a HODLR matrix of rank k. However, in practice, we often know from, e.g, the physics of the problem that the inverse *should* be compressible. When this is the case, we can combat the potential increase in rank by recompressing the off-diagonal blocks to reveal their true rank. Once this modification is incorporated, we obtain the inversion algorithm shown in Figure 3.3. Observe that in executing this algorithm, all matrix operations have to exploit that $\mathbf{A}_{11}$, $\mathbf{A}_{22}$, $\mathbf{X}_{11}$, and $\mathbf{X}_{22}$ are all HODLR matrices.

Given a HODLR matrix $\mathbf{A}$, compute its inverse $\mathbf{C}$ in HODLR matrix format.

(1) **function** $\mathbf{C}$ = `invert_matrix`($\mathbf{A}$)
(2) **if** ($\dim(\mathbf{A}) < 2k$) **then**
(3) Invert by brute force: $\mathbf{C} = \mathbf{A}^{-1}$.
(4) **else**
(5) Split $\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$.
(6) Let $\mathbf{A}_{12} = \mathbf{U}_1 \tilde{\mathbf{A}}_{12} \mathbf{V}_2^*$ and $\mathbf{A}_{21} = \mathbf{U}_2 \tilde{\mathbf{A}}_{21} \mathbf{V}_1^*$ denote low-rank factorizations of $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$.
(7) $\mathbf{X}_{11}$ = `invert_matrix`($\mathbf{A}_{11}$).
(8) $\mathbf{X}_{22}$ = `invert_matrix`($\mathbf{A}_{22}$).
(9) Compute $\mathbf{Y} = \begin{bmatrix} \mathbf{V}_1^* \mathbf{X}_{11} \mathbf{U}_1 & \tilde{\mathbf{A}}_{12} \\ \tilde{\mathbf{A}}_{21} & \mathbf{V}_2^* \mathbf{X}_{22} \mathbf{U}_2 \end{bmatrix}^{-1}$.
(10) $\mathbf{C} = \begin{bmatrix} \mathbf{X}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{X}_{22} \end{bmatrix} - \begin{bmatrix} \mathbf{X}_{11}\mathbf{U}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{X}_{22}\mathbf{U}_2 \end{bmatrix} \mathbf{Y} \begin{bmatrix} \mathbf{V}_1^*\mathbf{X}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^*\mathbf{X}_{22} \end{bmatrix}$.
(11) Recompress $\mathbf{C}$ to combat potential increase in ranks of off-diagonal blocks on line (10).
(11) **end if**
(12) **end function**

FIGURE 3.3. Inversion of a HODLR matrix. Observe that there is no general guarantee that the inverse of a HODLR matrix of rank k is another HODLR matrix of rank k. In general, the ranks of the off-diagonal blocks grow during the computation. The algorithm give involves a recompression step, but this of course only works when the underlying matrix truly has low numerical rank.

REMARK 3.2. *The technique for inverting a 2×2 block matrix via the Woodbury formula given in Section 3.2 is perhaps less commonly used than the following formula:*

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{X}_{11} & -\mathbf{X}_{11}\,\mathbf{A}_{12}\,\mathbf{A}_{22}^{-1} \\ -\mathbf{A}_{22}^{-1}\,\mathbf{A}_{21}\,\mathbf{X}_{11} & \mathbf{A}_{22}^{-1} + \mathbf{A}_{22}^{-1}\,\mathbf{A}_{21}\,\mathbf{X}_{11}\,\mathbf{A}_{12}\,\mathbf{A}_{22}^{-1} \end{bmatrix}, \tag{3.5}$$

where

$$\mathbf{X}_{11} = \left(\mathbf{A}_{11} - \mathbf{A}_{12}\,\mathbf{A}_{22}^{-1}\,\mathbf{A}_{21}\right)^{-1}.$$

Formula 3.5 is simpler, and can be used directly to implement a recursive algorithm for inverting a HODLR matrix since it too reduces the task of inverting a matrix of size $2n \times 2n$ to the task of inverting two matrices, each of size $n \times n$. However, when formula (3.5) is used, the two matrix inversions have to be executed consecutively — first we compute $\mathbf{A}_{22}^{-1}$, then we compute the inverse of $\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21}$. This makes the algorithm far more difficult to implement efficiently in parallel.

3.5. LU factorization and matrix-matrix multiplication

Section 3.4 describes a technique for rapidly inverting a HODLR matrix which is provably fast whenever all intermediate matrices can be recompressed into HODLR matrices of the same rank k as the original matrix. Let us next describe a similar technique for computing an LU factorization of a HODLR matrix. In other words, we seek a factorization

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ \mathbf{0} & \mathbf{U}_{22} \end{bmatrix},$$

where $\mathbf{L}_{11}$ and $\mathbf{L}_{22}$ are lower triangular, and $\mathbf{U}_{11}$ and $\mathbf{U}_{22}$ are upper triangular. The computation can be done via the following steps:

- $[\mathbf{L}_{11}, \mathbf{U}_{11}] =$ `lu_structured`($\mathbf{A}_{11}$)

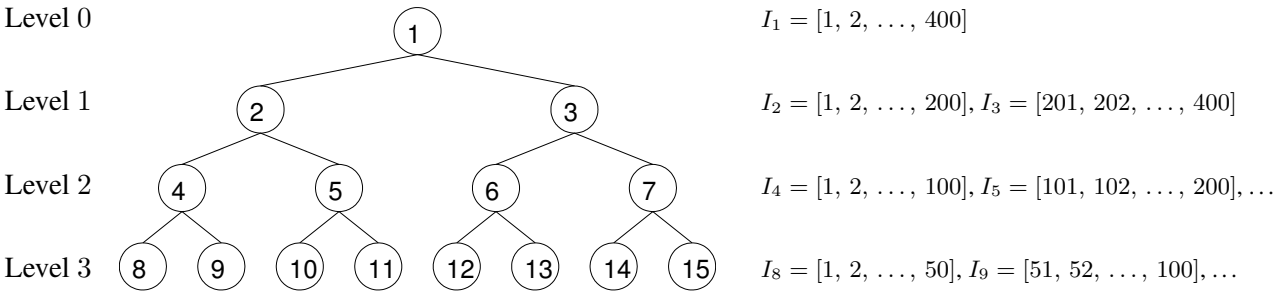

FIGURE 3.4. Numbering of nodes in a fully populated binary tree with $L = 3$ levels. The root is the original index vector $I = I_1 = [1, 2, \ldots, 400]$.

- $\mathbf{L}_{21} = \mathbf{A}_{21}\mathbf{U}_{11}^{-1}$ (execute via triangular solve on one of the factors of $\mathbf{A}_{21}$)
- $\mathbf{U}_{12} = \mathbf{L}_{11}^{-1}\mathbf{A}_{12}$ (execute via triangular solve on one of the factors of $\mathbf{A}_{12}$)
- $[\mathbf{L}_{22}, \mathbf{U}_{22}] = \texttt{lu\_structured}(\mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12})$

Note that since $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ are of rank k, the matrices $\mathbf{L}_{21}$ and $\mathbf{U}_{12}$ will automatically be of rank k as well. Like the matrix inversion formula (3.5) this procedure is inherently sequential, *first* $\mathbf{A}_{11}$ is factored, *then* the Schur complement $\mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12}$ is factored. As in all algorithms described in this section, recompression down to rank k is essential after forming the sum $\mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12}$. Chapter ? describes a variation of block LU that is not sequential in this way, and does not require recompression.

Next, let us consider the process of multiplying two compressible matrices $\mathbf{A}$ and $\mathbf{B}$ to compute their product $\mathbf{C} = \mathbf{AB}$. We again block the computation, to find the relations

$$\begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \left[\begin{array}{c|c} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \hline \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{array}\right].$$

We see that all terms in the product involves at least one factor that is a low-rank matrix, except for the two terms $\mathbf{A}_{11}\mathbf{B}_{11}$ and $\mathbf{A}_{22}\mathbf{B}_{22}$. Again the pattern repeats itself that the task of performing one operation on a matrix of size $N \times N$ gets reduced to the task of performing two operations on matrices of size $N/2 \times N/2$. However, re-compression is again required, and it cannot in general be guaranteed that ranks will not grow.

3.6. Hierarchical partitions of the index vector and a non-recursive definition of a HODLR matrix

In Section 3.3, we saw a *recursive* definition of a HODLR matrix. This formulation is conceptually simple, and leads to clean formulas. However, for practical efficiency, and for ease of analysis, it is often worthwhile to make the effort to introduce a notational framework that allows us to explicitly keep track of all submatrices at different levels.

Given an $N \times N$ matrix $\mathbf{A}$, the first step is to introduce a hierarchical partition of the index vector $I = [1, 2, \ldots, N]$. For simplicity, we limit attention to binary tree structures in which every level is fully populated. We let I form the root of the tree, and give it the index 1, $I_1 = I$. We next split the root into two roughly equi-sized vectors I_2 and I_3 so that $I_1 = I_2 \cup I_3$. The full tree is then formed by continuing to subdivide any interval that holds more than some preset fixed number n of indices. We use the integers $\ell = 0, 1, \ldots, L$ to label the different levels, with 0 denoting the coarsest level. A *leaf* is a node corresponding to a vector that never got split. For a non-leaf node τ, its *children* are the two boxes α and β such that $I_\tau = I_\alpha \cup I_\beta$, and τ is then the *parent* of α and β. Two boxes with the same parent are called *siblings*. These definitions are illustrated in Figure 3.4

Given the hierarchical partitioning of an index vector, we now say that $\mathbf{A}$ is a HODLR matrix of rank k, if for every sibling pair $\{\sigma, \tau\}$ of nodes in the tree, the corresponding off-diagonal block of $\mathbf{A}$ admits a

low-rank factorization

$$\begin{array}{cccccc} \mathbf{A}(I_\tau, I_\sigma) & = & \mathbf{U}_\tau & \tilde{\mathbf{A}}_{\tau,\sigma} & \mathbf{V}_\sigma^*, \\ N_\tau \times N_\sigma & & N_\tau \times k & k \times k & k \times N_\sigma \end{array} \tag{3.6}$$

where N_τ and N_σ denote the number of elements in I_τ and I_σ, respectively.

Let us calculate how much memory is required to store the various factors representing the matrix. We know that each leaf holds $O(k)$ nodes. Let b denote the scaling constant, so that each leaf holds roughly bk nodes. Then there are N/bk leaves, and the storage M_{diag} required for the diagonal blocks works out to

$$M_{\text{diag}} \approx \frac{N}{bk} (bk)^2 \sim bNk.$$

Let M_1 denote the amount of memory required to store the off-diagonal blocks on level 1, namely $\mathbf{A}_{2,3}$ and $\mathbf{A}_{3,2}$. These are matrices of size $N/2 \times N/2$ of rank k, so the amount of storage required is

$$M_1 \approx 2 \times \frac{N}{2} \times k = Nk$$

(We ignore terms of order k^2 or less, so it does not make a difference whether the SVD, ID, QR, etc, is used.) At the next finer level (level 2), there are four blocks, each of size $N/4 \times N/4$, so the amount of storage required is

$$M_2 \approx 4 \times \frac{N}{4} \times k = Nk$$

More generally, we see that at level ℓ, we need $2^\ell \times (N\, 2^{-\ell}) \times k$ storage, so the total storage required is

$$M_{\text{offdiag}} = \sum_{\ell=1}^{L} 2^\ell \times (N\, 2^{-\ell}) \times k = \sum_{\ell=1}^{L} Nk = LNk,$$

where L denotes the number of levels (the "depth" of the tree). Finally, observe that since there are bk nodes in each leaf box, and 2^L leaf boxes, the number L satisfies

$$N \approx 2^L\, bk,$$

which is to say that $L \sim \log_2(N/bk)$. We may pick b to be any fixed small number (say $b = 2$ or $b = 4$ or something along those line), which means that the total amount M of memory required to store a HODLR matrix satisfies

$$M = M_{\text{diag}} + M_{\text{offdiag}} \sim Nk + Nk\, \log(N/k) \sim Nk\, \log(N/k).$$

3.7. Non-recursive formulas for HODLR matrix operations

With the definition provided in Section 3.6, the matrix-vector multiplication can easily be expressed as a loop, as shown in Figure 3.5. Observe in particular that we are now at liberty to process the various nodes in the tree in any order which greatly simplifies implementation on distributed memory architectures.

Since each entry in the compressed version of the matrix is touched precisely once in the algorithm, the analysis in Section 3.6 that tells us that a HODLR matrix requires $O(kN \log(N/k))$ storage also says that the complexity of the HODLR matrix-vector multiplication is $O(kN \log(N/k))$.

A non-recursive algorithm for inversion of a HODLR matrix is given in Figure 3.6. We observe that once the recursions have been "rolled out" to obtain an explicit loop, the resulting technique involves a pass through the tree that goes from smaller boxes to larger.

Given a vector $\mathbf{q}$ and an S-matrix $\mathbf{A}$, form $\mathbf{u} = \mathbf{A}\mathbf{q}$.

(1) **for** (τ is a leaf)
(2) $\quad \mathbf{u}(I_\tau) = \mathbf{A}(I_\tau, I_\tau)\, \mathbf{q}(I_\tau)$.
(3) **end for**

(4) **for** (τ is a parent)
(5) $\quad$ Let $\{\alpha, \beta\}$ denote the children of τ.
(6) $\quad \mathbf{u}(I_\alpha) = \mathbf{u}(I_\alpha) + \mathbf{U}_\alpha \left(\tilde{\mathbf{A}}_{\alpha,\beta} \left(\mathbf{V}_\beta^*\, \mathbf{q}(I_\beta)\right)\right)$.
(7) $\quad \mathbf{u}(I_\beta) = \mathbf{u}(I_\beta) + \mathbf{U}_\beta \left(\tilde{\mathbf{A}}_{\beta,\alpha} \left(\mathbf{V}_\alpha^*\, \mathbf{q}(I_\alpha)\right)\right)$.
(8) **end for**

FIGURE 3.5. A non-recursive algorithm for applying a HODLR matrix to a vector. Observe that the loops can be transversed in any order. If the leaf boxes are not processed first, one needs to initialize by setting $\mathbf{u} = \mathbf{0}$ at first.

Given a HODLR matrix $\mathbf{A}$ and a computational tolerance, compute its inverse $\mathbf{C}$ in the HODLR format.

(1) **for** $\tau = N_{\text{boxes}} : (-1) : 1$
(2) $\quad$ **if** (τ is a leaf) **then**
(3) $\quad\quad$ Invert by brute force: $\mathbf{C}_\tau = \big(\mathbf{A}(I_\tau, I_\tau)\big)^{-1}$.
(4) $\quad$ **else**
(5) $\quad\quad$ Let $\{\alpha, \beta\}$ denote the children of τ.
(6) $\quad\quad$ Compute $\mathbf{Y} = \begin{bmatrix} \mathbf{V}_\alpha^*\mathbf{C}_\alpha\mathbf{U}_\alpha & \tilde{\mathbf{A}}_{\alpha\beta} \\ \tilde{\mathbf{A}}_{\beta\alpha} & \mathbf{V}_\beta^*\mathbf{C}_\beta\mathbf{U}_\beta \end{bmatrix}^{-1}$.
(7) $\quad\quad$ Invert by brute force: $\mathbf{C}_\tau = \begin{bmatrix} \mathbf{C}_\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_\beta \end{bmatrix} - \begin{bmatrix} \mathbf{C}_\alpha\mathbf{U}_\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_\beta\mathbf{U}_\beta \end{bmatrix} \mathbf{Y} \begin{bmatrix} \mathbf{V}_\alpha^*\mathbf{C}_\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_\beta^*\mathbf{C}_\beta \end{bmatrix}$.
(11) $\quad\quad$ Recompress $\mathbf{C}_\tau$ to combat potential increase in ranks of off-diagonal blocks on line (10).
(11) $\quad$ **end if**
(12) **end for**

FIGURE 3.6. A non-recursive algorithm for inversion of a HODLR matrix. The method assumes that the nodes in the tree are ordered so that if τ is the parent of τ, then $\tau < \sigma$. Observe that line (7) involves first applying the HODLR matrices $\mathbf{C}_\alpha$ and $\mathbf{C}_\beta$ to the columns of the basis matrices $\mathbf{U}_\alpha, \mathbf{V}_\alpha, \mathbf{U}_\beta, \mathbf{V}_\beta$, and then performing a low-rank update to the HODLR matrix $\texttt{diag}(\mathbf{C}_\alpha, \mathbf{C}_\beta)$.

3.8. Extensions

The purpose of the discussion in this chapter is to introduce some key ideas underpinning the machinery for manipulating rank-structured matrices efficiently. We deliberately restricted attention to the simplest possible case. Let us briefly describe some complications that may arise when rank-structured matrices are used in practice.

First let us observe that we have entirely avoided the topic of how to obtain the compressed representation of the matrix in the first place. This step is often a key challenge in a practical application, and involves first finding a good ordering of the columns and rows, and then finding the factors in (3.6) for each off-diagonal block.

One complication is that practical applications often involve non balanced trees where not every level in the is fully populated. A typical example could involve a matrix representing a discretized integral or

differential operator that corresponds to a physical mesh that involves local refinement. These generalizations are generally speaking not all that complicated from the point of view of building algorithms, but can make attaining high practical efficiency in software challenging. It also makes complexity analysis substantially more intricate.

While the $O(N \log N)$ typical storage complexity of the HODLR format is a substantial improvement over the $O(N^2)$ complexity of a dense unstructured matrix, it is in many applications possible to eliminate the $\log N$ factor and get all the way down to $O(N)$. The key is to avoid having to store the matrices $\mathbf{U}_\tau$ and $\mathbf{V}_\tau$ in (3.6) explicitly, as these matrices get large for large boxes in the tree. This can be achieved by representing each such basis matrix recursively and recycle information that is already stored in representing the basis matrices for the children of the box. We will explore these ideas in depth in chapters ?? and ??.

Finally, the HODLR format that we describe in this chapter relies on a very aggressive assumption in terms of which off-diagonal blocks have low numerical rank. Most matrices that arise in practical applications must be tesselated further into smaller pieces, and only some of these will admit low-rank approximations. A look ahead at Figure ? shows a tessellation pattern that is perhaps more representative. By cutting up the matrix into more pieces, and representing many of them using an uncompressed dense representation, one can often dramatically decrease the numerical ranks, but at the cost of making all algorithms for matrix arithmetic far more complex. We discuss these questions further in chapter ?? and ??.

3.9. References

The survey [**3**] is an excellent starting point for a reader wanting to learn more about the topic of rank-structured matrices. A foundational work of Hackbusch on $\mathcal{H}$-matrices was published in 1999 in [**18**]. Comprehensive descriptions and analysis of the $\mathcal{H}$ and $\mathcal{H}^2$ matrix formats can be found in [**4**] and [**6**].

The *Hierarchically Block Separable (HBS)* is described in the survey [**13**], which draws on earlier work [**31, 35, 33, 16**]. These solvers are algebraically essentially identical to the *recursive skeletonization* technique of [**22**]. The closely related *Hierarchically Semi Separable (HSS)* matrices are described in [**44, 42, 43**], while a different treatment of *Hierarchically Off-Diagonal Low Rank* matrices can be found in [**2**].

Bibliography

[1] Nir Ailon and Bernard Chazelle, *Approximate nearest neighbors and the fast johnson-lindenstrauss transform*, Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, ACM, 2006, pp. 557–563.

[2] Sivaram Ambikasaran and Eric Darve, *An $o(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices*, Journal of Scientific Computing **57** (2013), no. 3, 477–501 (English).

[3] Jonas Ballani and Daniel Kressner, *Matrices with hierarchical low-rank structures*, pp. 161–209, Springer International Publishing, Cham, 2016.

[4] Mario Bebendorf, *Hierarchical matrices*, Lecture Notes in Computational Science and Engineering, vol. 63, Springer-Verlag, Berlin, 2008, A means to efficiently solve elliptic boundary value problems. MR 2451321 (2009k:15001)

[5] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al., *An updated set of basic linear algebra subprograms (blas)*, ACM Transactions on Mathematical Software **28** (2002), no. 2, 135–151.

[6] Steffen Börm, *Efficient numerical methods for non-local operators*, EMS Tracts in Mathematics, vol. 14, European Mathematical Society (EMS), Zürich, 2010, $\mathcal{H}^2$-matrix compression, algorithms and analysis. MR 2767920

[7] H. Cheng, Z. Gimbutas, P.G. Martinsson, and V. Rokhlin, *On the compression of low rank matrices*, SIAM Journal of Scientific Computing **26** (2005), no. 4, 1389–1404.

[8] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software (TOMS) **16** (1990), no. 1, 1–17.

[9] P. Drineas and M.W. Mahoney, *Lectures on randomized linear algebra*, The Mathematics of Data (M.W. Mahoney, J.C. Duchi, and A.C. Gilbert, eds.), vol. 25, American Mathematical Society, 2018, IAS/Park City Mathematics Series, pp. 1 – 48.

[10] Petros Drineas, Ravi Kannan, and Michael W. Mahoney, *Fast Monte Carlo algorithms for matrices. II. Computing a low-rank approximation to a matrix*, SIAM J. Comput. **36** (2006), no. 1, 158–183 (electronic). MR MR2231644 (2008a:68243)

[11] Carl Eckart and Gale Young, *The approximation of one matrix by another of lower rank*, Psychometrika **1** (1936), no. 3, 211–218.

[12] A. Frieze, R. Kannan, and S. Vempala, *Fast Monte Carlo algorithms for finding low-rank approximations*, J. ACM **51** (2004), no. 6, 1025–1041, (electronic). MR MR2145262 (2005m:65006)

[13] Adrianna Gillman, Patrick Young, and Per-Gunnar Martinsson, *A direct solver $o(n)$ complexity for integral equations on one-dimensional domains*, Frontiers of Mathematics in China **7** (2012), 217–247, 10.1007/s11464-012-0188-3.

[14] Gene H. Golub and Charles F. Van Loan, *Matrix computations*, third ed., Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, Baltimore, MD, 1996.

[15] Lars Grasedyck and Wolfgang Hackbusch, *Construction and arithmetics of $\mathcal{H}$-matrices*, Computing **70** (2003), no. 4, 295–334.

[16] Leslie Greengard, Denis Gueyffier, Per-Gunnar Martinsson, and Vladimir Rokhlin, *Fast direct solvers for integral equations in complex three-dimensional domains*, Acta Numer. **18** (2009), 243–275.

[17] Ming Gu and Stanley C. Eisenstat, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM J. Sci. Comput. **17** (1996), no. 4, 848–869. MR 97h:65053

[18] Wolfgang Hackbusch, *A sparse matrix arithmetic based on H-matrices; Part I: Introduction to H-matrices*, Computing **62** (1999), 89–108.

[19] N. Halko, *Randomized methods for computing low-rank approximations of matrices*, Ph.D. thesis, Applied Mathematics, University of Colorado at Boulder, 2012.

[20] Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, and Mark Tygert, *An algorithm for the principal component analysis of large data sets*, SIAM Journal on Scientific Computing **33** (2011), no. 5, 2580–2594.

[21] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review **53** (2011), no. 2, 217–288.

[22] K.L. Ho and L. Greengard, *A fast direct solver for structured linear systems by recursive skeletonization*, SIAM Journal on Scientific Computing **34** (2012), no. 5, 2507–2532.

[23] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert, *Randomized algorithms for the low-rank approximation of matrices*, Proc. Natl. Acad. Sci. USA **104** (2007), no. 51, 20167–20172.

[24] Michael W Mahoney, *Randomized algorithms for matrices and data*, Foundations and Trends® in Machine Learning **3** (2011), no. 2, 123–224.

[25] Per-Gunnar Martinsson, *Randomized methods for matrix computations and analysis of high dimensional data*, arXiv preprint arXiv:1607.01649 (2016).
[26] Per-Gunnar Martinsson, *Randomized methods for matrix computations*, The Mathematics of Data (M.W. Mahoney, J.C. Duchi, and A.C. Gilbert, eds.), vol. 25, American Mathematical Society, 2018, IAS/Park City Mathematics Series, pp. 187 – 231.
[27] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert, *A randomized algorithm for the approximation of matrices*, Tech. Report Yale CS research report YALEU/DCS/RR-1361, Yale University, Computer Science Department, 2006.
[28] ———, *A randomized algorithm for the decomposition of matrices*, Appl. Comput. Harmon. Anal. **30** (2011), no. 1, 47–68. MR 2737933 (2011i:65066)
[29] P.G. Martinsson, G. Quintana Orti, and N. Heavner, *randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization*, arXiv preprint arXiv:1703.00998 (2017).
[30] P.G. Martinsson, G. Quintana-Ortí, N. Heavner, and R. van de Geijn, *Householder qr factorization with randomization for column pivoting (hqrrp)*, SIAM Journal on Scientific Computing **39** (2017), no. 2, C96–C115.
[31] P.G. Martinsson and V. Rokhlin, *A fast direct solver for boundary integral equations in two dimensions*, J. Comp. Phys. **205** (2005), no. 1, 1–23.
[32] P.G. Martinsson and S. Voronin, *A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices*, SIAM Journal on Scientific Computing **38** (2016), no. 5, S485–S507.
[33] E. Michielssen, A. Boag, and W. C. Chew, *Scattering from elongated objects: direct solution in $O(N \log^2 N)$ operations*, IEE Proc. Microw. Antennas Propag. **143** (1996), no. 4, 277 – 283.
[34] Tamas Sarlos, *Improved approximation algorithms for large matrices via random projections*, Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on, IEEE, 2006, pp. 143–152.
[35] Page Starr and Vladimir Rokhlin, *On the numerical solution of two-point boundary value problems. II*, Comm. Pure Appl. Math. **47** (1994), no. 8, 1117–1159.
[36] G. W. Stewart, *On the early history of the singular value decomposition*, SIAM Rev. **35** (1993), no. 4, 551–566. MR 1247916 (94f:15001)
[37] G. Strang, *Linear algebra and learning from data*, Welleslley-Cambridge Press, 2019.
[38] Lloyd N Trefethen and David Bau III, *Numerical linear algebra*, vol. 50, Siam, 1997.
[39] S. Voronin and P.G. Martinsson, *Efficient algorithms for cur and interpolative matrix decompositions*, Advances in Computational Mathematics (2016), 1–22.
[40] David P. Woodruff, *Sketching as a tool for numerical linear algebra*, Foundations and Trends in Theoretical Computer Science **10** (2014), no. 12, 1–157.
[41] Franco Woolfe, Edo Liberty, Vladimir Rokhlin, and Mark Tygert, *A fast randomized algorithm for the approximation of matrices*, Applied and Computational Harmonic Analysis **25** (2008), no. 3, 335–366.
[42] J. Xia, *Randomized sparse direct solvers*, SIAM Journal on Matrix Analysis and Applications **34** (2013), no. 1, 197–227.
[43] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numerical Linear Algebra with Applications **17** (2010), no. 6, 953–976.
[44] ———, *Superfast multifrontal method for large structured linear systems of equations*, SIAM J. Matrix Anal. Appl. **31** (2010), no. 3, 1382–1411. MR 2587783 (2011c:65072)