Matrix factorizations and low rank approximation

The first section of the course provides a quick review of basic concepts from linear algebra that we will use frequently. Note that the pace is fast here, and assumes that you have seen these concepts in prior course-work. If not, then additional reading on the side is strongly recommended!

1. Notation, etc

1.1. **Norms.** Let $\mathbf{x} = [x_1,\, x_2,\, \dots,\, x_n]$ 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

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

Observe that

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

1.2. **Transpose and adjoint.** Given an $m \times n$ matrix $\mathbf{A}$, the *transpose* $\mathbf{A}^{\rm t}$ 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

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

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

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

1.4. **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 $\mathbf{A}^{\rm t} = \mathbf{A}$.
- 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 $\mathbf{A}^* = \mathbf{A}^{-1}$.

2. Low rank approximation

2.1. **Exact rank deficiency.** Let 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 A span a subspace of $\mathbb{R}^m$ of dimension k.
- The rows of A span a subspace of $\mathbb{R}^n$ of dimension k.
- The nullspace of A has dimension $n - k$.
- The nullspace of A^* has dimension $m - k$.

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

$$\begin{array}{ccccc} \mathsf{A} & = & \mathsf{E} & \mathsf{F}. \\ m \times n & & m \times k & k \times n \end{array}$$

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

- Storing A requires mn words of storage.
  Storing E and F requires $km + kn$ words of storage.
- Given a vector x, computing Ax requires mn flops.
  Given a vector x, computing $\mathsf{Ax} = \mathsf{E}(\mathsf{Fx})$ requires $km + kn$ flops.
- The factors E and 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 E are orthonormal, and F is upper triangular (up to permutations of the columns).

2.2. **Approximate rank deficiency.** The condition that 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 2.1. Let A be an $m \times n$ matrix, and let ε be a positive real number. We then define the *ε-rank of* A as the unique integer k such that both the following two conditions hold:

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

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

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

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

3. The eigenvalue decomposition

Let A be an $n \times n$ matrix (it must be *square* for eigenvalues and eigenvectors to exist). We then say that λ is an eigenvalue and v is an eigenvector of A if $\mathsf{v} \neq \mathsf{0}$ and

$$\mathsf{Av} = \lambda \mathsf{v}.$$

Theorem 1. *Let A be an $n \times n$ matrix. Then A is normal (meaning that $\mathsf{AA}^* = \mathsf{A}^*\mathsf{A}$) if and only if A admits a factorization of the form*

$$\mathsf{A} = \mathsf{VDV}^* \qquad (1)$$

where V is unitary and D is diagonal.

The equation (1) can alternatively be written

$$\mathbf{A} = \sum_{j=1}^{n} \lambda_j \, \mathbf{v}_j \, \mathbf{v}_j^*,$$

where $\{\lambda_j,\, \mathbf{v}_j\}$ are the *eigenpairs* of $\mathbf{A}$, and

$$\mathbf{V} = \big[\mathbf{v}_1 \, \mathbf{v}_2 \, \cdots \, \mathbf{v}_n\big],$$

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}.$$

In other words, the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{A}$. These eigenvectors form an orthonormal basis for $\mathbb{C}^n$. In the basis $\{\mathbf{v}_j\}_{j=1}^n$, the matrix $\mathbf{A}$ is diagonal.

Recall that *self-adjoint*, *skew-adjoint*, and *unitary* matrices are special cases of *normal* matrices, so these classes all allow spectral decompositions. It is easy to verify that:

$\mathbf{A}$ is self-adjoint	$\Leftrightarrow$	$\mathbf{A}^* = \mathbf{A}$	$\Leftrightarrow$	Every eigenvalue is real.
$\mathbf{A}$ is skew-adjoint	$\Leftrightarrow$	$\mathbf{A}^* = -\mathbf{A}$	$\Leftrightarrow$	Every eigenvalue is imaginary.
$\mathbf{A}$ is unitary	$\Leftrightarrow$	$\mathbf{A}^* = \mathbf{A}^{-1}$	$\Leftrightarrow$	Every eigenvalue satisfies $\lvert\lambda_j\rvert = 1$.

Note that even a matrix whose entries are all real may have complex eigenvalues and eigenvectors.

What about non-normal matrices? Every square matrix has at least one (possibly complex) eigenvalue and one eigenvector. But if $\mathbf{A}$ is not normal, then there is no orthonormal basis consisting of eigenvectors. While eigenvalue decompositions of non-normal matrices are still very useful for certain applications, the lack of an ON-basis consisting of eigenvectors typically makes the *singular value decomposition* a much better tool for low-rank approximation in this case.

4. The Singular Value Decomposition

4.1. **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

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times p}{\mathbf{U}} \; \underset{p \times p}{\mathbf{D}} \; \underset{p \times n}{\mathbf{V}^*}, \tag{2}$$

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

$$\mathbf{U} = \big[\mathbf{u}_1 \, \mathbf{u}_2 \, \cdots \, \mathbf{u}_p\big],$$

$$\mathbf{V} = \big[\mathbf{v}_1 \, \mathbf{v}_2 \, \cdots \, \mathbf{v}_p\big],$$

$$\mathbf{D} = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \sigma_p \end{bmatrix}.$$

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

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

4.2. **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 (2). Then for an integer $k \in \{1, 2, \dots, p\}$, 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 $\mathbf{A}$:

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

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

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

Remark 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 ED could be complex even for a real matrix.)

4.3. **Connection between the SVD and the eigenvalue decomposition.** Let $\mathbf{A}$ be an $m \times n$ matrix. Then form the $m \times m$ matrix

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

Observe that $\mathbf{B}$ is self-adjoint, 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 $\mathbf{A}$.

4.4. **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 [3, 4], 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 $O(m\,n\,\min(m,n))$.
- While the asymptotic complexity of algorithms for computing full factorizations tend to be $O(m\,n\,\min(m,n))$ regardless of the choice of factorization (LU, QR, SVD, etc), the scaling constants are different. In particular pivoted QR is slower than non-pivoted QR, and the SVD is even slower.
- 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 (as of 2016) 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)      $i_j = \text{argmin}\{\|\mathbf{A}(:,\ell)\| :\ \ell = 1,2,\dots,n\}$
(4)      $\mathbf{q} = \mathbf{A}(:,i)/\|\mathbf{A}(:,i)\|$.
(5)      $\mathbf{r} = \mathbf{q}^*\mathbf{A}_{j-1}$
(6)      $\mathbf{Q}_j = [\mathbf{Q}_{j-1}\ \mathbf{q}]$
(7)      $\mathbf{R}_j = \begin{bmatrix} \mathbf{R}_{j-1} \\ \mathbf{r} \end{bmatrix}$
(8)      $\mathbf{A}_j = \mathbf{A}_{j-1} - \mathbf{q}\mathbf{r}$
(9)  end for
(10) $\mathbf{Q} = \mathbf{Q}_p$; $\mathbf{R} = \mathbf{R}_p$;
```

FIGURE 1. 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 $\mathbf{A} = \mathbf{QR}$. 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 $\mathbf{A}(:,[i_1,i_2,\dots,i_k])$.

5. THE QR FACTORIZATION

Let $\mathbf{A}$ be an $m \times n$ matrix. Set $p = \min(m,n)$. Let $\{\mathbf{a}_j\}_{j=1}^n$ denote the columns of $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix}\mathbf{a}_1\ \mathbf{a}_2\ \cdots\ \mathbf{a}_n\end{bmatrix}.$$

Our objective is now to find an orthonormal set of vectors $\{\mathbf{q}_j\}_{j=1}^p$ that form a "good" set of basis vectors for expressing the columns of $\mathbf{A}$. In other words, if we set

$$\mathbf{Q}_k = \begin{bmatrix}\mathbf{q}_1\ \mathbf{q}_2\ \cdots\ \mathbf{q}_k\end{bmatrix},$$

then we want

$$\|\mathbf{A} - \mathbf{Q}_k\mathbf{Q}_k^*\mathbf{A}\| \approx \inf\{\|\mathbf{A} - \mathbf{B}\| :\ \mathbf{B} \text{ has rank } k\} = \sigma_{k+1}.$$

We recall that the *optimal* basis (in this sense) is the left singular vectors of $\mathbf{A}$. However, these are tricky to compute, and we seek a simpler and faster algorithm that leads to *close to* optimal basis vectors.

5.1. The Gram-Schmidt process.

The Gram-Schmidt process is a simple "greedy" algorithm that can be coded efficiently. (Or at least reasonably efficiently, see Section 5.5.) Informally speaking, the idea is to take the collection of vectors $\{\mathbf{a}_j\}_{j=1}^n$, grab the largest one, normalize it to make its length one, and then use the resulting vector as the first basis vector. Then project the remaining $n-1$ vectors away from the one that was first chosen. Then take the largest vector of the remaining ones, normalize it to form the second basis vector, project the remaining $n-2$ vectors away from the new basis vector, etc. The resulting algorithm is shown in Figure 1.

Remark 2 (Break-down for rank-deficient matrices)**.** The process shown in Figure 1 will break down if the matrix has exact rank that is less than $\min(m,n)$. In this case, the residual matrix $\mathbf{A}$ will at some point be exactly zero. For purposes of formulating a mathematically correct algorithm, all one needs to do is to introduce a stopping criterion that breaks the loop if this happens. In practice, since all computations are carried out in finite precision, the residual is very unlikely to ever be *exactly* zero. However, when the residual gets small, round-off errors will create serious loss of accuracy. Techniques that overcome this problem are described in Section 5.2.

5.2. The QR factorization.

Starting with the simplistic process described in Section (5.1), we will make two sets of improvements:

(1) *Improving numerical accuracy:* The method described in Figure 1 works perfectly when executed in exact arithmetic. However, for large matrices, the basis vectors generated tend to lose orthonormality as the

computation proceeds. To avoid this, we perform an additional re-orthonormalization step. Specifically, after line (4), one should insert two new lines:

(4') $\mathbf{q} = \mathbf{q} - \mathbf{Q}_{j-1}\,(\mathbf{Q}_{j-1}^{*}\mathbf{q})$.

(4") $\mathbf{q} = \mathbf{q}/\|\mathbf{q}\|$.

These additional steps would make no difference in exact arithmetic, but they are important in practical computations. Without them, you often lose orthonormality in the basis vectors, which greatly degrades the utility of the computed factorization.

(2) *Better pivoting:* In practice, it is convenient to explicitly swap out the pivot vector you choose at step k to the k'th column in the matrix $\mathbf{Q}$ that you build. One must also do the analogous swap in the $\mathbf{R}$ matrix being built. (Moreover, it is possible to improve computational speed by accelerating the pivot selection step on line (3). The idea is to maintain a vector of length $1 \times n$ that holds the sum of the squares of all remaining residuals. This vector can be cheaply downdated at the end of the loop, which saves us the need to compute the norms of the columns of $\mathbf{Q}$. See [3, Sec. 5.4.1].)

(3) *Improved book-keeping:* The algorithm as written is wasteful of memory. One reasonably storage efficient way of implementing the method is to define at the outset a new matrix $\mathbf{Q}$ of size $m \times n$ and simply copying $\mathbf{A}$ over to $\mathbf{Q}$. This matrix $\mathbf{Q}$ is used to hold both the new basis vectors that are stored in $\mathbf{Q}_k$ in Figure 1 and the residual columns that are stored in $\mathbf{A}_k$ in Figure 1. Observe that in each step, we zero out one residual vector, and create one new basis vector, so there is a perfect match. To be precise, at the end of the k'th step, the matrix $\mathbf{Q}$ holds $\mathbf{Q} = \big[\mathbf{Q}_k\ \tilde{\mathbf{A}}_k\big]$, where $\mathbf{Q}_k$ holds the computed k columns of $\mathbf{Q}$, and $\tilde{\mathbf{A}}_k$ holds the remaining $n-k$ residual vectors.

The algorithm resulting from incorporating these improvements is given in Figure 2.

Remark 3 (Householder QR)**.** The QR factorization algorithm described in Figure 2 can be optimized further. In professional software packages, standard practice is to form the matrix $\mathbf{Q}$ as a product of so called *Householder reflectors.* These provide optimal stability and maintain very high orthonormality among the columns of $\mathbf{Q}$. Further, when Householder reflectors are used, all the information needed to form $\mathbf{R}$ and $\mathbf{Q}$ can be stored in a single array of size $m \times n$, as opposed to the formulation we give where two copies of the array are used. The trick is to use the zero elements formed "under the diagonal" in $\mathbf{R}$ to store enough information to uniquely define $\mathbf{Q}$. See [3, Sec. 5.4] or [4, Lecture 10].

5.3. **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. Consider the algorithm given in Figure 2. After k steps of the algorithm, we find that the matrices $\mathbf{Q}$ and $\mathbf{R}$ hold the following entries:

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_k \\ \mathbf{0} \end{bmatrix} \qquad \text{and} \qquad \mathbf{Q} = \big[\mathbf{Q}_k\ \tilde{\mathbf{A}}_k\big].$$

The matrix $\mathbf{R}_k$ is the matrix of size $k \times n$ holding the top k rows of $\mathbf{R}$, the matrix $\mathbf{Q}_k$ is of size $m \times k$ and holds the first k columns of $\mathbf{Q}$, and the matrix $\tilde{\mathbf{A}}_k$ is of size $m \times (n-k)$ and holds the "remainder" of the columns that have not yet been chosen as pivot columns (in other words, it contains the non-zero columns of the matrix $\mathbf{A}_k$ in Figure 1). Then the partial factorization we have computed after k steps reads

$$\mathbf{A}(:,J) = \mathbf{Q}_k\mathbf{R}_k + \big[\mathbf{0}\ \tilde{\mathbf{A}}_k\big]. \tag{3}$$

Let $\mathbf{P}_k$ denote the permutation matrix defined by the index vector J so that

$$\mathbf{A}(:,J) = \mathbf{A}\mathbf{P}_k.$$

Then we can rewrite (3) as (note that $\mathbf{P}_k^{-1} = \mathbf{P}_k^{*}$)

$$\mathbf{A} = \mathbf{Q}_k\mathbf{R}_k\mathbf{P}_k^{*} + \big[\mathbf{0}\ \tilde{\mathbf{A}}_k\big]\mathbf{P}_k^{*}.$$

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

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 $\|\tilde{\mathbf{A}}_k\|_{\mathrm{F}}$ and stop when this quantity drops below ε. The resulting algorithm is given in Figure 3.

```
       Create and initialize the output matrices.
(1)    $\mathbf{Q} = \mathbf{A}$; $\mathbf{R} = \mathtt{zeros}(\min(m,n), n)$; $J = 1 : n$;
(2)    for $j = 1 : \min(m,n)$
           Find the pivot column $i$.
(3)        $i = \text{argmin}\{\|\mathbf{Q}(:,\ell)\| : \ell = j, j+1, j+2, \dots, n\}$
           Move the chosen pivot column to the $j$'th slot.
(4)        $J([j,i]) = J([i,j])$; $\mathbf{Q}(:,[j,i]) = \mathbf{Q}(:,[i,j])$; $\mathbf{R}(:,[j,i]) = \mathbf{R}(:,[i,j])$;
(5)        $\rho = \|\mathbf{Q}(:,j)\|$
(6)        $\mathbf{R}(j,j) = \rho$
           Perform the "paranoid" reorthonormalization.
(7)        $\mathbf{q} = (1/\rho)\,\mathbf{Q}(:,j)$
(8)        $\mathbf{q} = \mathbf{q} - \mathbf{Q}(:,1:(j-1))\,(\mathbf{Q}(:,1:(j-1))^*\,\mathbf{q})$
(9)        $\mathbf{q} = (1/\|\mathbf{q}\|)\,\mathbf{q}$
(10)       $\mathbf{Q}(:,j) = \mathbf{q}$
           Compute the expansion coefficients and update Q and R.
(11)       $\mathbf{r} = \mathbf{q}^*\,\mathbf{Q}(:,(j+1):n)$
(12)       $\mathbf{R}(j,(j+1):n) = \mathbf{r}$
(13)       $\mathbf{Q}(:,(j+1):n) = \mathbf{Q}(:,(j+1):n) - \mathbf{q}\mathbf{r}$
(14)   end for
       If $n > m$, then we need to delete the last columns of Q.
(15)   $\mathbf{Q} = \mathbf{Q}(:,1:\min(m,n))$
```

FIGURE 2. QR factorization via Gram Schmidt. The algorithm takes as input an $m \times n$ matrix $\mathbf{A}$. The output is, with $p = \min(m,n)$, an index vector J, an $m \times p$ ON matrix $\mathbf{Q}$, and a $p \times n$ upper triangular matrix $\mathbf{R}$ such that $\mathbf{A}(:,J) = \mathbf{QR}$.

5.4. **Getting the SVD from the QR factorization.** The technique described in Section 5.3 results in a factorization of the form

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times k}{\mathbf{Q}}\ \underset{k\times n}{\mathbf{R}\mathbf{P}^*} + \underset{m\times n}{\mathbf{E}} \tag{4}$$

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

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

(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 $\mathbf{RP}^*$, which is cheap since $\mathbf{R}$ is small (it has only k rows)

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

(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

$$\mathbf{A} = \mathbf{Q}\underbrace{\mathbf{RP}^*}_{=\hat{\mathbf{U}}\mathbf{DV}^*} + \mathbf{E} = \underbrace{\mathbf{Q}\hat{\mathbf{U}}}_{=\mathbf{U}}\mathbf{DV}^* + \mathbf{E} = \mathbf{UDV}^* + \mathbf{E}. \tag{5}$$

We have obtained a partial SVD. Observe in particular that the error term $\mathbf{E}$ is exactly the same in both (4) and (5).

```
          Create and initialize the output matrices.
(*)       $\mathbf{Q} = \mathbf{A}$; $\mathbf{R} = \texttt{zeros}(\min(m,n),n)$; $J = 1:n$;
(*)       for $j = 1:\min(m,n)$
              Find the pivot column $i$.
(*)           $i = \text{argmin}\{\|\mathbf{Q}(:,\ell)\| : \ell = j, j+1, j+2, \dots, n\}$
(*)           $J([j,i]) = J([i,j])$; $\mathbf{Q}(:,[j,i]) = \mathbf{Q}(:,[i,j])$; $\mathbf{R}(:,[j,i]) = \mathbf{R}(:,[i,j])$;
              Move the chosen pivot column to the $j$'th slot.
(*)           $\rho = \|\mathbf{Q}(:,j)\|$
(*)           $\mathbf{R}(j,j) = \rho$
              Perform the "paranoid" reorthonormalization.
(*)           $\mathbf{q} = (1/\rho)\,\mathbf{Q}(:,j)$
(*)           $\mathbf{q} = \mathbf{q} - \mathbf{Q}(:,1:(j-1))\,(\mathbf{Q}(:,1:(j-1))^{*}\,\mathbf{q})$
(*)           $\mathbf{q} = (1/\|\mathbf{q}\|)\,\mathbf{q}$
(*)           $\mathbf{Q}(:,j) = \mathbf{q}$
              Compute the expansion coefficients and update $\mathbf{Q}$ and $\mathbf{R}$.
(*)           $\mathbf{r} = \mathbf{q}^{*}\,\mathbf{Q}(:,(j+1):n)$
(*)           $\mathbf{R}(j,(j+1):n) = \mathbf{r}$
(*)           $\mathbf{Q}(:,(j+1):n) = \mathbf{Q}(:,(j+1):n) - \mathbf{q}\mathbf{r}$
              Check the accuracy of the partial factorization.
              if $\texttt{sum}(\texttt{sum}(\mathbf{Q}(:,(j+1):n).*\mathbf{Q}(:,(j+1):n))) \leq \varepsilon^2$ then break
(*)       end for
(*)       $k = j$; $\mathbf{Q} = \mathbf{Q}(:,1:k)$; $\mathbf{R} = \mathbf{R}(1:k,:)$;
```

FIGURE 3. Partial QR factorization via Gram Schmidt. The algorithm takes as input an $m \times n$ matrix $\mathbf{A}$ and a tolerance ε. The output is, an index vector J, an $m \times k$ ON matrix $\mathbf{Q}$, and a $k \times n$ upper triangular matrix $\mathbf{R}$ such that $\|\mathbf{A}(:,J) - \mathbf{QR}\|_{\text{F}} \leq \varepsilon$. The integer k is the computed rank, and is an output parameter. (Observe that the computation of the Frobenius norm of the remainder matrix, and the determination of the pivot vectors can be optimized.)

5.5. **Blocking of algorithms and execution speed.** The various QR factorization algorithms described in this section are extremely 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

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

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) [2, 1].

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

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

On most computer architectures we have $C_{\rm QR} < C_{\rm CPQR} < C_{\rm SVD}$, 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 **??**.

6. Applications of low-rank approximation

This section briefly reviews some applications of low-rank approximation. Throughout the section, suppose that $\mathbf{A}$ is an $m \times n$ matrix of rank k. (Typically, this would be an *approximate* rank, but for simplicity, let us ignore the error for now.) Our starting point here is that we have computed rank k factorizations, either a QR factorization

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{Q} & \mathbf{R} & \mathbf{P}^* \\ m\times n & & m\times k & k\times n & n\times n \end{array} \tag{6}$$

or an SVD

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^*. \\ m\times n & & m\times k & k\times k & k\times n \end{array} \tag{7}$$

6.1. **The column space.** This is easy, the columns of $\mathbf{Q}$ and $\mathbf{U}$ directly form ON-bases for col$(\mathbf{A})$. The two bases are typically different.

6.2. **The row space.** If you have the SVD (7), then the columns of $\mathbf{V}$ form an ON-basis for row$(\mathbf{A})$. If you have the QR factorization (7), then the rows of $\mathbf{R}$ in principle form an ON-basis for the row space of $\mathbf{A}$, but it is not orthonormal. Observe however that since k is small, it is easy to obtain an ON-basis by simply performing Gram-Schmidt on the rows of $\mathbf{R}$ (pivoting is typically not required). For instance, if we execute

$$[\mathbf{S}, \sim] = \texttt{qr}(\mathbf{R}^*, 0)$$

then the columns of $\mathbf{S}$ form an ON basis for row$(\mathbf{A})$.

6.3. **The nullspace of a matrix.** The *partial* factorizations we computed do not directly provide a basis for the nullspace of a matrix. If the matrix is small, then you could compute the full factorizations, and get the bases that way. For big matrices, this tends to not be feasible, though. However, observe that the information provided is enough to *characterize* the null-space. Suppose that we have access to an $n \times k$ matrix $\mathbf{V}$ holding an ON basis for the row space of $\mathbf{A}$ (e.g. the matrix of right singular vectors). Observe that we can then split the identity operator as

$$\mathbf{I} = \mathbf{V}\mathbf{V}^* + \big(\mathbf{I} - \mathbf{V}\mathbf{V}^*\big).$$

The first term is the orthogonal projection onto row$(\mathbf{A})$, and the second term is the orthogonal projection onto row$(\mathbf{A})^\perp$ = null$(\mathbf{A})$. In other words, given a vector $\mathbf{x} \in \mathbb{R}^n$, we can write

$$\mathbf{x} = \underbrace{\mathbf{V}\mathbf{V}^*\mathbf{x}}_{\in \text{row}(\mathbf{A})} + \underbrace{\big(\mathbf{I} - \mathbf{V}\mathbf{V}^*\big)\mathbf{x}}_{\in \text{null}(\mathbf{A})}.$$

6.4. **Solving a (least squares) linear system.**

References

[1] 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(2):135–151, 2002.

[2] 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(1):1–17, 1990.

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

[4] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.