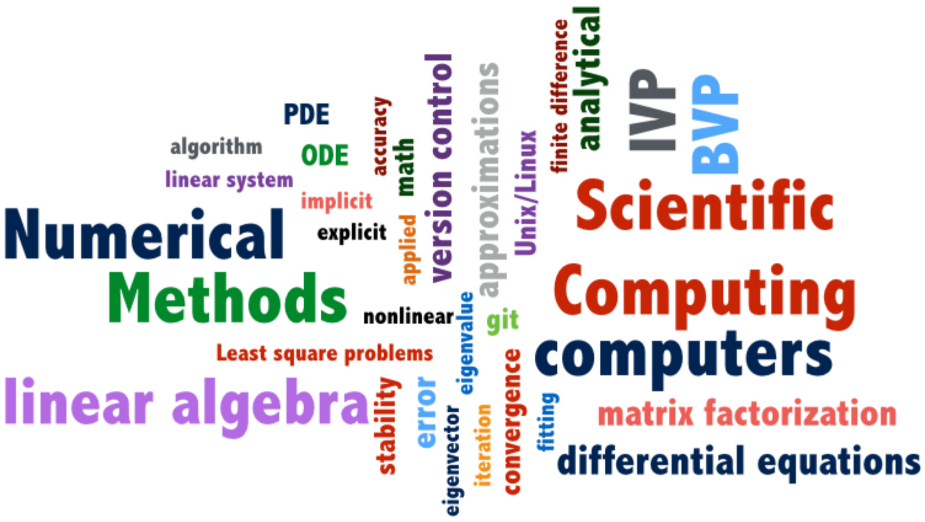
Lecture Note on AM 213A: Numerical Linear Algebra

Instructor: Dongwook Lee (dlee79@ucsc.edu)

Written by
Profs. Dongwook Lee, Pascale Garaud, and Mr. Skylar Trigueiro



Contents

Chapter 1

Introduction and basic tools

1. Course Description

This introductory course is designed to teach you fundamental topics in Numerical Linear Algebra. As described in the introduction to our textbook, numerical linear algebra is really a course in *applied* linear algebra, in which we learn (i) applied problems where techniques of linear algebra are used in real life (ii) and how to solve them numerically.

Over the course of this quarter, we will cover the following topics (not necessarily in this order):

- How to compute solutions of *well-posed* linear systems of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ using such as Gaussian elimination, LU factorization, and Cholesky factorization. We will also see a few examples of where such problems may arise in various applied mathematical research questions.

- How to compute approximate solutions of overdetermined linear system of equations $\mathbf{A}\mathbf{x} \approx \mathbf{b}$, such as one that may arise from linear least squares fitting problems. Primary topics include normal equations, orthogonal transformations, QR factorizations, and some popular orthogonalization methods such as Gram-Schmidt, and Householder transformations;

- How to find the eigenvector and eigenvalues in eigenvalue problems $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. Eigenvalue problems arise in a wide ranges of science and engineering fields and play a key role in the analysis of their stability for instance. Topics include numerical approaches that allow us to compute eigenvalues and eigenvectors (e.g., power iteration, Rayleigh quotient iteration, QR iteration).

- Singular value decomposition and its applications.

- Finally, how to solve linear systems of equations using iterative methods, which are usually more easily parallelizable than direct methods. This includes simple methods such as the Jacobi, Gauss-Seidel and Successive Over-relaxation methods, and more sophisticated Krylov methods including Conjugate Gradient, GMRES, Arnoldi, and Lanczos methods.

In addition to these topics, we will also spend some time learning about stability issues that arise specifically from the numerical treatment of linear algebra problems, because numerical arithmetic (specifically, floating point arithmetic) is not exact. Finally, we will also have a mind towards the use of some of these algorithms in high-performance computing, and will discuss, when appropriate, issues such as storage, computing efficiency, and parallelization. For further information on that last topic, see the AM 250 course.

In this first Chapter, we begin by reviewing some basics of linear algebra, and introduce some definitions that will be used throughout the course. Note that this is not meant to be a course on Linear Algebra, but rather, a very brief recap of material that you should already be familiar with and that is a prerequisite for the course (e.g. AM 10 for instance, or any undergraduate Linear Algebra course). We will then continue by looking at the issues associated with floating point arithmetic and their potential effect on the stability of matrix operations.

2. Matrices and Vectors

See Chapter 1 of the textbook

2.1. Matrix and vector multiplications, component notations, inner and outer products

2.1.1. Basic operations and component notations Given an $m \times n$ matrix $\mathbf{A}$, it has m rows and n columns. Its components are expressed as $(\mathbf{A})_{ij} = a_{ij}$ where $i = 1, \dots, m$ spans the rows, and $j = 1, \dots, n$ spans the columns. The coefficients a_{ij} can be real or complex.

We can compute the product of $\mathbf{A}$ with a vector $\mathbf{x}$ as $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ has n entries and $\mathbf{b}$ has m entries. Written in component form, this is

$$\mathbf{A}\mathbf{x} = \mathbf{b} \to b_i = \sum_{j=1}^{n} a_{ij} x_j \text{ for } i = 1, \dots, m \tag{1.1}$$

This can also be interpreted as

$$\mathbf{b} = \begin{pmatrix} \\ \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \\ \\ \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n \tag{1.2}$$

where the $\mathbf{a}_i$ are the column vectors of $\mathbf{A}$, showing that $\mathbf{A}\mathbf{x} = \mathbf{b}$ effectively expresses the vector $\mathbf{b}$ in the space spanned by the column vectors of $\mathbf{A}$.

Being a linear operation, multiplication of a vector by $\mathbf{A}$ has the following properties:

$$\mathbf{A}(\mathbf{x} + \mathbf{y}) = \mathbf{A}\mathbf{x} + \mathbf{A}\mathbf{y} \tag{1.3}$$

$$\mathbf{A}(\alpha\mathbf{x}) = \alpha\mathbf{A}\mathbf{x} \tag{1.4}$$

It is easy to show these properties using the component form (1.1) for instance.

We can also take the product of two suitably-sized matrices. Suppose $\mathbf{A}$ is an $m \times n$ matrix, and $\mathbf{B}$ is an $n \times l$ matrix, then their product $\mathbf{C} = \mathbf{AB}$ is a $m \times l$ matrix. Component-wise the product is expressed as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \text{ for } i = 1, \dots, m \text{ and } j = 1, \dots, l \tag{1.5}$$

Another way of interpreting the matrix multiplication is column-by-column multiplication:

$$\mathbf{C} = \mathbf{AB} = \mathbf{A} \begin{pmatrix} \\ \mathbf{b}_1 & \mathbf{b}_2 & \dots & \mathbf{b}_l \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{Ab}_1 & \mathbf{Ab}_2 & \dots & \mathbf{Ab}_l \\ \\ \end{pmatrix} \tag{1.6}$$

Recall that, in general, matrix multiplication does *not* commute so $\mathbf{AB} \neq \mathbf{BA}$.

2.1.2. Other basic matrix operations **The transpose** of a matrix $\mathbf{A}$ of size $m \times n$ is the $n \times m$ matrix $\mathbf{A}^T$ whose components are $(\mathbf{A}^T)_{ij} = a_{ji}$. Note the switch in indices, so the rows of $\mathbf{A}^T$ are the columns of $\mathbf{A}$, and vice versa.

For linear algebra with complex matrices, the notion of transpose must be replaced with that of the **adjoint**. **The adjoint** of a complex matrix $\mathbf{A}$ of size $m \times n$ is the $n \times m$ complex matrix $\mathbf{A}^*$, which is the *complex conjugate* of the transpose of $\mathbf{A}$, so that $(\mathbf{A}^*)_{ij} = a^*_{ji}$. Note that the following applies:

- The transpose and adjoint operations are nearly linear in the sense that $(\alpha\mathbf{A}+\beta\mathbf{B})^T = \alpha\mathbf{A}^T+\beta\mathbf{B}^T$ though one must be careful that $(\alpha\mathbf{A}+\beta\mathbf{B})^* = \alpha^*\mathbf{A}^* + \beta^*\mathbf{B}^*$.
- The transpose and adjoint of a product satisfy $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ and $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$

A symmetric matrix satisfies $\mathbf{A} = \mathbf{A}^T$. The complex equivalent is a **Hermitian matrix**, which satisfies $\mathbf{A} = \mathbf{A}^*$.

2.1.3. Inner and outer products. It is sometimes useful to think of vectors as matrices. By default, vectors are thought of as column-vectors, so that a column-vector with m components is an $m \times 1$ matrix. However, it is also possible to consider row-vectors, where an m-component row-vector is a $1 \times m$ matrix. It is easy to see that we can create a row-vector by taking the transpose (or adjoint, for complex vectors) of a column vector.

With these definitions, we can then define the inner and outer products of column- and row-vectors as:

- The **inner product** is the product of a $1 \times m$ (row) vector $\mathbf{x}^T$ with a $m \times 1$ (column) vector $\mathbf{y}$, and the result is a scalar (a 1×1 matrix):

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{m} x_{1i}y_{i1} = \sum_{i=1}^{m} x_i y_i \tag{1.7}$$

  For complex vectors, this becomes

$$\mathbf{x}^*\mathbf{y} = \sum_{i=1}^{m} x_{1i}^* y_{i1} = \sum_{i=1}^{m} x_i^* y_i \tag{1.8}$$

- The **outer product** is the product of a $m \times 1$ (column) vector $\mathbf{x}$ with a $1 \times m$ (row) vector $\mathbf{y}^T$, and the result is an $m \times m$ matrix whose components are

$$(\mathbf{x}\mathbf{y}^T)_{ij} = x_{i1}y_{1j} = x_i y_j \tag{1.9}$$

  Again, for complex vectors this becomes

$$(\mathbf{x}\mathbf{y}^*)_{ij} = x_{i1}y_{1j}^* = x_i y_j^* \tag{1.10}$$

2.1.4. Range, Nullspace and Rank The matrix operation $\mathbf{x} \to \mathbf{A}\mathbf{x}$ is a function from $\mathbb{C}^n$ to $\mathbb{C}^m$, which, as we saw, returns a vector $\mathbf{A}\mathbf{x}$ that is a linear combination of the columns of $\mathbf{A}$. However, depending on $\mathbf{A}$, the operation $\mathbf{A}\mathbf{x}$ may generate vectors that only span a subspace of $\mathbb{C}^m$, rather than the whole of $\mathbb{C}^m$. We therefore define **the range** of the matrix $\mathbf{A}$ as the range of the function $\mathbf{A}\mathbf{x}$, namely the subspace of $\mathbb{C}^m$ spanned by all possible vectors $\mathbf{A}\mathbf{x}$ when $\mathbf{x}$ varies in the whole of $\mathbb{C}^n$. Having seen that $\mathbf{A}\mathbf{x}$ is necessarily a linear combination of the columns of $\mathbf{A}$, we have the theorem:

Theorem: *The range of* $\mathbf{A}$ *is the subspace spanned by the columns of* $\mathbf{A}$.

The nullspace of $\mathbf{A}$ is the set of all vectors $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} = \mathbf{0}$. Its dimension is often written null($\mathbf{A}$) and called the **nullity**.

And finally **the column rank** of $\mathbf{A}$ is the dimension of the space spanned by its column-vectors, and **the row rank** of a matrix is the space spanned by its row vectors. Even if the matrix is not square, *an important property of matrices is that the row rank and column ranks are always equal* (we will demonstrate this later), so we usually refer to them simply as the rank of the matrix. For a matrix $\mathbf{A}$ of size $m \times n$, we have that

$$\text{rank}(\mathbf{A}) + \text{null}(\mathbf{A}) = n, \tag{1.11}$$

i.e. the sum of the rank and the nullity is equal to the number of columns of $\mathbf{A}$.

A **full rank** matrix is a matrix that has maximal possible rank: if the matrix is $m \times n$, then the rank is equal to $\min(m, n)$. This implies that the nullity of a full rank matrix is $n - \min(m, n)$, and is 0 if the matrix has $m \geq n$ (square or tall matrices) but greater than 0 if the matrix has $m < n$ (wide matrices). Full rank matrices with $m \geq n$ therefore have an important property:

Theorem: *Given an $m \times n$ matrix $\mathbf{A}$ with $m \geq n$, it has full rank if and only if no two distinct vectors $\mathbf{x}$ and $\mathbf{y}$ exist such that $\mathbf{Ax} = \mathbf{Ay}$.*

The same is not true for matrices with $m < n$, however.

2.2. The inverse of a matrix

The notion of inverse of a matrix only makes sense when considering square matrices, which is what we now assume in this section.

A full rank square matrix of size $m \times m$ is invertible, and we denote the inverse of the matrix $\mathbf{A}$ as $\mathbf{A}^{-1}$. The inverse $\mathbf{A}^{-1}$ is also of size $m \times m$, and by definition, satisfies the following property:

$$\mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{1.12}$$

for any vector $\mathbf{x}$ in $\mathbb{C}^m$. By analogy with earlier, we therefore see that this expresses the vector $\mathbf{x}$ in the space spanned by the columns of $\mathbf{A}^{-1}$. Now, suppose we construct the m vectors $\mathbf{z}_i$ such that

$$\mathbf{z}_i = \mathbf{A}^{-1}\mathbf{e}_i \tag{1.13}$$

where the vectors $\mathbf{e}_i$ are the unit vectors in $\mathbb{C}^m$. Then by definition, $\mathbf{Az}_i = \mathbf{e}_i$. This implies

$$\begin{aligned}\mathbf{AZ} \quad &= \mathbf{A}\begin{pmatrix}\mathbf{z}_1 & \mathbf{z}_2 & \dots & \mathbf{z}_m\end{pmatrix} = \begin{pmatrix}\mathbf{Az}_1 & \mathbf{Az}_2 & \dots & \mathbf{Az}_m\end{pmatrix} \\ &= \begin{pmatrix}\mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_m\end{pmatrix} = \mathbf{I}\end{aligned} \tag{1.14}$$

where $\mathbf{I}$ is the identity matrix. In short, this proves that $\mathbf{AA}^{-1} = \mathbf{I}$, and it can easily be proved that this also implies $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$

There are many equivalent conditions for a matrix to be invertible (also called non-singular). These are, for instance:

Theorem: For any given square matrix of size $m \times m$, the following statements are equivalent:

- $\mathbf{A}$ has an inverse, $\mathbf{A}^{-1}$.
- $\text{rank}(\mathbf{A}) = m$ (the matrix has full rank)
- The range of $\mathbf{A}$ is $\mathbb{R}^m$ (or $\mathbb{C}^m$, for complex matrices)
- $\text{null}(\mathbf{A}) = 0$ (the nullspace is empty)
- 0 is not an eigenvalue of $\mathbf{A}$ (see later for definition)
- 0 is not a singular value of $\mathbf{A}$ (see later for definition)
- $\det(\mathbf{A}) \neq 0$

An important property of the inverse is that $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$. In combination with the transpose or the adjoint, we also have that $(\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$, and similarly $(\mathbf{A}^*)^{-1} = (\mathbf{A}^{-1})^*$. For this reason, you may sometimes also find the notations $\mathbf{A}^{-T}$ or $\mathbf{A}^{-*}$ to denote the successive application of the inverse and transpose/adjoint – the order does not matter.

3. Orthogonal vectors and matrices

See Chapter 2 of the textbook

3.1. Orthogonality of two vectors

The inner product defined in the previous section, when applied to two real vectors, is in fact the well known dot product. It therefore has the following properties:

- **The Eucledian length** of a vector is defined as $||\mathbf{x}|| = \sqrt{\mathbf{x}^T\mathbf{x}}$ (or by extension, $||\mathbf{x}|| = \sqrt{\mathbf{x}^*\mathbf{x}}$ for complex vectors)
- The cosine of the angle between two vectors $\mathbf{x}$ and $\mathbf{y}$ is given by

$$\cos\alpha = \frac{\mathbf{x}^T\mathbf{y}}{||\mathbf{x}^T||||\mathbf{y}||} \text{ for real vectors or } \cos\alpha = \frac{\mathbf{x}^*\mathbf{y}}{||\mathbf{x}^*||||\mathbf{y}||} \text{ otherwise} \quad (1.15)$$

- Two non-zero vectors $\mathbf{x}$ and $\mathbf{y}$ are therefore **orthogonal** provided their inner product $\mathbf{x}^*\mathbf{y} = 0$. If the vectors are real, this means that they lie at right-angles to one another in $\mathbb{R}^m$.

From this definition, we can now define an **orthogonal set** of non-zero vectors, as a set in which every possible pair of vectors is orthogonal (i.e. all members of the set are pair-wise orthogonal). An **orthonormal set** is a orthogonal set where all vectors have unit length, so $||\mathbf{x}|| = 1$ for all $\mathbf{x}$.

Orthogonal sets have a very important and very useful property:

Theorem: *The vectors in an orthogonal set are linearly independent.*

Linear independence means that it is not possible to write any vector in the set as a linear combination of other vectors in the set.

As a corollary, it is sufficient in $\mathbb{C}^m$ to find m orthogonal vectors to have a basis for the whole space $\mathbb{C}^m$. Using the orthogonality property, it is then very easy to express any vector in $\mathbb{C}^m$ as a linear combination of this new orthogonal basis. Indeed, suppose we have the orthonormal basis $\{\mathbf{q}_k\}_{k=1,\dots,m}$, and we want to write the vector $\mathbf{b}$ in that basis. We know that it is possible, i.e. that there is a series of coefficients β_k such that

$$\mathbf{b} = \sum_{k=1}^{m} \beta_k \mathbf{q}_k \tag{1.16}$$

To find the β_k coefficient, we simply take the inner product of this expression with $\mathbf{q}_k$ (using the fact that the basis vectors have been normalized), to get

$$\beta_k = \mathbf{q}_k^* \mathbf{b} \tag{1.17}$$

Hence,

$$\mathbf{b} = \sum_{k=1}^{m} (\mathbf{q}_k^* \mathbf{b}) \mathbf{q}_k = \sum_{k=1}^{m} (\mathbf{q}_k \mathbf{q}_k^*) \mathbf{b} \tag{1.18}$$

where the second expression was obtained by commuting the scalar and the vector (which can always be done) and re-pairing the terms differently. While the first expression merely expresses $\mathbf{b}$ in the basis $\{\mathbf{q}_k\}_{k=1,\dots,m}$, the second expression writes $\mathbf{b}$ as a sum of vectors that are each orthogonal projections of $\mathbf{b}$ on each of the $\mathbf{q}_k$, and identifies the projection operator onto $\mathbf{q}_k$ as the matrix $\mathbf{q}_k \mathbf{q}_k^*$. This property will be explored at length later in the course.

3.2. Unitary matrices

An orthogonal matrix is a real square matrix $\mathbf{Q}$ of size $m \times m$, satisfying the property $\mathbf{Q}^T = \mathbf{Q}^{-1}$, or equivalently $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. Extending this to complex matrices, a **unitary matrix** is a complex square matrix $\mathbf{Q}$ of size $m \times m$, satisfying the property $\mathbf{Q}^* = \mathbf{Q}^{-1}$ or equivalently $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$.

Unitary (or orthogonal) matrices have the interesting property that their columns form an orthonormal basis. To see this, simply note that the ij coefficient of the product of two matrices is the inner product of the i-th row vector of the first matrix, with the $j-$th column vector of the second one. In order words, in the case of the product $\mathbf{Q}^*\mathbf{Q}$, and calling $\mathbf{q}_j$ the $j-$th column of $\mathbf{Q}$,

$$(\mathbf{Q}^*\mathbf{Q})_{ij} = \mathbf{q}_i^* \mathbf{q}_j = \mathbf{I}_{ij} = \delta_{ij} \tag{1.19}$$

where δ_{ij} is the Kronecker symbol. This proves that the pairwise product of any two column-vectors of $\mathbf{Q}$ is 0 if the vectors are different and 1 is the vectors are identical – the set $\{\mathbf{q}\}_{k=1,\dots,m}$ is an orthonormal set.

Another interesting property of unitary matrices is that their effect preserves lengths and angles (modulo the sign) between vectors, and as such they can be viewed either as **rotations** or **reflections**. To see why, note how the inner product between two vectors is preserved by the action of $\mathbf{Q}$ on both vectors:

$$(\mathbf{Qx})^*(\mathbf{Qy}) = \mathbf{x}^*\mathbf{Q}^*\mathbf{Qy} = \mathbf{x}^*\mathbf{y} \tag{1.20}$$

This in turn means that the Euclidean norm of a vector is preserved by the action of $\mathbf{Q}$, and this together with the interpretation of the inner product as a dot product, also shows that the cosines of the angles are preserved (with possible change of sign of the actual angle)

4. Eigenvalues and singular values

See Chapter 24 and Chapter 4 of the textbook

4.1. Eigenvalues and eigenvectors

4.1.1. Definition Let us now restrict our attention to square matrices $\mathbf{A}$ of size $m \times m$. If a vector $\mathbf{v}$ satisfies the property that

$$\mathbf{Av} = \lambda\mathbf{v} \tag{1.21}$$

for any scalar $\lambda \in \mathbb{C}$, then this vector is called an **eigenvector of A** and λ is the **eigenvalue** associated with that vector. Eigenvectors represent special directions along which the action of the matrix $\mathbf{A}$ reduces to a simple multiplication. Eigenvectors and eigenvalues have very general uses in many branches of applied mathematics, from the solution of PDEs, to the analysis of the stability of physical problems, etc. We shall see some of these examples later in the course.

Note that eigenvectors are not unique but they can be scaled infinitely different ways: if $\mathbf{Av} = \lambda\mathbf{v}$, then for any nonzero scalar γ, $\gamma\mathbf{v}$ is also an eigenvector corresponding to λ because $\mathbf{A}(\gamma\mathbf{v}) = \lambda(\gamma\mathbf{v})$. For this reason, we usually consider eigenvectors to be normalized. This tells us that the fundamental object of interest is not really any particular choice of eigenvector, but rather the *direction(s)* spanned by the eigenvector(s).

A few definitions: the space spanned by all eigenvectors associated with the same eigenvalue λ is called the **eigenspace** corresponding to λ. Within that eigenspace, multiplication by the matrix $\mathbf{A}$ reduces to a scalar multiplication. The set consisting of all the eigenvalues of $\mathbf{A}$ is called the **spectrum** of $\mathbf{A}$. Finally, the largest possible value of $|\lambda|$ over all the eigenvalues in the spectrum of $\mathbf{A}$ is called the **spectral radius** of $\mathbf{A}$, and is denoted by $\rho(\mathbf{A})$.

4.1.2. Characteristic Polynomials The equation $\mathbf{Av} = \lambda\mathbf{v}$ is equivalent to considering a homogeneous system of linear equations

$$\Big(\mathbf{A} - \lambda\mathbf{I}\Big)\mathbf{v} = \mathbf{0}. \tag{1.22}$$

Recall that this system has a nonzero solution $\mathbf{v}$ if and only if $\mathbf{A} - \lambda\mathbf{I}$ is singular, which is further equivalent to,

$$\det\Big(\mathbf{A} - \lambda\mathbf{I}\Big) = 0. \tag{1.23}$$

The relation in Eq. (1.23) is a polynomial of degree m in λ, and is called the **characteristic polynomial** $p_A(\lambda)$ of $\mathbf{A}$, which can be written as,

$$\begin{aligned} p_A(\lambda) &= \lambda^m + \cdots + c_1\lambda + c_0 && (1.24)\\ &= (\lambda - \lambda_1)(\lambda - \lambda_2)\cdots(\lambda - \lambda_m) && (1.25) \end{aligned}$$

with $c_k \in \mathbb{C}$ (or $c_k \in \mathbb{R}$ for real matrices). The roots of $p_A(\lambda)$ are the eigenvalues of $\mathbf{A}$. Note that this implies that even if a matrix is real, its eigenvalues may be complex. Finally, to find the eigenvectors, we then solve for each λ_k,

$$\mathbf{A}\mathbf{v}_k = \lambda_k\mathbf{v}_k. \tag{1.26}$$

Example: Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{1.27}$$

The characteristic polynomial of this matrix is

$$(\lambda - 2)^2 - 1 = \lambda^2 - 4\lambda + 3 = (\lambda - 3)(\lambda - 1) = 0, \tag{1.28}$$

therefore has two distinct roots $\lambda_1 = 1$ and $\lambda_2 = 3$, which are the eigenvalues of $\mathbf{A}$. To find the eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$, we solve

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \lambda_k \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{1.29}$$

which tells us that $2x_1 + x_2 = x_1$ for $\mathbf{v}_1$ and $2x_1 + x_2 = 3x_1$ for $\mathbf{v}_2$. This then shows that

$$\mathbf{v}_1 = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \text{ and } \mathbf{v}_2 = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \tag{1.30}$$

where both eigenvectors have been normalized. □

Note: This illustrates that when seeking eigenvalues and eigenvectors *analytically*, one usually first solves for the characteristic polynomial to find the eigenvalues λ_k, and then find the eigenvectors. As we shall see in this course the numerical approach to finding eigenvectors and eigvenvalues is very different. This is because the characteristic polynomial turns out not to be too useful as a means of actually computing eigenvalues for matrices of nontrivial size. Several issues that can arise in numerical computing include:

- computing the coefficients of $p_A(\lambda)$ for a given matrix $\mathbf{A}$ is, in general, a substantial task,

- the coefficients of $p_A(\lambda)$ can be highly sensitive to perturbations in the matrix $\mathbf{A}$, and hence their computation is unstable,
- rounding error incurred in forming $p_A(\lambda)$ can destroy the accuracy of the roots subsequently computed,
- computing the roots of any polynomial of high degree numerically is another substantial task.

4.1.3. Multiplicity, Defectiveness, and Diagonalizability Since the characteristic polynomial can be written as $p_A(\lambda) = \Pi_{i=1}^m(\lambda - \lambda_i)$, we can define the **algebraic multiplicity** of a particular eigenvalue λ_k simply as the multiplicity of the corresponding root of $p_A(\lambda)$, i.e., how many times the factor $(\lambda - \lambda_k)$ appears in $p_A(\lambda)$.

By contrast, the **geometric multiplicity** of λ_k is the dimension of the eigenspace associated with λ_k. In other words, it is the maximal number of linearly independent eigenvectors corresponding to λ_k.

In general, the algebraic multiplicity is always greater or equal to the geometric multiplicity. If the algebraic multiplicity is strictly greater than the geometric multiplicity then the eigenvalue λ_k is said to be *defective*. By extension, an $m \times m$ matrix that has fewer than m linearly independent eigenvectors is said to be **defective**.

Example: The matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & 3 \end{bmatrix} \tag{1.31}$$

has $p_A(\lambda) = (\lambda-1)(\lambda-3)+1 = (\lambda-2)^2 = 0$, hence has an eigenvalue $\lambda = 2$ as a double root. Therefore the **algebraic multiplicity** of $\lambda = 2$ is 2. Let's assume that there are two linearly independent eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$. If $\mathbf{v}_1 = (x_1, x_2)^T$ then x_1 and x_2 must satisfy $x_1 + x_2 = 2x_1$, or in other words $x_1 = x_2$ so $\mathbf{v}_1 = (1/\sqrt{2}, 1/\sqrt{2})^T$. Similarly we seek $\mathbf{v}_2 = (x_1, x_2)^T$, and its components must satisfy $-x_1 + 3x_2 = 2x_2$, or equivalently $x_1 = x_2$ again. This shows that $\mathbf{v}_1$ and $\mathbf{v}_2$ are actually the same vector, which implies that the **geometric multiplicity** of $\lambda = 2$ is only equal to one. This eigenvalue is therefore **defective**, and so is the matrix $\mathbf{A}$.

Example: $\lambda = 1$ is the only eigenvalue with algebraic multiplicity two for both matrices:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{1.32}$$

Its geometric multiplicity, however, is one for the first and two for the latter. □

If $m \times m$ matrix $\mathbf{A}$ is *nondefective*, then it has a full set of linearly independent eigenvectors $\mathbf{v}_1, \cdots, \mathbf{v}_m$ corresponding to the eigenvalues $\lambda_1, \cdots, \lambda_m$. If we let $\mathbf{D}$ be the diagonal matrix formed with all the eigenvalues λ_1 to λ_m, and $\mathbf{V}$

is the matrix formed by the column vectors $\mathbf{v}_1$ to $\mathbf{v}_m$ (in the same order), then $\mathbf{V}$ is nonsingular since the vectors are linearly independent and we have

$$\mathbf{AV} = \mathbf{VD} \tag{1.33}$$

To see why, note that

$$\mathbf{AV} = \begin{pmatrix} \mathbf{A}\mathbf{v}_1 & \mathbf{A}\mathbf{v}_2 & \dots & \mathbf{A}\mathbf{v}_m \end{pmatrix} = \begin{pmatrix} \lambda_1\mathbf{v}_1 & \lambda_2\mathbf{v}_2 & \dots & \lambda_m\mathbf{v}_m \end{pmatrix} \tag{1.34}$$

while

$$\mathbf{VD} = \begin{pmatrix} \mathbf{V}\lambda_1\mathbf{e}_1 & \mathbf{V}\lambda_2\mathbf{e}_2 & \dots & \mathbf{V}\lambda_m\mathbf{e}_m \end{pmatrix} = \begin{pmatrix} \lambda_1\mathbf{v}_1 & \lambda_2\mathbf{v}_2 & \dots & \lambda_m\mathbf{v}_m \end{pmatrix} \tag{1.35}$$

Multiplying on both sides by $\mathbf{V}^{-1}$ yields

$$\mathbf{V}^{-1}\mathbf{AV} = \mathbf{D}. \tag{1.36}$$

This shows that $\mathbf{A}$ is **diagonalizable**, i.e., can be put into a diagonal form using a **similarity transformation**.

4.1.4. Similarity transformation, change of base Let $\mathbf{A}$ and $\mathbf{B}$ be two $m \times m$ square matrices. Then $\mathbf{A}$ is **similar** to $\mathbf{B}$ if there is a nonsingular matrix $\mathbf{P}$ for which

$$\mathbf{B} = \mathbf{P}^{-1}\mathbf{AP}. \tag{1.37}$$

The operation $\mathbf{P}^{-1}\mathbf{AP}$ is called a **similarity transformation** of $\mathbf{A}$. Note that this is a symmetric relation (i.e., $\mathbf{B}$ is *similar* to $\mathbf{A}$), since

$$\mathbf{A} = \mathbf{Q}^{-1}\mathbf{BQ}, \text{ with } \mathbf{Q} = \mathbf{P}^{-1}. \tag{1.38}$$

A similarity transformation is simply a change of base for matrices, i.e. $\mathbf{B}$ can be interpreted as being the matrix $\mathbf{A}$ expressed in the basis formed by the column vectors of $\mathbf{P}$. As such, many of the geometric properties of $\mathbf{A}$ are preserved. To be specific, if $\mathbf{A}$ and $\mathbf{B}$ are similar, then the followings statements are true.

1. $p_A(\lambda) = p_B(\lambda)$.

   Proof: Then

$$\begin{aligned} p_B(\lambda) &= \det(\mathbf{B} - \lambda\mathbf{I}) \\ &= \det(\mathbf{P}^{-1}(\mathbf{A} - \lambda\mathbf{I})\mathbf{P}) \\ &= \det(\mathbf{P}^{-1})\det(\mathbf{A} - \lambda\mathbf{I})\det(\mathbf{P}) \\ &= \det(\mathbf{A} - \lambda\mathbf{I}) \\ &= p_A(\lambda). \end{aligned} \tag{1.39}$$

   since $\det(\mathbf{P}^{-1}) = 1/\det(\mathbf{P})$ for any nonsingular matrix. □

2. The eigenvalues of $\mathbf{A}$ and $\mathbf{B}$ are exactly the same, $\lambda(\mathbf{A}) = \lambda(\mathbf{B})$, and there is a one-to-one correspondence of the eigenvectors.

   Proof: Let $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. Then

   $$\mathbf{P}^{-1}\mathbf{A}\mathbf{P}(\mathbf{P}^{-1}\mathbf{v}) = \lambda\mathbf{P}^{-1}\mathbf{v}, \tag{1.40}$$

   or equivalently,

   $$\mathbf{B}\mathbf{u} = \lambda\mathbf{u}, \text{ with } \mathbf{u} = \mathbf{P}^{-1}\mathbf{v}. \tag{1.41}$$

   Also the one-to-one correspondence between $\mathbf{v}$ and $\mathbf{u}$ is trivial with the relationship $\mathbf{u} = \mathbf{P}^{-1}\mathbf{v}$, or $\mathbf{v} = \mathbf{P}\mathbf{u}$. □

3. The trace and determinant are unchanged. This can easily be shown from the fact that the characteristic polynomial remains the same:

   $$\begin{aligned} \text{trace}(\mathbf{A}) &= \text{trace}(\mathbf{B}), \qquad (1.42) \\ \det(\mathbf{A}) &= \det(\mathbf{B}). \qquad (1.43) \end{aligned}$$

4.2. Singular values and singular vectors

By contrast with eigenvectors and eigenvalues, singular vectors and singular values have a very simple geometric interpretation and do not require the matrix to be square in order to be computed. We therefore consider here any matrix $\mathbf{A}$ of size $m \times n$. The notion of singular values and singular vectors can be understood easily if one notes first that the image of the unit ball in $\mathbb{R}^n$ (or more generally $\mathbb{C}^n$) is a hyperellipse in $\mathbb{R}^m$ (or more generally $\mathbb{C}^m$).

Example: Consider the matrix $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$ then if $\mathbf{y} = \mathbf{A}\mathbf{x}$, and we only consider vectors $\mathbf{x} = (x_1, x_2)^T$ such that $x_1^2 + x_2^2 = 1$, their image $\mathbf{y} = (y_1, y_2)^T$ satisfies $y_1 = 2x_1 + x_2$ and $y_2 = x_2$ so that $x_2 = y_2$, $x_1 = (y_1 - y_2)/2$ and therefore

$$\frac{(y_1 - y_2)^2}{4} + y_2^2 = 1 \tag{1.44}$$

which is indeed the equation of a slanted ellipse (see Figure 1). □

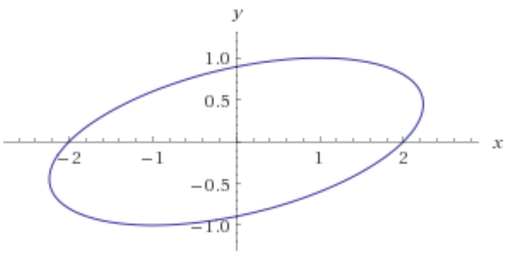


Figure 1. Image of the unit circle after application of the matrix $\mathbf{A}$.

Based on this geometrical interpretation of $\mathbf{A}$, we define **the singular values** of $\mathbf{A}$ as the lengths of the principal axes of the hyperellipse that is the image of the unit ball. They are denoted as $\{\sigma_i\}_{i=1,\dots,k}$, and are usually ordered, such that $\sigma_1 \geq \sigma_2 \geq \sigma_3 \cdots \geq \sigma_k > 0$. Note that k could be smaller than m. Then we define the **left singular vectors** of $\mathbf{A}$, $\{\mathbf{u}_i\}_{i=1,\dots,k}$, which are the directions of the principal axes of the hyperellipse corresponding to σ_i. Finally, we define the **right singular vectors** of $\mathbf{A}$, $\{\mathbf{w}_i\}_{i=1,\dots,k}$, such that $\mathbf{A}\mathbf{w}_i = \sigma_i \mathbf{u}_i$ for $i = 1, \dots, k$.

Further properties of the singular values and singular vectors, together with how to compute them, will be discussed later in the course.

5. Norms

See Chapter 3 of the textbook

In much of what we will learn in this course and in AMS 213B, we will need to have tools to measure things such as the quality of a numerical solution in comparison with the true solution (when it is known), or the rate of convergence of an iterative numerical algorithm to a solution. To do so usually involves measuring the size (loosely speaking) of a vector or a matrix, and in this respect we have several options that all fall under the general description of norms.

5.1. Vector norms

5.1.1. Definitions of vector norms A norm is a function that assigns a real valued number to each vector in $\mathbb{C}^m$. There are many possible definitions of vector norms, but they must all satisfy the following conditions:

- A norm must be positive for all non-zero vectors: $||\mathbf{x}|| > 0$ unless $\mathbf{x} = \mathbf{0}$, and $||\mathbf{0}|| = 0$.
- $||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}||$ for any two vectors $\mathbf{x}$ and $\mathbf{y}$. This is called the **triangle inequality**
- $||\alpha\mathbf{x}|| = |\alpha| ||\mathbf{x}||$ for any vectors $\mathbf{x}$ and any real α.

A large class of norms consists of the ***p*-norms**. The p-norm (or l^p-norm) of an n-vector $\mathbf{x}$ is defined by

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} \left| x_i \right|^p \right)^{\frac{1}{p}}. \tag{1.45}$$

Important special cases are:

- 1-norm:

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} \left| x_i \right| \tag{1.46}$$

- 2-norm: (also called the Euclidean length, defined in the previous section)

$$||\mathbf{x}||_2 = \left(\sum_{i=1}^{n} \left| x_i \right|^2 \right)^{\frac{1}{2}} = \sqrt{\mathbf{x}^T\mathbf{x}} \tag{1.47}$$

- ∞-norm (or max-norm):

$$||\mathbf{x}||_\infty = \max_{1\leq i\leq n} \left| x_i \right| \tag{1.48}$$

As we shall see, different norms come in handy in different kinds of applications although in practice the most useful ones are the $1-$norm, $2-$norm and ∞-norms.

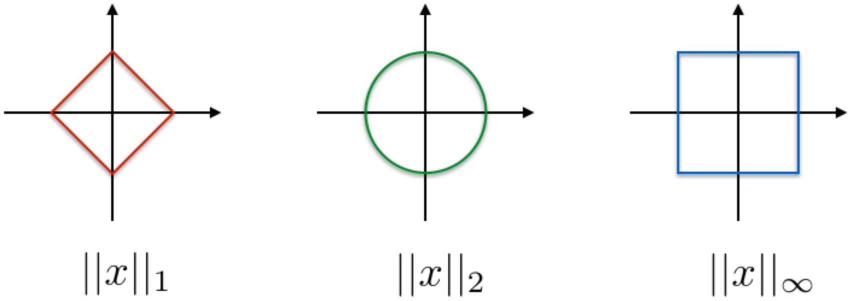


Figure 2. Illustrations of a unit sphere in $\mathbb{R}^2$, $||x|| = 1$, in three different norms: 1-norm, 2-norm and ∞-norm.

Example: For the vector $\mathbf{x} = (-1.6, 1.2)^T$, we get

$$||\mathbf{x}||_1 = 2.8, ||\mathbf{x}||_2 = 2.0, ||\mathbf{x}||_\infty = 1.6. \tag{1.49}$$

□

5.1.2. The Cauchy-Schwarz and Hölder inequalities When proving various theorems on stability or convergence of numerical algorithms, we shall sometimes use the Hölder inequality.

Theorem: *For any two vectors* $\mathbf{x}$ *and* $\mathbf{y}$*,*

$$|\mathbf{x}^*\mathbf{y}| \leq ||\mathbf{x}||_p||\mathbf{y}||_q, \tag{1.50}$$

where p and q satisfy $\frac{1}{p} + \frac{1}{q} = 1$ with $1 \leq p, q \leq \infty$.

The bound is tight in the sense that, if $\mathbf{x}$ and $\mathbf{y}$ are parallel vectors (such that $\mathbf{x} = \alpha\mathbf{y}$ for some α), the inequality becomes an equality. When applied to the $2-$norm, the inequality is called the Cauchy-Schwarz inequality.

5.2. Matrix norms

5.2.1. Definitions of matrix norms Similar to vector norms, matrix norms are functions that take a matrix and return a positive scalar, with the following properties:

- $||\mathbf{A}|| > 0$ unless $\mathbf{A} = \mathbf{0}$ in which case $||\mathbf{0}|| = 0$.
- $||\mathbf{A} + \mathbf{B}|| \leq ||\mathbf{A}|| + ||\mathbf{B}||$
- $||\alpha\mathbf{A}|| = |\alpha| ||\mathbf{A}||$.

We could, in principle, define matrix norms exactly as we defined vector norms, by summing over all components a_{ij} of the matrix. In fact, a commonly used norm is the **Hilbert-Schmidt norm**, also called the **Frobenius norm**:

$$||\mathbf{A}||_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2} \tag{1.51}$$

which is analogous to the $2-$norm for vectors.

An interesting property of this norm is that it can be written compactly as $||\mathbf{A}||_F = \sqrt{\mathrm{Tr}(\mathbf{A}\mathbf{A}^*)} = \sqrt{\mathrm{Tr}(\mathbf{A}^*\mathbf{A})}$. However, aside from this norm, other more useful definitions of norms can be obtained by defining a norm that measures the *effect* of a matrix on a vector.

Definition: The matrix p-norm of $m \times n$ matrix $\mathbf{A}$ can be defined by

$$||\mathbf{A}||_p = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{||\mathbf{A}\mathbf{x}||_p}{||\mathbf{x}||_p}. \tag{1.52}$$

In practice, what this does is to estimate the maximum amount by which the $p-$norm of a vector $\mathbf{x}$ can be *stretched* by the application of the matrix. And since any vector can be written as $\mathbf{x} = \alpha\hat{\mathbf{x}}$ where $\hat{\mathbf{x}}$ is a unit vector in the direction of $\mathbf{x}$, it suffices to look at the action of $\mathbf{A}$ on all possible unit vectors:

$$||\mathbf{A}||_p = \sup_{\hat{\mathbf{x}}} ||\mathbf{A}\hat{\mathbf{x}}||_p \tag{1.53}$$

Some matrix norms are easier to compute than others. For instance, by this definition, and by the definition of singular values, we simply have

$$||\mathbf{A}||_2 = \sigma_1 \tag{1.54}$$

that is, the length of the largest principal axis of the image of the unit ball.

Let's also look for instance at the case of the $1-$norm, for an $m \times n$ matrix $\mathbf{A}$. We begin by computing

$$||\mathbf{A}\hat{\mathbf{x}}||_1 = || \sum_{i=1}^{n} \mathbf{a}_i \hat{x}_i ||_1 \leq \sum_{i=1}^{n} |\hat{x}_i| ||\mathbf{a}_i||_1 \tag{1.55}$$

using the column vectors $\mathbf{a}_i$ of $\mathbf{A}$, and the triangle inequality. Then, using the fact that $||\hat{\mathbf{x}}||_1 = \sum_{i=1}^n |\hat{x}_i| = 1$, and noting that $||\mathbf{a}_i||_1 \leq \max_i ||\mathbf{a}_i||_1$ for any i, we have

$$||\mathbf{A}\hat{\mathbf{x}}||_1 \leq \sum_{i=1}^n |\hat{x}_i| ||\mathbf{a}_i||_1 \leq \max_i ||\mathbf{a}_i||_1 \sum_{i=1}^n |\hat{x}_i| = \max_i ||\mathbf{a}_i||_1 \tag{1.56}$$

so $||\mathbf{A}\hat{\mathbf{x}}||_1 \leq \max_i ||\mathbf{a}_i||_1$. As it turns out, the inequality becomes an equality when $\hat{\mathbf{x}} = \mathbf{e}_I$, where I is the index i for which $||\mathbf{a}_i||_1$ is largest, so the maximum possible value of $||\mathbf{A}\hat{\mathbf{x}}||_1$ over all possible unit vectors $\hat{\mathbf{x}}$ is indeed $\max_i ||\mathbf{a}_i||_1$. In short

$$||\mathbf{A}||_1 = \max_j \sum_{i=1}^m \left| a_{ij} \right| \tag{1.57}$$

It can similarly be shown that

$$||\mathbf{A}||_\infty = \max_i \sum_{j=1}^n \left| a_{ij} \right| \tag{1.58}$$

Finally, the $p-$ norm of *diagonal matrices* is easy to compute: $||\mathbf{D}||_p = \max_i |d_i|$, where d_i is the i-th diagonal component of the matrix $\mathbf{D}$.

5.2.2. Properties of matrix norms Similar to the Hölder inequality, it is possible to bound the norm of a product of two matrices. For any one of the $p-$norms associated with a vector norm, or for the Hilbert-Schmidt/Frobenius norms, we have

$$||\mathbf{AB}|| \leq ||\mathbf{A}|| ||\mathbf{B}|| \tag{1.59}$$

In general, the inequality is strict. However, in the special case where one of the matrices is unitary (orthogonal), and where we use the $2-$norm or the Hilbert-Schmidt/Frobenius norms, we have

$$||\mathbf{QA}||_2 = ||\mathbf{A}||_2 \text{ and } ||\mathbf{QA}||_F = ||\mathbf{A}||_F \tag{1.60}$$

6. Properties of Machine Arithmetics

See Chapter 13 of textbook + additional material on computer representation of numbers from various sources

So far, we have merely summarized the most salient results of Linear Algebra that will come in handy when trying to design numerical algorithms to solve linear algebra problems. Up to now, every result discussed was exact. At this point in time, however, we must now learn about one of the most important issues related to the numerical implementation of numerical algorithms, namely, the fact that numbers are not represented *exactly*, but rather, *approximately*, in the computer.

Most computers have different modes for representing integers and real numbers, *integer mode* and *floating-point mode*, respectively. Let us now take a look at these modes.

6.1. Integer mode

The representation of an integer number is (usually) *exact*. Recall that we can represent an integer as a sequence of numbers from 0 to 9, for instance, as an expansion in base 10:

$$a_n a_{n-1} \cdots a_0 = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \cdots + a_0 \times 10^0. \tag{1.61}$$

Example: Base 10

$$159 = 1 \times 10^2 + 5 \times 10^1 + 9 \times 10^0. \tag{1.62}$$

□

However, the number base used in computers is seldom decimal (e.g., base 10), but instead binary (e.g., base 2) where one “bit” is either 0 or 1. In binary form, any positive integers can be written as

$$a_n a_{n-1} \cdots a_0 = a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_0 \times 2^0. \tag{1.63}$$

Example: Base 2

$$\begin{aligned} 159 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 10011111_2. \end{aligned} \tag{1.64}$$

□

A **signed integer** is typically stored on a finite number of bytes (recall, 1 byte = 8 bits), usually using 1-bit for the sign (though other conventions also exist).

In Fortran there are two common ways to represent integers, *normal* and *long* types. The normal integers are stored on 4 bytes, or equivalently 32 bits where one bit is reserved for the sign and the rest 31 bits for the value itself. On the other hand, the long integers are stored on 8 bytes which are equivalent to 64 bits, where one bit for the sign and the rest 63 bits for the value.

As a consequence for a given integer type there are only finite numbers of integers which can be used in programing:

- for normal 4-byte integers: between -2^{31} and 2^{31},
- for normal 8-byte integers: between -2^{63} and 2^{63}.

This means that any attempts to reach numbers beyond these values will cause problems. Note that we have $2^{31} \approx 2.1$ billion which is not so big a number.

6.2. Floating-point mode

The base 10 notation (decimal) for real numbers can be written as

$$\begin{aligned}
&a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m \\
&= a_n \times 10^n + a_{n-1} \times 10^{n-1} + \cdots + a_0 \times 10^0 \\
&+ b_1 \times 10^{-1} + b_2 \times 10^{-2} + \cdots + b_m \times 10^{-m},
\end{aligned} \tag{1.65}$$

and by analogy we write a real number in base 2 (binary) as

$$\begin{aligned}
&a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m \\
&= a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_0 \times 2^0 \\
&+ b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_m \times 2^{-m},
\end{aligned} \tag{1.66}$$

Definition: We note that we can only store finite numbers of a_i and b_j as every computer has finite storage limit. This implies that there are cases when real numbers can only be *approximated* with finitely many combinations of a_i and b_j. The error associated with this approximation is called *roundoff errors.*

6.2.1. Standard notations In fact, the numbers are not stored as written above. Rather, they are stored as

$$2^n \Big(a_n + a_{n-1} 2^{-1} + \cdots + a_0 2^{-n} + b_1 2^{-n-1} + \cdots + b_m 2^{-n-m} \Big), \tag{1.67}$$

or

$$2^{n+1} \Big(a_n 2^{-1} + a_{n-1} 2^{-2} + \cdots + a_0 2^{-n-1} + b_1 2^{-n-2} + \cdots + b_m 2^{-n-m-1} \Big). \tag{1.68}$$

In the first case a_n can be chosen to be nonzero by assumption, which necessarily gives $a_n = 1$. The first is referred to as *IEEE standard* and the second as *DEC standard.*

Example: The representation of 27.25 in base 2 becomes

$$\begin{aligned}
&27.25 \\
&= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\
&= 11011.01_2,
\end{aligned} \tag{1.69}$$

which can be written in two ways as just shown above:

$$11011.01_2 = \begin{cases} 2^4 \Big(1.1011101_2 \Big) \text{ IEEE standard,} \\ \\ 2^5 \Big(0.11011101_2 \Big) \text{ DEC standard.} \end{cases} \tag{1.70}$$

□

Definition: In general, this takes of the form of

$$x = 2^k \times f, \tag{1.71}$$

where k and f are called the *exponent* and *mantissa*, respectively.

6.2.2. Standard Fortran storage types: double vs. single precisions There are two standard storage types available in Fortran. In addition to them, one can define any new type as needed in Fortran 90 and above. The two standard storage types are

- single precision : `type REAL(SP)`. Storage is on 4 bytes (i.e., 32 bits = 1 bit for sign + 8 bits for the exponent + 23 bits for the mantissa),
- double precision : `type REAL(DP)`. Storage on 8 bytes (i.e., 64 bits = 1 bit for sign + 11 bits for the exponent + 52 bits for the mantissa).

Note: The bits in the exponent store integers from L to U, where usually, L is the lowest exponent, a negative integer; and U is the highest exponent, a positive integer, with

$$U - L \approx \begin{cases} 2^8 \text{ for single precision,} \\ \\ 2^{11} \text{ for double precision,} \end{cases} \tag{1.72}$$

where

$$L = \begin{cases} -126 \text{ for single precision,} \\ \\ -1022 \text{ for double precision,} \end{cases} \tag{1.73}$$

and

$$U = \begin{cases} 127 \text{ for single precision,} \\ \\ 1023 \text{ for double precision.} \end{cases} \tag{1.74}$$

□

6.2.3. Floating-point arithmetic and roundoff errors Arithmetic using floating-point numbers is *quite* different from real arithmetic. In order to understand this let's work in base 10 for simplicity. If we represent π in, say, DEC standard, we get

- a representation up to 2 decimal places (significant digits) is 0.31, and
- a representation up to 6 decimal places (significant digits) is 0.314159.

For a given number of significant digits (i.e., a given length of mantissa), the "distance" between two consecutive numbers is dependent on the value of the exponent. Let us consider the following example.

Example: Suppose we have 3 possible values of the exponent, $k = -2, -1$ and 0, and 2 digits for mantissa. Positive numbers we can possibly create from this condition are

$$\left.\begin{array}{c} 0.10 \times 10^{-2} \\ 0.11 \times 10^{-2} \\ \vdots \\ 0.98 \times 10^{-2} \\ 0.99 \times 10^{-2} \end{array}\right\} \text{all separated by } 10^{-4} \tag{1.75}$$

$$\left.\begin{array}{c} 0.10 \times 10^{-1} \\ 0.11 \times 10^{-1} \\ \vdots \\ 0.98 \times 10^{-1} \\ 0.99 \times 10^{-1} \end{array}\right\} \text{all separated by } 10^{-3} \tag{1.76}$$

$$\left.\begin{array}{c} 0.10 \times 10^{0} \\ 0.11 \times 10^{0} \\ \vdots \\ 0.98 \times 10^{0} \\ 0.99 \times 10^{0} \end{array}\right\} \text{all separated by } 10^{-2} \tag{1.77}$$

Here we note that the continuous real line has been discretized which inevitably introduces roundoff errors. Also, the discretization does not produce equidistance uniform spacing, instead the non-uniform spacing depends on the absolute value of the numbers considered. □

Note: The floating-point arithmetic operation is not associative as a result of roundoff errors. To see this, let's consider a case with 6-digit mantissa. Let's take three real numbers,

$$a = 472635 = 0.472635 \times 10^6, \tag{1.78}$$

$$b = 472630 = 0.472630 \times 10^6, \tag{1.79}$$

$$c = 27.5013 = 0.275013 \times 10^2. \tag{1.80}$$

We see that the floating-point operation fails to preserve associative rule,

$$(a - b) + c \neq a - (b - c). \tag{1.81}$$

In the first case, we have

$$\begin{aligned} (a-b)+c &= (472635 - 472630) + 27.5013 \\ &= 5.00000 + 27.5013 \\ &= 32.5013, \end{aligned} \tag{1.82}$$

whereas the second case gives

$$\begin{aligned} a-(b-c) &= 472635-(472630-27.5013) \\ &= 472635-\underbrace{472602.4987}_{\text{more than 6 digits hence must be rounded off}} \\ &= 472635-472602 \\ &= 33.0000. \end{aligned} \tag{1.83}$$

As can be seen the error on the calculation is huge! It is of the order of the discretization error for the largest of the numbers considered (i.e., a and b in this case). □

6.2.4. Machine accuracy ϵ It is a similar concept – now with the question of what is the *largest* number ϵ that can be added to 1 such that in floating-point arithmetic, one gets

$$1+\epsilon=1? \tag{1.84}$$

Let's consider the following example:

Example: Consider 6-digit mantissa. Then we have

$$1=0.100000\times 10^{1}, \tag{1.85}$$

and then

$$1+10^{-7}=0.1000001\times 10^{1}. \tag{1.86}$$

However, the last representation exceeds 6-digit limit and hence needs to be rounded *down* to 0.100000×10^{1}, resulting

$$1+10^{-7}=0.100000\times 10^{1}. \tag{1.87}$$

This implies that the machine accuracy is $\epsilon\approx 10^{-7}$. □

Note: For floating-point arithmetic in base 2, with mantissa of size m, we have

$$\epsilon\approx 2^{-m}=\begin{cases} 2^{-23}\approx 10^{-7} \text{ in real single precision,} \\ 2^{-52}\approx 10^{-16} \text{ in real double precision.} \end{cases} \tag{1.88}$$

□

6.2.5. Overflow and underflow problems There exists a smallest and a largest number (in absolute value) that can be represented in floating-point notation. For instance, let us suppose that the exponent k ranges from -4 to 4, and the mantissa has 8 significant digits. This gives us that the smallest possible number in base 10 is

$$x_{\min}=0.10000000\times 10^{-4}=10^{-4-1}, \tag{1.89}$$

and the largest possible number is

$$x_{\max} = 0.9999999 \times 10^4 \approx 10^4. \tag{1.90}$$

Therefore in general, in base 2, we have

$$x_{\min} = 2^{L-1} = \begin{cases} 2^{-127} \approx 10^{-38} \text{ in real single precision,} \\ 2^{1023} \approx 10^{-308} \text{ in real double precision.} \end{cases} \tag{1.91}$$

$$x_{\max} = 2^{U} = \begin{cases} 2^{127} \approx 10^{38} \text{ in real single precision,} \\ 2^{1023} \approx 10^{308} \text{ in real double precision.} \end{cases} \tag{1.92}$$

If the outcome of a floating-point operation yields $|x| < x_{\min}$ then an underflow error occurs. In this case x will be set to be zero usually and computation will continue. In contrast, if $|x| > x_{\max}$ then an overflow error occurs, causing a fatal termination of program.

7. Conditioning and condition number

See Chapter 12 of the textbook

Having seen that numbers are not represented exactly in the numerical world, one may well begin to worry about the effects of this approximate representation on simple things such as matrix operations. In other words, what happens (for instance) to $\mathbf{Ax}$ if the entries of $\mathbf{x}$ are only known approximately, or if the entries of $\mathbf{A}$ are only known approximately? Does this have a significant effect or a small effect on $\mathbf{Ax}$?

As it turns out, this notion is quite general and does not necessarily only apply to matrices. It is called **conditioning** and is more generally applied to any function of $\mathbf{x}$.

Loose definition: A function $\mathbf{f}(\mathbf{x})$ (where $\mathbf{f}$ and $\mathbf{x}$ are either scalars or vectors) is *well-conditioned* near the point $\mathbf{x}_0$ provided small perturbation $\delta\mathbf{x}$ around $\mathbf{x}_0$ only lead to small perturbations $\delta\mathbf{f} = \mathbf{f}(\mathbf{x}_0 + \delta\mathbf{x}) - \mathbf{f}(\mathbf{x}_0)$. A problem is *ill-conditioned* if a small $\delta\mathbf{x}$ leads to a large $\delta\mathbf{f}$.

What small and large mean, in this definition, can depend on the application of interest. We would also like to create a number (the **condition number**) that can become a diagnostic of the conditioning properties of a problem. To do so, we now introduce the following more mathematical definitions.

7.1. Absolute condition number

For a fixed $\mathbf{x}_0$, we define the **absolute condition number** at $\mathbf{x} = \mathbf{x}_0$ as

$$\hat{\kappa}(\mathbf{x}_0) = \lim_{\epsilon \to 0} \sup_{||\delta \mathbf{x}|| \leq \epsilon} \frac{||\delta \mathbf{f}||}{||\delta \mathbf{x}||} \tag{1.93}$$

where $\delta \mathbf{f}$ was defined above. This is therefore the limit, when ϵ tends to 0, of the maximum possible value of $||\delta \mathbf{f}||/||\delta \mathbf{x}||$ over all possible $\delta \mathbf{x}$ whose norm is less than or equal to ϵ.

If the function $\mathbf{f}(\mathbf{x})$ is differentiable, then we can write $\delta \mathbf{f} = \mathbf{J}_0 \delta \mathbf{x}$ in the limit $\delta \mathbf{x} \to \mathbf{0}$, where $\mathbf{J}_0$ is the Jacobian of $\mathbf{f}$ (i.e. the matrix of partial derivatives) at $\mathbf{x} = \mathbf{x}_0$. In that case

$$\hat{\kappa}(\mathbf{x}_0) = \lim_{\epsilon \to 0} \sup_{||\delta \mathbf{x}|| \leq \epsilon} \frac{||\delta \mathbf{f}||}{||\delta \mathbf{x}||} = \lim_{\epsilon \to 0} \sup_{||\delta \mathbf{x}|| \leq \epsilon} \frac{||\mathbf{J}_0 \delta \mathbf{x}||}{||\delta \mathbf{x}||} = ||\mathbf{J}(\mathbf{x}_0)|| \tag{1.94}$$

for any $p-$norm. In short,

$$\hat{\kappa}(\mathbf{x}_0) = ||\mathbf{J}(\mathbf{x}_0)|| \tag{1.95}$$

7.2. Relative condition number

In many cases, especially when working with floating point arithmetic, it makes more sense to establish the conditioning of relative changes $||\delta \mathbf{f}||/||\mathbf{f}||$ rather than of $||\delta \mathbf{f}||$ itself. To do so, we consider instead the **relative condition number** at $\mathbf{x} = \mathbf{x}_0$, defined as

$$\kappa(\mathbf{x}_0) = \lim_{\epsilon \to 0} \sup_{||\delta \mathbf{x}|| \leq \epsilon} \frac{||\delta \mathbf{f}||/||\mathbf{f}||}{||\delta \mathbf{x}||/||\mathbf{x}||} \tag{1.96}$$

Using the same trick as above for differentiable functions $\mathbf{f}(\mathbf{x})$, we then have

$$\kappa(\mathbf{x}_0) = \frac{||\mathbf{J}_0|| ||\mathbf{x}_0||}{||\mathbf{f}(\mathbf{x}_0)||} \tag{1.97}$$

The relative condition number is more commonly used in numerical linear algebra because the rounding errors (which can be the cause of $\delta \mathbf{x}$ and therefore $\delta \mathbf{f}$) are relative to a given $\mathbf{x}_0$, see the previous sections.

Example 1: Consider the function $f(x) = \sqrt{x}$, whose derivative (Jacobian) at any point $x > 0$ is $0.5x^{-1/2}$. The relative condition number

$$\kappa = \left| \frac{f'(x)x}{f(x)} \right| = \left| \frac{x}{2\sqrt{x}\sqrt{x}} \right| = \frac{1}{2} \tag{1.98}$$

is finite and small, suggesting a well-conditioned problem. Note, however, that the absolute condition number

$$\hat{\kappa} = \left| f'(x) \right| = \frac{1}{2\sqrt{x}} \to \infty \text{ as } x \to 0 \tag{1.99}$$

suggesting ill-conditioning as $x \to 0$. Should we worry about it? The answer is no – this is a specific case where we care more about the relative changes than the absolute changes in δx, since it doesn't make sense to take the limit $x \to 0$ unless $|\delta x|$ also goes to 0. □

Example 2: Consider the function $f(x_1, x_2) = x_2 - x_1$. The Jacobian of this function is the row-vector $\mathbf{J} = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{pmatrix} = (-1 \;\; 1)$. The ∞-norm is the sum of the absolute values of the components of that row (see previous section), namely $||\mathbf{J}||_\infty = 2$, so

$$\kappa(x_1, x_2) = \frac{||\mathbf{J}||_\infty ||\mathbf{x}||_\infty}{|f(x_1, x_2)|} = \frac{2 \max(|x_1|, |x_2|)}{|x_2 - x_1|} \tag{1.100}$$

In this case, we see that there can be serious ill-conditioning in terms of relative errors when $x_1 \to x_2$. This recovers our earlier findings for floating point arithmetic that the truncation errors are always of the order of the largest terms (here x_1 and x_2), so the relative error on $x_2 - x_1$ can be huge compared with $x_2 - x_1$. □

7.3. Condition number of Matrix-Vector multiplications

Suppose we consider the function that takes a vector $\mathbf{x}$ and multiplies the matrix $\mathbf{A}$ to it: $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$. In this case, by definition the Jacobian matrix $\mathbf{J}$ *is* the matrix $\mathbf{A}$, so we can immediately construct the relative condition number as

$$\kappa(\mathbf{x}_0) = \frac{||\mathbf{A}|| ||\mathbf{x}_0||}{||\mathbf{A}\mathbf{x}_0||} \tag{1.101}$$

Generally speaking, we would have to stop here, and evaluate κ. However, suppose we know that $\mathbf{A}$ is an invertible square matrix. Then if we write $\mathbf{x}_0 = \mathbf{A}^{-1}\mathbf{A}\mathbf{x}_0$, we have, by this equality, that $||\mathbf{x}_0|| = ||\mathbf{A}^{-1}\mathbf{A}\mathbf{x}_0|| \le ||\mathbf{A}^{-1}|| ||\mathbf{A}\mathbf{x}_0||$ using the matrix multiplication bound discussed in the previous section. So

$$\kappa(\mathbf{x}_0) = \frac{||\mathbf{A}|| ||\mathbf{x}_0||}{||\mathbf{A}\mathbf{x}_0||} \le ||\mathbf{A}|| ||\mathbf{A}^{-1}|| \tag{1.102}$$

regardless of $\mathbf{x}_0$, for any non-singular matrix $\mathbf{A}$.

7.4. Condition number of the solution of a set of linear equations

Suppose we now want to solve the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{b}$ is known exactly, but some uncertainty exists in the entries of $\mathbf{A}$. What are the impacts of this uncertainty on the numerical evaluation of the solution $\mathbf{x}$? Since solving this problem requires computing the function $\mathbf{x} = \mathbf{f}(\mathbf{b}) = \mathbf{A}^{-1}\mathbf{b}$, which is a matrix multiplication by $\mathbf{A}^{-1}$, we can use all of the results of the previous section to show that the condition number of this problem is bounded by

$$\kappa(\mathbf{b}) \le ||\mathbf{A}^{-1}|| ||\mathbf{A}|| \tag{1.103}$$

as before.

7.5. Condition number of a matrix

The quantity $||\mathbf{A}||||\mathbf{A}^{-1}||$ is so important and comes up so often that it is often referred to simply as **the condition number** of the matrix $\mathbf{A}$, $\text{cond}(\mathbf{A})$. If we use the $2-$norm to compute it, then $||\mathbf{A}||_2 = \sigma_1$, and it can also be shown that $||\mathbf{A}^{-1}|| = 1/\sigma_m$. In that case,

$$\text{cond}(\mathbf{A}) = ||\mathbf{A}||||\mathbf{A}^{-1}|| = \frac{\sigma_1}{\sigma_m} \tag{1.104}$$

and the condition number of $\mathbf{A}$ can be re-interpreted as being related to the eccentricity of the hyperellipse image of the unit ball.

The following important properties of the condition number are easily derived from the definition using the $2-$norm, and in fact hold for any norm:

1. For any matrix $\mathbf{A}$, $\text{cond}(\mathbf{A}) \geq 1$.
2. For the identity matrix, $\text{cond}(\mathbf{I}) = 1$.
3. For any matrix $\mathbf{A}$ and nonzero γ, $\text{cond}(\gamma\mathbf{A}) = \text{cond}(\mathbf{A})$.
4. For any diagonal matrix $\mathbf{D}$, $\text{cond}(\mathbf{D}) = \frac{\max_i |d_{ii}|}{\min_i |d_{ii}|}$ or in other words, is the ratio of the largest to the smallest eigenvalue (in absolute value).

Final remarks:

- These results shows that if $\mathbf{A}$ is ill-conditioned, then numerical algorithms that are prone to round-off errors will have problems both with simple matrix multiplications by $\mathbf{A}$, and with matrix multiplications by $\mathbf{A}^{-1}$ (or equivalently, with solving any linear problem of the kind $\mathbf{Ax} = \mathbf{b}$).
- The reason for this is that the condition number effectively measures how close a matrix is to being singular: a matrix with a large condition number has its smallest singular value very close to zero, which means that the matrix operation is fairly insensitive to anything that happens along that singular direction. Another way of seing this is that this direction is very close to being part of the nullspace. A matrix with a condition number close to 1 on the other hand is far from being singular.
- Notice that the determinant of a matrix is *not* a good indicator of near singularity. In other words, the magnitude of $\det(\mathbf{A})$ has no information on how close to singular the matrix $\mathbf{A}$ may be. For example, $\det(\alpha\mathbf{I}_n) = \alpha^n$. If $|\alpha| < 1$ the determinant can be very small, yet the matrix $\alpha\mathbf{I}_n$ is perfectly well-conditioned for any nonzero α.
- The usefulness of the condition number is in accessing the accuracy of solutions to linear system. However, the calculation of the condition number is not trivial as it involves the inverse of the matrix. Therefore, in practice, one often seeks for a good estimated approach to approximate condition numbers.

Chapter 2

Solutions of systems of Linear Equations

There are many problems in science and engineering that requires the solution of a linear problem of the kind

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where $\mathbf{A}$ is a square $m \times m$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are both m-long column vectors. In what follows, we will assume that the problem is non-singular, that is, that we have somehow already demonstrated that $\det(\mathbf{A}) \neq 0$.

1. Examples of applications that require the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$

1.1. Fitting a hyperplane to m points in $\mathbb{R}^m$

Given 2 distinct points on a plane ($\mathbb{R}^2$), there is only 1 line that passes through both points. Similarly, given 3 distinct *non-aligned* points in 3D space ($\mathbb{R}^3$), there is only 1 plane that goes through all three points. Given 4 distinct *non-coplanar* points in 4D space ($\mathbb{R}^4$), there is only 1 hyperplane that goes through all four points – and so forth! Finding the equation of this hyperplane requires solving a linear system of the kind $\mathbf{A}\mathbf{x} = \mathbf{b}$. Indeed, the equation for an $m-$dimensional hyperplane is the linear equation

$$a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots + a_m x_m = \text{const} \tag{2.2}$$

where $\mathbf{x} = (x_1, \ldots, x_m)^T$ are the coordinates of a point in the hyperplane, the coefficients $\{a_i\}_{i=0,\ldots,m}$ are real numbers, and the constant const is arbitrary and can be chosen to be equal to one without loss of generality. Given m points in the hyperplane denoted as $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(m)}$, we therefore have the system of equations

$$\begin{aligned}
&a_1 x_1^{(1)} + a_2 x_2^{(1)} + \cdots + a_m x_m^{(1)} = 1 \\
&a_1 x_1^{(2)} + a_2 x_2^{(2)} + \cdots + a_m x_m^{(2)} = 1 \\
&\vdots \\
&a_1 x_1^{(m)} + a_2 x_2^{(m)} + \cdots + a_m x_m^{(m)} = 1
\end{aligned} \tag{2.3}$$

which can be re-cast as $\mathbf{A}\mathbf{x} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_m^{(2)} \\ & & \vdots & \\ x_1^{(m)} & x_2^{(m)} & \dots & x_m^{(m)} \end{pmatrix}, \mathbf{x} = \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_m \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ \dots \\ 1 \end{pmatrix} \tag{2.4}$$

The solution of this equation is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ provided $\mathbf{A}$ is nonsingular.

1.2. Solving a partial differential equation

Suppose we want to solve the PDE

$$\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial x^2} \tag{2.5}$$

with some given initial condition $f(x, 0) = f_0(x)$. As you shall see at length in AMS 213B, a simple way of solving the problem consists in discretizing this equation both in space and time: letting space be discretized as a series of closely spaced $\{x_i\}$ so $x_{i+1} - x_i = \Delta x$, and similarly time be discretized as $\{t^{(n)}\}$ so $t^{(n+1)} - t^{(n)} = \Delta t$, we rewrite the PDE as the discrete equation

$$\frac{f_i^{(n+1)} - f_i^{(n)}}{\Delta t} = \frac{f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)}}{\Delta x^2} \tag{2.6}$$

where $f_i^{(n)} \equiv f(x_i, t^{(n)})$. This is called an explicit scheme, because we can then simply write the solution at the next timestep *explicitly* as a function of the solution at the previous timestep.

$$f_i^{(n+1)} = f_i^{(n)} + \frac{\Delta t}{\Delta x^2} \left(f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)} \right) \tag{2.7}$$

This can actually be cast more simply as the matrix equation $\mathbf{f}^{(n+1)} = \mathbf{C}\mathbf{f}^{(n)}$ where $\mathbf{C} = \mathbf{I} + \frac{\Delta t}{\Delta x^2}\mathbf{M}$ and

$$\mathbf{M} = \begin{pmatrix} \ddots & \ddots & \ddots & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \text{ and } \mathbf{f}^{(n)} = \begin{pmatrix} \vdots \\ f_{i-1}^{(n)} \\ f_i^{(n)} \\ f_{i+1}^{(n)} \\ \vdots \end{pmatrix} \tag{2.8}$$

In this algorithm, $\mathbf{f}^{(n+1)}$ can be obtained simply by matrix multiplication. However, this explicit algorithm is subject to satisfy a strict stability constraint (see AMS213B for more information), and better results can be obtained, both in terms of stability and accuracy, using a Crank-Nicholson scheme, in which

$$\frac{f_i^{(n+1)} - f_i^{(n)}}{\Delta t} = \frac{1}{2}\frac{f_{i+1}^{(n)} - 2f_i^{(n)} + f_{i-1}^{(n)}}{\Delta x^2} + \frac{1}{2}\frac{f_{i+1}^{(n+1)} - 2f_i^{(n+1)} + f_{i-1}^{(n+1)}}{\Delta x^2} \tag{2.9}$$

that is, by evaluating the time derivative half way between the timesteps $t^{(n)}$ and $t^{(n+1)}$. This algorithm, by contrast with the previous one, becomes $\mathbf{A}\mathbf{f}^{(n+1)} = \mathbf{B}\mathbf{f}^{(n)}$ where $\mathbf{A} = \mathbf{I} - \frac{\Delta t}{2\Delta x^2}\mathbf{M}$ and $\mathbf{B} = \mathbf{I} + \frac{\Delta t}{2\Delta x^2}\mathbf{M}$. In order to advance the solution in time, we therefore have to solve a matrix problem for $\mathbf{f}^{(n+1)}$. In principle, this can be done by finding the inverse of $\mathbf{A}$, and evaluating $\mathbf{f}^{(n+1)} = \mathbf{A}^{-1}\mathbf{B}\mathbf{f}^{(n)}$ at each timestep. While both examples require solving a matrix problem, their

practical use in large-scale computations (e.g. fitting *many* hyperplanes different sets of points, or advancing the PDE in time for *many* timesteps) is quite different. In the first case, each set of points gives rise to a different matrix $\mathbf{A}$, so the problem needs to be solved from scratch every time. In the second case on the other hand, assuming that the timestep Δt and the grid spacing Δx remain constant, the matrix $\mathbf{A}$ remains the same each time, so it is worth solving for $\mathbf{A}^{-1}$ just once ahead of time, save it, and then simply multiply each new right-hand-side by $\mathbf{A}^{-1}$ to evolve the solution forward in time. Or something similar, at the very least (as we shall see, things are done a little different in practice).

2. A little aside on invariant transformations

To solve a linear system, we usually transform it step by step into one that is much easier to solve, making sure in the process that the solution remains unchanged. In practice, this can easily be done in linear algebra noting that multiplication of the equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ by any nonsingular matrix $\mathbf{M}$ on both sides, namely:

$$\mathbf{M}\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{b} \tag{2.10}$$

leaves the solution unchanged. Indeed, let $\mathbf{z}$ be the solution of Eqn. 2.10. Then

$$\mathbf{z} = (\mathbf{M}\mathbf{A})^{-1}\mathbf{M}\mathbf{b} = \mathbf{A}^{-1}\mathbf{M}^{-1}\mathbf{M}\mathbf{b} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{x}. \tag{2.11}$$

These transformations, i.e., multiplication by a non-singular matrix, are called **invariant transformations**.

2.1. Permutations

A permutation matrix $\mathbf{P}$, a square matrix having exactly one 1 in each row and column and zeros elsewhere – which is also always a nonsingular – can always be multiplied without affecting the original solution to the system. For instance,

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.12}$$

permutes $\mathbf{v}$ as

$$\mathbf{P}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} v_3 \\ v_1 \\ v_2 \end{bmatrix}. \tag{2.13}$$

□

The same permutation matrix applied to a 3×3 matrix $\mathbf{A}$ would shuffle its rows in the same way. More generally, permutation matrices are operations that shuffle the rows of a matrix.

2.2. Row scaling

Another invariant transformation exists which is called *row scaling*, an outcome of a multiplication by a diagonal matrix $\mathbf{D}$ with nonzero diagonal entries $d_{ii}, i = 1, \dots m$. In this case, we have

$$\mathbf{DAx} = \mathbf{Db}, \tag{2.14}$$

by which each row of the transformed matrix $\mathbf{DA}$ gets to be scaled by d_{ii} from the original matrix $\mathbf{A}$. Note that the scaling factors are cancelled by the same scaling factors introduced on the right hand side vector, leaving the solution to the original system unchanged.

Note: The column scaling does not preserve the solution in general. □

3. Gaussian elimination

Chapter 2.2 of Numerical Recipes, and Chapter 20 of the textbook

Recall that in this chapter we are interested in solving a well-defined linear system given as

$$\mathbf{Ax} = \mathbf{b}, \tag{2.15}$$

where $\mathbf{A}$ is a $m \times m$ *square* matrix and $\mathbf{x}$ and $\mathbf{b}$ are m-vectors. The most standard algorithm for the solution of linear systems learned in introductory linear algebra classes is Gaussian elimination. Gaussian elimination proceeds in steps sweeping the matrix from left to right, and successively zeroing out (using clever linear operations on the rows), all the coefficients below the diagonal. The same operations are carried out on the right-hand-side, to ensure invariance of the solution.

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix} \tag{2.16}$$

The end product is an upper triangular matrix, and the associated set of linear equations can then be solved by back-substitution. Here is a sample algorithm describing the steps required for Gaussian elimination.

Algorithm: Basic Gaussian elimination:

```
do $j = 1$ to $m - 1$
    ![loop over column]
    if $a_{jj} = 0$ then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do $i = j + 1$ to $m$
    ![sweep over rows $\mathbf{r}_i$ below row $\mathbf{r}_j$]
        $\mathbf{r}_i = \mathbf{r}_i - \mathbf{r}_j a_{ij}/a_{jj}$
        ![zeros terms below $a_{jj}$ and transforms rest of matrix]

        $b_i = b_i - b_j a_{ij}/a_{jj}$
        ![carries over same operation on RHS]
    enddo
enddo
```

Note that the operation $\mathbf{r}_i = \mathbf{r}_i - \mathbf{r}_j a_{ij}/a_{jj}$ is another loop over all terms in row $\mathbf{r}_i$. It guarantees to zero out all of the elements in the column j below the diagonal, and since it only operates on rows below the diagonal, it does not affect any of the terms that have already been zeroed out.

This algorithm is however problematic for a few reasons. The first is that it stops if the diagonal term a_{jj} is zero, which may well happen even if the matrix is non-singular. The second problem is that even for non-singular, well-conditioned matrices, this algorithm is not very stable. Both problems are discussed in the next section. Finally it is also quite wasteful since it spends time calculating entries that we already know are zero. More on this later.

It is worth noting that the operations described in the previous algorithm can be written formally in terms of invariant transformations. Indeed, suppose we define the matrix $\mathbf{M}_1$ so that its action on $\mathbf{A}$ is to zero out the elements in the first column below the first row, and apply it to both the left-hand-side and right-hand-sides of $\mathbf{Ax} = \mathbf{b}$. Again, we repeat this process in the next step so that we find $\mathbf{M}_2$ such that the second column of $\mathbf{M}_2\mathbf{M}_1\mathbf{A}$ becomes zero below the second row, along with applying the equivalent multiplication on the right hand side, $\mathbf{M}_2\mathbf{M}_1\mathbf{b}$. This process is continued for each successive column until all of the subdiagonal entries of the resulting matrix have been annihilated.

If we define the final matrix $\mathbf{M} = \mathbf{M}_{n-1} \cdots \mathbf{M}_1$, the transformed linear system becomes

$$\mathbf{M}_{n-1} \cdots \mathbf{M}_1 \mathbf{Ax} = \mathbf{MAx} = \mathbf{Mb} = \mathbf{M}_{n-1} \cdots \mathbf{M}_1 \mathbf{b}. \tag{2.17}$$

To show that this is indeed an invariant transformation, we simply have to show that $\mathbf{M}$ is non-singular. The easiest (non-rigorous) way of showing this is to look at the structure of $\mathbf{M}$, through an example.

Example: Consider

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 10 \\ 1 \end{bmatrix} = \mathbf{b}. \tag{2.18}$$

The first question is to find a matrix $\mathbf{M}_1$ that annihilates the subdiagonal entries of the first column of $\mathbf{A}$. This can be done if we consider a matrix $\mathbf{M}_1$ that can subtract twice the first row from the second row, four times the first row from the third row, and three times the first row from the fourth row. The matrix $\mathbf{M}_1$ is then identical to the identity matrix $\mathbf{I}_4$, except for those multiplication factors in the first column:

$$\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}, \tag{2.19}$$

where we treat the blank entries to be zero entries.

The next step is to annihilate the third and fourth entries from the second column (3 and 4), which gives the next matrix $\mathbf{M}_2$ that has the form:

$$\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}, \tag{2.20}$$

The last matrix $\mathbf{M}_3$ completes the process, resulting an upper triangular matrix $\mathbf{U}$:

$$\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = \mathbf{U}, \tag{2.21}$$

together with the right hand side:

$$\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{b} = \begin{bmatrix} 3 \\ 0 \\ -2 \\ -6 \end{bmatrix} = \mathbf{y}. \tag{2.22}$$

We see that the matrix $\mathbf{M}$ formed in the process is the product of three lower-triangular matrices, which is itself lower triangular (you can easily check this out!). Since lower triangular matrices are singular if and only if one of their diagonal entries is zero (which is not the case since they are all 1), $\mathbf{M}$ is non-singular. □

Example: In the previous example, we see that the final transformed linear system $\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \mathbf{MA}$ yields $\mathbf{MAx} = \mathbf{Mb}$ which is equivalent to $\mathbf{Ux} = \mathbf{y}$, an upper triangular system which we wanted and can be solved easily by back-substitution, starting from obtaining $x_4 = -3$, followed by x_3, x_2, and x_1 in reverse order to find a complete solution

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ -3 \end{bmatrix}. \tag{2.23}$$

Furthermore, the full LU factorization of $\mathbf{A}$ can be established as $\mathbf{A} = \mathbf{LU}$ if we compute

$$\mathbf{L} = (\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1)^{-1} = \mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1}. \tag{2.24}$$

At first sight this looks like an expensive process as it involves inverting a series of matrices. Surprisingly, however, this turns out to be a trivial task. The inverse of $\mathbf{M}_i$, $i = 1, 2, 3$ is just itself but with each entry below the diagonal negated. Therefore, we have

$$\begin{aligned}
\mathbf{L} &= \mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1} \\
&= \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix}^{-1} \\
&= \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & 3 & 1 & \\ & 4 & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}.
\end{aligned} \tag{2.25}$$

Notice also that the matrix multiplication $\mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_3^{-1}$ is also trivial and is just the unit lower triangle matrix with the nonzero subdiagonal entries of $\mathbf{M}_1^{-1}$, $\mathbf{M}_2^{-1}$, and $\mathbf{M}_3^{-1}$ inserted in the appropriate places. (Notice that the similar is not true for $\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1$.)

All together, we finally have our decomposition $\mathbf{A} = \mathbf{LU}$:

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}. \tag{2.26}$$

□

While we have proved it only for this particular example, it is easy to see how the proof could generalize for any non-singular matrix $\mathbf{A}$, and in the process, suggest how to write down a much more compact version of the Gaussian elimination algorithm:

Algorithm:

```
do $j = 1$ to $m - 1$
    $\mathbf{M}_j = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & -a_{j+1,j}/a_{jj} & 1 & & & \\ & & -a_{j+2,j}/a_{jj} & & \ddots & & \\ & & \vdots & & & \ddots & \\ & & -a_{m,j}/a_{jj} & & & & 1 \end{pmatrix}$
    $\mathbf{A} = \mathbf{M}_j \mathbf{A}$
    $\mathbf{b} = \mathbf{M}_j \mathbf{b}$
enddo
```

Note, however, that creating the matrix $\mathbf{M}_j$ and multiplying both $\mathbf{A}$ and $\mathbf{b}$ by it at every iteration is very wasteful both in memory and time, so this more compact version is never used in practical implementations of the algorithm. It is only really useful for illustration purposes, and sometimes for proving theorems.

In any case, the resulting transformed linear system is $\mathbf{MAx} = \mathbf{Ux} = \mathbf{Mb} = \mathbf{y}$, where $\mathbf{M} = \mathbf{M}_{m-1} \dots \mathbf{M}_1$ is lower triangular and $\mathbf{U}$ is upper triangular. The equivalent problem $\mathbf{Ux} = \mathbf{y}$ can be solved by back-substitution to obtain the solution to the original linear system $\mathbf{Ax} = \mathbf{b}$. This is done simply as

Algorithm: Backsubstitution of $\mathbf{Ux} = \mathbf{y}$ ($\mathbf{U}$ is upper triangular):

```
$x_m = y_m/u_{mm}$ ![stop if $u_{mm}$ is zero, singular matrix]

do $i = m - 1$ to 1
    ![loop over lines, bottom to top]
    if $u_{ii} = 0$ then
        stop ![stop if entry is zero, singular matrix]
    endif
    sum = 0.0 ![initialize to zero first]
    do $k = i + 1$ to $m$
        sum = sum + $u_{ik}x_k$
    enddo
    $x_i = (y_i - \mathrm{sum})/u_{ii}$
enddo
```

Note:

- Some algorithms simply return the solution within the RHS vector $\mathbf{y}$ instead of returning it as a separate vector $\mathbf{x}$. To do so simply replace the last operation with $y_i = (y_i - \mathtt{sum})/u_{ii}$, which gradually overwrites the entries of $\mathbf{y}$ as they are no longer needed.

- It is very easy to do the same operations at the same time on many RHS vectors $\mathbf{y}$. To do so, create a matrix $\mathbf{Y}$ formed by all the RHS column-vectors and perform the Gaussian elimination and backsubstitution on the whole matrix $\mathbf{Y}$ at the same time (see, e.g., Numerical Recipes for an example).

4. Gaussian elimination with pivoting

We obviously run into trouble when the choice of a divisor – called a **pivot** – is zero, whereby the Gaussian elimination algorithm breaks down. The solution to this singular pivot issue is fairly straightforward: if the pivot entry is zero at stage k, i.e., $a_{kk} = 0$, then we interchange row k of *both* the matrix and the right hand side vector with some subsequent row whose entry in column k is nonzero and resume the process as usual. Recall that permutation is an invariant transformation that does not alter the solution to the system.

This row interchanging is part of a process called **pivoting**, which is illustrated in the following example.

Example: Suppose that after zeroing out the subdiagonal in the first column, the next diagonal entry is zero (red line). Then simply swap it with one of the rows below that, e.g. here the blue one.

$$\begin{bmatrix} * & * & * & * \\ 0 & 0 & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{\mathbf{P}} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \end{bmatrix} \tag{2.27}$$

where the permutation matrix $\mathbf{P}$ is given as

$$\mathbf{P} = \begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix}. \tag{2.28}$$

□

Problems do not only occur with zero pivots, but also when the pivots are very small, i.e. close to or below machine precision ϵ_{mach}. Recall that we have $\epsilon_{\text{mach}} \approx 10^{-7}$ for single precision, and $\epsilon_{\text{mach}} \approx 10^{-16}$ for double precision.

Example: Let us consider the problem

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \tag{2.29}$$

where $\epsilon < \epsilon_{\text{mach}} \approx 10^{-16}$, say, $\epsilon = 10^{-20}$. The real solution of this problem is

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{1-\epsilon} \\ \frac{1-2\epsilon}{1-\epsilon} \end{pmatrix} \simeq \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{2.30}$$

However, let's see what a numerical algorithm would give us. If we proceed without any pivoting (i.e., no row interchange) then the first step of the elimination algorithm gives

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2-\epsilon^{-1} \end{pmatrix}, \tag{2.31}$$

which is numerically equivalent to

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ -\epsilon^{-1} \end{pmatrix}, \tag{2.32}$$

in floating point arithmetic since $1/\epsilon \simeq 10^{20} \gg 1$. The solution via back-substitution is then

$$y = \frac{-\epsilon^{-1}}{-\epsilon^{-1}} = 1 \text{ and } x = \frac{1-1}{\epsilon} = 0 \tag{2.33}$$

which is very, very far from being the right answer!! We see that using a small pivot, and a correspondingly large multiplier, has caused an unrecoverable loss of information in the transformation. Also note that the original matrix here is far from being singular, it is in fact a very well-behaved matrix.

As it turns out, we can easily cure the problem by interchanging the two rows first, which gives

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \tag{2.34}$$

so this time the elimination proceeds as

$$\begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1-2\epsilon \end{pmatrix}, \tag{2.35}$$

which is numerically equal to

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \tag{2.36}$$

in floating-point arithmetic, and whose solution is $\mathbf{x} = (1,1)^T$. This time, we actually get the correct answer within machine accuracy. □

The foregoing example is rather extreme, but large errors would also occur even if we had $\epsilon_{\text{mach}} \ll \epsilon \ll 1$. This suggests a general principle in which we want to make sure to always work with the largest possible pivot, in order to produce smaller errors in floating-point arithmetic, and stabilizes an otherwise very unstable algorithm. Gaussian elimination with **partial pivoting** therefore proceeds as below. At step j, find $p = \max_{k=j,\dots,m} |a_{kj}|$ and select it as the $j-$th pivot. Then switch the lines j and K (if $j \neq K$), where K is the value of k for which $|a_{kj}|$ is maximal before zeroing out the subdiagonal elements of column j.

$$\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & a_{kj} & * & * \\ & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ & a_{kj} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * \\ & a_{kj} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix} \tag{2.37}$$

Note that it is possible to use **full pivoting**, i.e. by selecting a pivot from *all* the entries in the lower right submatrix below the element a_{jj}. In practice, however, the extra counting and swapping work required is not usually worth it.

Algorithm: Gaussian elimination with Partial Pivoting:

```
do $j = 1$ to $m - 1$
    ![loop over column]
    Find index $K$ and pivot $p$ such that $p = |a_{Kj}| = \max_{k=j,\dots,m} |a_{kj}|$
    if $K \neq j$ then
        interchange rows $K$ and $j$
        ![interchange rows if needed]
    endif
    if $a_{jj} = 0$ then
        stop
        ![matrix is singular]
    endif
    do $i = j + 1$ to $m$
    ![loop over rows below row $j$]
        $\mathbf{r}_i = \mathbf{r}_i - a_{ij}\mathbf{r}_j / a_{jj}$
        ![transformation of remaining submatrix]
        $b_i = b_i - a_{ij} b_j / a_{jj}$
        ![transformation of RHS vector]
    enddo
enddo
```

Finally, note that there is a slight oddity in the algorithm in the sense that if a whole row of the matrix is multiplied by a large number, and similarly the corresponding entry in the RHS is multiplied by the same value, then this row is *guaranteed* to contain the first pivot even though the actual problem is exactly the same as the original one. If you are worried about this, you may consider using the **implicit pivoting** algorithm, where each row of the augmented matrix (i.e. matrix and RHS) is first scaled by its largest entry in absolute value (see Numerical Recipes for detail).

5. Gauss-Jordan elimination for the calculation of the inverse of a matrix.

Chapter 2.1 of Numerical Recipes

Gaussian elimination (with pivoting) works very well as long as the RHS vector(s) $\mathbf{b}$ is (are) known ahead of time, since the algorithm needs to operate on the RHS at the same time as it operates on $\mathbf{A}$. This is fine for applications such as fitting linear functions to a set of points (as in the first example above), where each new set of points gives rise to a new matrix problem that needs to be solved from scratch. On the other hand, as discussed above, the evolution of the solution of the PDE using the Crank-Nicholson algorithm from timesteps $t^{(n)}$ to $t^{(n+1)}$ requires the solution of $\mathbf{A}\mathbf{f}^{(n+1)} = \mathbf{B}\mathbf{f}^{(n)}$, where the right-hand-side vector $\mathbf{B}\mathbf{f}^{(n)}$ changes at each timestep $t^{(n)}$. In this case it is better to calculate the inverse of $\mathbf{A}$, store it, and then merely perform the matrix multiplication $\mathbf{f}^{(n+1)} = \mathbf{A}^{-1}\mathbf{B}\mathbf{f}^{(n)}$ at each timestep to advance the solution (although, see also the next section on LU factorization).

A very basic and direct method for obtaining and storing the inverse of a matrix $\mathbf{A}$ is to use the so-called **Gauss-Jordan elimination** on the **augmented matrix** formed by $\mathbf{A}$ and $\mathbf{I}$.

Definition: Let us introduce a form called **augmented matrix** of the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ which writes the $m \times m$ matrix $\mathbf{A}$ and the m-vector $\mathbf{b}$ together in a new $m \times (m+1)$ matrix form:

$$\left[\mathbf{A}\middle|\mathbf{b}\right]. \tag{2.38}$$

The use of augmented matrix allows us to write each transformation step of the linear system (i.e., both $\mathbf{A}$ and $\mathbf{b}$) in a compact way. Note that we could have done this for Gaussian elimination too.

Example: Consider the following system using Gauss-Jordan elimination without pivoting:

$$\begin{array}{llll} x_1 & +x_2 & +x_3 & = 4 \\ 2x_1 & +2x_2 & +5x_3 & = 11 \\ 4x_1 & +6x_2 & +8x_3 & = 24 \end{array}, \tag{2.39}$$

which can be put in as an augmented matrix form:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 5 & 11 \\ 4 & 6 & 8 & 24 \end{array}\right]. \tag{2.40}$$

First step is to annihilate the first column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 2 & 5 & 11 \\ 4 & 6 & 8 & 24 \end{array}\right] \xrightarrow{\mathbf{M}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 0 & 3 & 3 \\ 0 & 2 & 4 & 8 \end{array}\right], \text{ where } \mathbf{M}_1 = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ -4 & & 1 \end{bmatrix}. \tag{2.41}$$

Next we permute to get rid of the zero (so there is some basic pivoting involved):

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 0 & 3 & 3 \\ 0 & 2 & 4 & 8 \end{array}\right] \xrightarrow{\mathbf{P}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 3 & 3 \end{array}\right], \text{ where } \mathbf{P}_1 = \begin{bmatrix} 1 & & \\ & & 1 \\ & 1 & \end{bmatrix}. \tag{2.42}$$

Next row scaling by multiplying a diagonal matrix $\mathbf{D}_1$:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & 3 & 3 \end{array}\right] \xrightarrow{\mathbf{D}_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{D}_1 = \begin{bmatrix} 1 & & \\ & 1/2 & \\ & & 1/3 \end{bmatrix}. \tag{2.43}$$

Next annihilate the remaining upper diagonal entries in the third column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 1 \end{array}\right] \xrightarrow{\mathbf{M}_2} \left[\begin{array}{ccc|c} 1 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{M}_2 = \begin{bmatrix} 1 & & -1 \\ & 1 & -2 \\ & & 1 \end{bmatrix}. \tag{2.44}$$

Finally, annihilate the upper diagonal entry in the second column:

$$\left[\begin{array}{ccc|c} 1 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right] \xrightarrow{\mathbf{M}_3} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right], \text{ where } \mathbf{M}_3 = \begin{bmatrix} 1 & -1 & \\ & 1 & \\ & & 1 \end{bmatrix}. \tag{2.45}$$

□

In this example the right hand side is a single m-vector. What happens if we perform the same procedure using multiple m-vectors? Then we get the augmented matrix

$$\left[\mathbf{A}\middle|\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \cdots \sqcup \mathbf{b}_n\right]. \tag{2.46}$$

We see that the same operation can easily be performed simultaneously on individual $\mathbf{b}_i, 1 \le i \le n$.

Especially, if we choose m vectors $\mathbf{b}_i = \mathbf{e}_i$ then the matrix formed by these column vectors is the identity matrix $\mathbf{I}$. In this case we see that Gauss-Jordan elimination yields the inverse of $\mathbf{A}$, that is, the solution of $\mathbf{AX} = \mathbf{I}$.

$$\left[\mathbf{A}\middle|\mathbf{e}_1 \sqcup \mathbf{e}_2 \sqcup \cdots \sqcup \mathbf{e}_n\right] = \left[\mathbf{A}\middle|\mathbf{I}\right] \to \cdots \to \left[\mathbf{I}\middle|\mathbf{A}^{-1}\right]. \tag{2.47}$$

Quick summary: The process of calculating the inverse of a matrix by Gauss-Jordan elimination can be illustrated as in the following pictorial steps:

$$\begin{aligned}
&\left[\begin{array}{cccc|cccc} * & * & * & * & 1 & 0 & 0 & 0 \\ * & * & * & * & 0 & 1 & 0 & 0 \\ * & * & * & * & 0 & 0 & 1 & 0 \\ * & * & * & * & 0 & 0 & 0 & 1 \end{array}\right] \to \left[\begin{array}{cccc|cccc} 1 & * & * & * & * & 0 & 0 & 0 \\ 0 & * & * & * & * & 1 & 0 & 0 \\ 0 & * & * & * & * & 0 & 1 & 0 \\ 0 & * & * & * & * & 0 & 0 & 1 \end{array}\right] \\
\to &\left[\begin{array}{cccc|cccc} 1 & 0 & * & * & * & * & 0 & 0 \\ 0 & 1 & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & 1 & 0 \\ 0 & 0 & * & * & * & * & 0 & 1 \end{array}\right] \to \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & * & * & * & * & 0 \\ 0 & 1 & 0 & * & * & * & * & 0 \\ 0 & 0 & 1 & * & * & * & * & 0 \\ 0 & 0 & 0 & * & * & * & * & 1 \end{array}\right] \\
\to &\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & * & * & * & * \\ 0 & 1 & 0 & 0 & * & * & * & * \\ 0 & 0 & 1 & 0 & * & * & * & * \\ 0 & 0 & 0 & 1 & * & * & * & * \end{array}\right]
\end{aligned} \tag{2.48}$$

where the matrix on the RHS of the last iteration is $\mathbf{A}^{-1}$. □

Note that pivoting is just as important for Gauss-Jordan elimination as it is for Gaussian elimination. As a result, here is a basic algorithm for Gauss-Jordan elimination with partial pivoting. This assumes that $\mathbf{C} = [\mathbf{A}|\mathbf{B}]$ where $\mathbf{A}$ is a non-singular $m \times m$ matrix, and $\mathbf{B}$ is $m \times n$ matrix consisting of all the RHS vectors. If $\mathbf{B} = \mathbf{I}$, then the algorithm returns $\mathbf{A}^{-1}$ in the place of $\mathbf{B}$. Otherwise, it simply returns the solution to $\mathbf{AX} = \mathbf{B}$ in that place.

Algorithm: Gauss-Jordan elimination with Partial Pivoting:

```
do $j = 1$ to $m$
    ![loop over column]
    Find index $K$ and pivot $p$ such that $p = |c_{Kj}| = \max_{k=j,\ldots,m} |c_{kj}|$
    if $K \neq j$ then
        interchange rows $K$ and $j$
    endif
    if $c_{jj} = 0$ then
        stop ![matrix is singular]
    endif
    $\mathbf{r}_j = \mathbf{r}_j / c_{jj}$     ![scale row $j$ so diagonal element is 1]
    do $i = 1$ to $m$
        if $i \neq j$ then
            ![loop over all rows except $j$]
            $\mathbf{r}_i = \mathbf{r}_i - c_{ij}\mathbf{r}_j$
            ![transformation of remaining submatrix]
        endif
    enddo
enddo
```

6. Operation counts for basic algorithms

The two main *efficiency* concerns for numerical linear algebra algorithm are:

- time efficiency
- storage efficiency

While these concerns are mild for small problems, they can become very serious when the matrices are very large. In this section, we will rapidly look at the first one. A good way of estimating the efficiency and speed of execution of an algorithm is to count the number of floating point operations performed by the algorithm.

Example 1: Matrix multiplication $\mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$ are $m \times m$ matrices
In computing $\mathbf{C} = \mathbf{AB}$, we have to compute the m^2 coefficients of $\mathbf{C}$, which each involves calculating

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{2.49}$$

or in other words, m multiplications and m additions. The total operation count is therefore of order $2m^3$, although because multiplications and divisions are much more costly than additions and subtractions, operation counts usually ignore the latter when they are dominated by the former. So the number of operations in matrix multiplications is $\sim m^3$. □

Example 2: Back-substitution of $\mathbf{U}\mathbf{x} = \mathbf{y}$, where $\mathbf{U}$ is an $m \times m$ matrix Running through the algorithm we have

$$
\begin{aligned}
x_m &= \frac{y_m}{u_{mm}} \rightarrow \text{ 0 add/sub, 1 mult/div} \\
x_{m-1} &= \frac{y_{m-1} - u_{m-1,m}x_m}{u_{m-1,m-1}} \rightarrow \text{ 1 add/sub, 2 mult/div} \\
x_{m-2} &= \frac{y_{m-2} - u_{m-2,m-1}x_{m-1} - u_{m-2,m}x_m}{u_{m-2,m-2}} \rightarrow \text{ 2 add/sub, 3 mult/div} \\
&\vdots \\
x_1 &= \frac{y_1 - \sum_{k=2}^m u_{1,k}x_k}{u_{11}} \rightarrow m - 1\text{add/sub and } m \text{ mult/div}
\end{aligned}
\tag{2.50}
$$

This gives an operation count of order $m(m+1)/2$ multiplications/divisions, and $m(m-1)/2$ additions and subtractions, so a gran total of $\sim m^2/2$ multiplication/division operations per RHS vector if m is large. If we do a backsubstitution on n vectors simultaneously, then the operation count is $m^2n/2$. □

Operation counts have been computed for the algorithms discussed so far for the solution of linear systems, namely Gaussian elimination + back-substitution and Gauss-Jordan elimination. This gives, for n right-hand-sides

- Gaussian elimination + backsubstitution: of order $\frac{m^3}{3} + \frac{m^2n}{2} + \frac{m^2n}{2}$.
- Gauss-Jordan elimination : of order $\frac{m^3}{2} + \frac{m^2n}{2}$.

so the latter is about 1.5 times as expensive as the former for small m, but 33% cheaper for $m = n$. In the case of matrix inversion, however, *if we cleverly avoid computing entries that we know are zero anyway* (which requires re-writing the GE algorithm on purpose), the operation counts between GE and GJ are the same, and about m^3 in both cases.

Finally, suppose we now go back to the problem of advancing a PDE forward in time, save the inverse, and apply it by matrix multiplication to each new right-hand-side. In this case, the matrix multiplication of a single vector at each timestep takes m^2 multiplications and m^2 additions.

7. LU factorization

Chapter 20 of the textbook

Another very popular way of solving linear systems, that has the advantage of having a minimal operation count *and* has the flexibility to be used as in the PDE problem, in a deferred way with multiple RHS vectors but the same matrix $\mathbf{A}$, is the **LU algorithm**. The idea is the following. Suppose we find a way of decomposing the matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{2.51}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. In that case, the system $\mathbf{Ax} = \mathbf{b}$ is equivalent to $\mathbf{LUx} = \mathbf{b}$, which can be solved in two steps:

- Solve $\mathbf{Ly} = \mathbf{b}$
- Solve $\mathbf{Ux} = \mathbf{y}$

The second of these steps looks very familiar – in fact, it is simply the same backsubstitution step as in the Gaussian elimination algorithm. Furthermore, we saw that Gaussian elimination can be interpreted as a sequence of multiplications by lower triangular matrices to transform $\mathbf{A}$ into $\mathbf{U}$, and this can be used to construct the LU decomposition.

7.1. Relationship between Gaussian elimination and LU factorization

Ignoring pivoting for a moment, recall that transforming the matrix $\mathbf{A}$ via Gaussian elimination into an upper-triangular matrix involves writing $\mathbf{U}$ as

$$\mathbf{U} = \mathbf{MA} = \mathbf{M}_{m-1} \dots \mathbf{M}_1 \mathbf{A} \tag{2.52}$$

where

$$\mathbf{M}_j = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{j+1,j} & 1 & & \\ & & -l_{j+2,j} & & \ddots & \\ & & \vdots & & & \ddots \\ & & -l_{m,j} & & & & 1 \end{pmatrix}$$

and the coefficients l_{ij} are constructed from the elements below the diagonal of the matrix $\mathbf{M}_{j-1} \dots \mathbf{M}_1 \mathbf{A}$ (i.e. the matrix obtained at the previous step of the iteration process).

It is easy to show that since all the $\mathbf{M}_j$ matrices are lower triangular, then so is their product $\mathbf{M}$. Furthermore, it is also relatively easy to show that the inverse of a lower-triangular matrix is also lower triangular, so writing $\mathbf{MA} = \mathbf{U}$ is equivalent to $\mathbf{A} = \mathbf{M}^{-1}\mathbf{U}$ which is in LU form with $\mathbf{L} = \mathbf{M}^{-1}$! All that remains to be done is to evaluate $\mathbf{M}^{-1}$. As it turns out, calculating the inverse of $\mathbf{M}$ is actually fairly trivial as long as the coefficients l_{ij} of each matrix $\mathbf{M}_j$ are known – and this can be done by Gaussian elimination.

Indeed, first note that if we define the vector $\mathbf{l}_j = (0, 0, \dots, 0, l_{j+1,j}, l_{j+2,j}, \dots, l_{m,j})^T$, then $\mathbf{M}_j$ can be written more compactly in terms of $\mathbf{l}_j$ as $\mathbf{M}_j = \mathbf{I} - \mathbf{l}_j \mathbf{e}_j^*$. We can then verify that $\mathbf{M}_j^{-1} = \mathbf{I} + \mathbf{l}_j \mathbf{e}_j^*$ simply by constucting the product

$$\mathbf{M}_j \mathbf{M}_j^{-1} = (\mathbf{I} - \mathbf{l}_j \mathbf{e}_j^*)(\mathbf{I} + \mathbf{l}_j \mathbf{e}_j^*) = \mathbf{I} - \mathbf{l}_j \mathbf{e}_j^* + \mathbf{l}_j \mathbf{e}_j^* - \mathbf{l}_j \mathbf{e}_j^* \mathbf{l}_j \mathbf{e}_j^* \tag{2.53}$$

Now it's easy to check that $\mathbf{e}_j^* \mathbf{l}_j$ is zero, proving that $\mathbf{M}_j \mathbf{M}_j^{-1} = \mathbf{I}$. In other words, finding the inverse of $\mathbf{M}_j$ merely requires negating its subdiagonal components. □

Next, another remarkable property of the matrices $\mathbf{M}_j$ and their inverses is that their product $\mathbf{L} = \mathbf{M}^{-1} = \mathbf{M}_1^{-1}\mathbf{M}_2^{-1} \dots \mathbf{M}_{m-1}^{-1}$ can also very easily be calculated. Indeed,

$$\mathbf{M}_i^{-1}\mathbf{M}_j^{-1} = (\mathbf{I} + \mathbf{l}_i\mathbf{e}_i^*)(\mathbf{I} + \mathbf{l}_j\mathbf{e}_j^*) = \mathbf{I} + \mathbf{l}_i\mathbf{e}_i^* + \mathbf{l}_j\mathbf{e}_j^* + \mathbf{l}_i\mathbf{e}_i^*\mathbf{l}_j\mathbf{e}_j^* \tag{2.54}$$

The last term is zero as long as $i \leq j$, which is always the case in the construction of $\mathbf{M}^{-1}$. This is because $\mathbf{e}_i^*\mathbf{l}_j = 0$, because the product would be equal to the i-th component of the vector $\mathbf{l}_j$, which is zero as long as $i \leq j$. This shows that $\mathbf{M}_i^{-1}\mathbf{M}_j^{-1} = \mathbf{I} + \mathbf{l}_i\mathbf{e}_i^* + \mathbf{l}_j\mathbf{e}_j^*$, and so $\mathbf{L} = \mathbf{M}^{-1} = \mathbf{I} + \mathbf{l}_1\mathbf{e}_1^* + \mathbf{l}_2\mathbf{e}_2^* + \dots + \mathbf{l}_{m-1}\mathbf{e}_{m-1}^*$, which is the matrix

$$\mathbf{L} = \begin{pmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{m1} & l_{m2} & \dots & l_{m,m-1} & 1 \end{pmatrix}$$

These considerations therefore suggest the following algorithm:

Algorithm: LU factorization by Gaussian elimination (without pivoting), version 1:

```
do $j = 1$ to $m - 1$
    ![loop over columns]
    if $a_{jj} = 0$ then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do $i = j + 1$ to $m$
        $l_{ij} = a_{ij}/a_{jj}$
        ![create the non-zero coefficients of $\mathbf{l}_j$]
    enddo
    $\mathbf{A} = \mathbf{A} - \mathbf{l}_j\mathbf{e}_j^*\mathbf{A}$
    ![Overwrite $\mathbf{A}$ by $\mathbf{M}_j\mathbf{A}$]
enddo
```

□

Note that the operation $\mathbf{A} = \mathbf{A} - \mathbf{l}_j\mathbf{e}_j^*\mathbf{A}$ can either be written component-wise, or as vector operations as written here, in which case the most efficient way of doing this is to evaluate first $\mathbf{e}_j^*\mathbf{A}$ then multiply by $\mathbf{l}_j$. Also note that this algorithm doesn't discuss storage (i.e. where to put the l_{ij} coefficients, etc...), and as such is merely illustrative. At the end of this algorithm, the matrix $\mathbf{A}$ becomes the matrix $\mathbf{U}$ and the matrix $\mathbf{L}$ can be formed separately if needed by the combination of all the $\mathbf{l}_j$ vectors, as discussed above.

A common way of storing the entries l_{ij} is to actually put them in the matrix $\mathbf{A}$, gradually replacing the zeroes that would normally occur by Gaussian elimination. This is very efficient storage-wise, but then prohibits the use of the compact form of the algorithm, requiring instead that the operation $\mathbf{A} = \mathbf{A} - \mathbf{l}_j \mathbf{e}_j^* \mathbf{A}$ be written out component-wise. The end product, after completion, are the matrix $\mathbf{L}$ and $\mathbf{U}$ stored into the matrix $\mathbf{A}$ as

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1,m-1} & u_{1m} \\ l_{21} & u_{22} & u_{23} & \dots & u_{2,m-1} & u_{2m} \\ l_{31} & l_{32} & u_{33} & \dots & u_{3,m-1} & u_{3m} \\ \vdots & \dots & \ddots & \ddots & \dots & \dots \\ l_{m1} & l_{m2} & \dots & & l_{m,m-1} & u_{mm} \end{pmatrix} \tag{2.55}$$

Algorithm: LU factorization by Gaussian elimination (without pivoting), version 2:

```
do $j = 1$ to $m-1$
    ![loop over columns]
    if $a_{jj} = 0$ then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do $i = j+1$ to $m$
        $a_{ij} = a_{ij}/a_{jj}$
        ![create the $l_{ij}$ and stores them in $a_{ij}$]
        do $k = j+1$ to $m$
            $a_{ik} = a_{ik} - a_{ij}a_{jk}$
        enddo
    ![Updates $\mathbf{A}$]
    enddo
enddo
```

□

This algorithm can now directly be implemented in Fortran as is. Also note that because the operations are *exactly the same* as for Gaussian elimination, the operation count is also exactly the same.

7.2. Pivoting for the LU algorithm

Chapter 21 of the textbook

Since LU decomposition and Gaussian elimination are essentially identical – merely different interpretations of the same matrix operations – pivoting is just as important here. However, how do we do it in practice, and how does it affect the LU decomposition? The key is to remember that in the Gaussian elimination algorithm, pivoting swaps rows of the RHS at the same time as it swaps rows of the matrix $\mathbf{A}$. In terms of matrix operations, the pivoted Gaussian elimination algorithm can be thought of as a series of operations

$$\mathbf{M}_{m-1}\mathbf{P}_{m-1}\dots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{A}\mathbf{x} \equiv \mathbf{U}\mathbf{x} = \mathbf{M}_{m-1}\mathbf{P}_{m-1}\dots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{b} \equiv \mathbf{y} \tag{2.56}$$

where the $\mathbf{M}_j$ matrices are defined as earlier, and where the $\mathbf{P}_j$ matrices are permutations matrices. We already saw an example of permutation matrix earlier, but it is worth looking at them in a little more detail now. The matrix $\mathbf{P}_j$ swaps row j with a row $k \geq j$. If $k = j$, then $\mathbf{P}_j$ is simply the identity matrix, but if $k > j$ then $\mathbf{P}_j$ is the identity matrix where the row k and j have been swapped. For instance, a matrix permuting rows 3 and 5 is

$$\mathbf{P} = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 0 & & 1 \\ & & & 1 & \\ & & 1 & & 0 \end{pmatrix} \tag{2.57}$$

By definition, applying $\mathbf{P}_j\mathbf{P}_j\mathbf{A} = \mathbf{A}$ which shows that $\mathbf{P}_j^{-1} = \mathbf{P}_j$. Also, it's easy to verify that the product $\mathbf{A}\mathbf{P}_j$ permutes the *columns* of $\mathbf{A}$ rather than its rows.

Let's now go back to the product $\mathbf{M}_{m-1}\mathbf{P}_{m-1}\dots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1$. By contrast with the unpivoted algorithm, where we were able to show that $\mathbf{M}_{m-1}\dots\mathbf{M}_1$ is a unit lower triangular matrix, it is not clear at all that the same property applies here – in fact, it doesn't! However, what we can show is that

$$\mathbf{M}_{m-1}\mathbf{P}_{m-1}\dots\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 = \mathbf{M}'\mathbf{P} \tag{2.58}$$

where $\mathbf{M}'$ is also a unit lower triangular matrix (different from the unpivoted $\mathbf{M}$) and where $\mathbf{P} = \mathbf{P}_{m-1}\dots\mathbf{P}_2\mathbf{P}_1$ is the product of all the permutation matrices applied by pivoting, which is itself a permutation matrix.

Proof: To show this, consider for simplicity a case where $m = 4$. Let's rewrite it as follows

$$\begin{aligned}
&\mathbf{M}_3\mathbf{P}_3\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 && (2.59)\\
&= \mathbf{M}_3\mathbf{P}_3\mathbf{M}_2(\mathbf{P}_3^{-1}\mathbf{P}_3)\mathbf{P}_2\mathbf{M}_1(\mathbf{P}_2^{-1}\mathbf{P}_3^{-1}\mathbf{P}_3\mathbf{P}_2)\mathbf{P}_1 && (2.60)\\
&= (\mathbf{M}_3)(\mathbf{P}_3\mathbf{M}_2\mathbf{P}_3^{-1})(\mathbf{P}_3\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1})(\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1) && (2.61)\\
&\equiv (\mathbf{M}_3^{'})(\mathbf{M}_2^{'})(\mathbf{M}_1^{'})\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1, && (2.62)
\end{aligned}$$

whereby we can define the $\mathbf{M}_i^{'}$ matrices as

$$\begin{aligned}
\mathbf{M}_3^{'} &= \mathbf{M}_3 && (2.63)\\
\mathbf{M}_2^{'} &= \mathbf{P}_3\mathbf{M}_2\mathbf{P}_3^{-1} && (2.64)\\
\mathbf{M}_1^{'} &= \mathbf{P}_3\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1} && (2.65)
\end{aligned}$$

These matrices look complicated, but in fact they are just equal to $\mathbf{M}_i$ with the *subdiagonal entries* permuted by the pivoting (as opposed to the whole rows permuted). To see why, note that in each expression the permutation matrices operating on $\mathbf{M}_j$ always have an index *greater* than j. Let's look at an example – suppose we consider the product $\mathbf{P}_2\mathbf{M}_1\mathbf{P}_2^{-1}$, and say, for the sake of example, that $\mathbf{P}_2$ swaps rows 2 and 4. Then

$$\mathbf{P}_2 \begin{pmatrix} 1 & & & \\ -l_{21} & 1 & & \\ -l_{31} & & 1 & \\ -l_{41} & & & 1 \end{pmatrix} \mathbf{P}_2^{-1} = \begin{pmatrix} 1 & & & \\ -l_{41} & 0 & & 1 \\ -l_{31} & & 1 & \\ -l_{21} & 1 & & 0 \end{pmatrix} \mathbf{P}_2 = \begin{pmatrix} 1 & & & \\ -l_{41} & 1 & & \\ -l_{31} & & 1 & \\ -l_{21} & & & 1 \end{pmatrix} \tag{2.66}$$

This easily generalizes, so we can see that the matrices $\mathbf{M}'_j$ have exactly the same properties as the matrices $\mathbf{M}_j$, implying for instance that their product $\mathbf{M}' = \mathbf{M}'_3\mathbf{M}'_2\mathbf{M}'_1$ is also unit lower triangular. We have therefore shown, as required, that $\mathbf{M}_3\mathbf{P}_3\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1 = \mathbf{M}'\mathbf{P}$ where $\mathbf{M}'$ is unit lower triangular, and where $\mathbf{P} = \mathbf{P}_3\mathbf{P}_2\mathbf{P}_1$ is a permutation matrix. □

Having shown that the pivoted Gaussian elimination algorithm equivalent to transforming the problem $\mathbf{Ax} = \mathbf{b}$ into

$$\mathbf{M}'\mathbf{PAx} = \mathbf{M}'\mathbf{Pb} \equiv \mathbf{y} \tag{2.67}$$

where $\mathbf{M}'\mathbf{PA}$ is an upper triangular matrix $\mathbf{U}$, we therefore have

$$\mathbf{PA} = \mathbf{LU} \tag{2.68}$$

where this time, $\mathbf{L} = (\mathbf{M}')^{-1}$.

What does this all mean in practice? Well, a few things.

- The first is that we can now create the LU decomposition just as before, but we need make sure to swap the lower diagonal entries of the matrix $\mathbf{L}$ as we progressively construct it. If these are stored in $\mathbf{A}$, as in version 2 of the LU algorithm, this is done trivially!
- Second, solving the problem $\mathbf{Ax} = \mathbf{b}$ is still equivalent to solving $\mathbf{Ux} = \mathbf{y}$ but now $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$. So we need to record the permutation matrix $\mathbf{P}$ to compute $\mathbf{y}$. Note that it is not necessary to save the entire matrix $\mathbf{P}$ to record the permutation – this would be quite wasteful. We can simply record the integer permutation vector $\mathbf{s}$.

These two considerations give us the revised pivoted LU algorithm, together with a corresponding backsubstitution algorithm.

Algorithm: LU factorization by Gaussian elimination (with partial pivoting):

```
![Initialize permutation vector]
do $j = 1$ to $m$
    $s_j = j$
enddo

do $j = 1$ to $m$
    ![loop over columns]
    Find index $K$ and pivot $p$ such that $p = |a_{Kj}| = \max_{k=j,\ldots,m} |a_{kj}|$
    if $K \neq j$ then
        interchange rows $K$ and $j$ of $\mathbf{A}$
        interchange $K$ and $j$ entries of $\mathbf{s}$, $[\mathbf{s}]_j = s_j$
        ![interchange rows and record permutation]
    endif
    if $a_{jj} = 0$ then
        stop
        ![stop if pivot (or divisor) is zero]
    endif
    do $i = j + 1$ to $m$
        $a_{ij} = a_{ij}/a_{jj}$
        ![create the $l_{ij}$ and stores them in $a_{ij}$]
        do $k = j + 1$ to $m$
            $a_{ik} = a_{ik} - a_{ij}a_{jk}$
        enddo
    ![Updates $\mathbf{A}$]
    enddo
enddo
```

Note: this time the loop goes from $j = 1$ to $j = m$. This is because we just need to record the last swapping index.

LU backsubstitution in general (i.e. with or without pivoting) is a little trickier than Gaussian elimination backsubstitution since we first have to create the vector $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$. This is called a forward substitution. The texbook is remarkably silent on the topic, but the algorithm can be worked out reasonably easily if we remember that

- The vector $\mathbf{Pb}$ is just a permutation of the entries of the vector $\mathbf{b}$, and the vector $\mathbf{s}$ was constructed so that $(\mathbf{Pb})_i = b_{s_i}$.
- Although the matrix $\mathbf{L}^{-1}$ is fairly tricky to compute directly, this computation is not actually needed. Indeed, recall that $\mathbf{L}^{-1} = \mathbf{M} = \mathbf{M}_{m-1} \dots \mathbf{M}_1$, and $\mathbf{M}_j = \mathbf{I} - \mathbf{l}_j\mathbf{e}_j^*$.

This implies that we can create $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$ by first letting $\mathbf{y} = \mathbf{Pb}$, and then successively applying the aglorithm $\mathbf{y} = \mathbf{M}_j\mathbf{y}$ where

$$\mathbf{M}_j\mathbf{y} = \mathbf{y} - \mathbf{l}_j\mathbf{e}_j^*\mathbf{y} = \mathbf{y} - y_j\mathbf{l}_j \tag{2.69}$$

which is now easy to express in component form since we know that $\mathbf{l}_j = (0, 0, \dots, 0, l_{j+1,j}, l_{j+2,j}, \dots, l_{mj})^T$. We therefore get

Algorithm: LU backsubstitution:

```
![Initialize $\mathbf{y}$ with $\mathbf{Pb}$]
do $j = 1$ to $m$
    $y_i = b_{s_i}$
enddo

![Forward substitution, $\mathbf{y} = \mathbf{L}^{-1}\mathbf{Pb}$]
do $j = 1$ to $m - 1$
![Do $\mathbf{y} = \mathbf{M}_j\mathbf{y}$]
    do $i = j + 1$ to $m$
        $y_i = y_i - y_j a_{ij}$
    enddo
enddo
```

```
![Backward substitution, $\mathbf{Ux} = \mathbf{y}$]
do $i = m$ to 1
    ![loop over lines, bottom to top]
    if $u_{ii} = 0$ then
        stop
        ![stop if entry is zero, singular matrix]
    endif
    sum = 0.
    do $k = i+1$ to $m$
        sum = sum + $u_{ik}x_k$
    enddo
    $x_i = (y_i -$sum$)/a_{ii}$
enddo
```

8. Cholesky factorization for Hermitian positive definite systems

Chapter 23 of the textbook

Thus far we have assumed that the linear system has a general square nonsingular matrix $\mathbf{A}$, that is otherwise unremarkable, and have learned algorithms for the solution of $\mathbf{Ax} = \mathbf{b}$ that work for any such matrix. However, in some cases we know that the matrix $\mathbf{A}$ has special properties that allow us to use more specialized algorithms, that are often faster, but restricted only to matrices that share these properties.

A well-known example of such algorithms is the Cholesky decomposition, that is only applicable to **positive definite Hermitian matrices**. Such matrices satisfy the property that

$$\mathbf{x}^*\mathbf{Ax} > 0 \tag{2.70}$$

for *any* nonzero vector $\mathbf{x}$.

The fact that $\mathbf{x}^*\mathbf{Ax}$ is real simply comes from the fact that the matrix is Hermitian. Indeed, the complex conjugate of $\mathbf{x}^*\mathbf{Ax}$ is $\mathbf{x}^*\mathbf{A}^*\mathbf{x} = \mathbf{x}^*\mathbf{Ax}$ since $\mathbf{A}^* = \mathbf{A}$. If a number is equal to its complex conjuate, this simply shows it is real. The property $\mathbf{x}^*\mathbf{Ax} > 0$ on the other hand is the defining property of **positive definite** matrices. It means that the action of a Hermitian positive definite matrix on a vector always return a vector whose projection on the original one is positive. Proving that a matrix is positive definite is not always easy, but they often arise in many linear algebra problems that derive from the solution of a PDE, for instance, and as such, are very common.

Note that if you have a **negative definite** matrix, i.e. a matrix with the property that

$$\mathbf{x}^*\mathbf{Ax} < 0 \tag{2.71}$$

for *any* $\mathbf{x}$, then it is very easy to create the positive definite matrix $\mathbf{B} = -\mathbf{A}$. Non positive definite or negative definite matrices are matrices for which $\mathbf{x}^*\mathbf{A}\mathbf{x}$ is sometimes positive, and sometimes negative, depending on the input vector $\mathbf{x}$.

If the matrix $\mathbf{A}$ is symmetric and positive definite (SPD), then an LU decomposition of $\mathbf{A}$ indicates that $\mathbf{A} = \mathbf{L}\mathbf{U}$. This then implies that

$$\mathbf{A}^* = (\mathbf{L}\mathbf{U})^* = \mathbf{U}^*\mathbf{L}^* = \mathbf{A} = \mathbf{L}\mathbf{U} \tag{2.72}$$

so we have just shown that the LU decomposition of a Hermitian matrix satisfies

$$\mathbf{U}^*\mathbf{L}^* = \mathbf{L}\mathbf{U} \tag{2.73}$$

Since the transpose of an upper-triangular matrix is a lower triangular one, and vice versa, this also shows that one should in principle be able to find a decomposition such that $\mathbf{L} = \mathbf{U}^*$, or $\mathbf{U} = \mathbf{L}^*$ so that

$$\mathbf{A} = \mathbf{L}\mathbf{U} = \mathbf{U}^*\mathbf{U} = \mathbf{L}\mathbf{L}^* \tag{2.74}$$

for any Hermitian matrix $\mathbf{A}$. In practice, this factorization only exists for positive definite matrices. To see why, note that

$$\mathbf{x}^*\mathbf{A}\mathbf{x} = \mathbf{x}^*\mathbf{U}^*\mathbf{U}\mathbf{x} = (\mathbf{U}\mathbf{x})^*(\mathbf{U}\mathbf{x}) > 0 \tag{2.75}$$

so the existence of a decomposition $\mathbf{A} = \mathbf{U}^*\mathbf{U}$ (or equivalently $\mathbf{L}\mathbf{L}^*$) does indeed depend on the positive definiteness of $\mathbf{A}$.

This decomposition is known as the *Cholesky factorization* of $\mathbf{A}$:

Definition: A *Cholesky factorization* of a Hermitian positive definite matrix $\mathbf{A}$ is given by $A = \mathbf{U}^*\mathbf{U} = \mathbf{L}\mathbf{L}^*$, where $\mathbf{U}$ is upper-triangular and $\mathbf{L}$ is lower triangular such that $u_{jj} = l_{jj} > 0$.

To prove that it exists for any positive definite Hermitian matrix, we simply need to come up with an algorithm to systematically create it. Let us consider the decomposition $\mathbf{A} = \mathbf{L}\mathbf{L}^*$, and write it out in component form:

$$a_{ij} = \sum_{k=1}^{m} (\mathbf{L})_{ik}(\mathbf{L})^*_{kj} = \sum_{k=1}^{m} l_{ik} l^*_{jk} = \sum_{k=1}^{\min(i,j)} l_{ik} l^*_{jk} \tag{2.76}$$

using successively, the definition of the Hermitian transpose, and the fact that $\mathbf{L}$ is lower triangular so its coefficients are zero if the column number is larger than the row number. Writing these out for increasing values of i we have

- $i = 1$:
$$a_{11} = l_{11} l^*_{11} \tag{2.77}$$

- $i = 2$:
$$\begin{aligned} a_{21} &= l_{21} l^*_{11} \\ a_{22} &= l_{21} l^*_{21} + l_{22} l^*_{22} \end{aligned} \tag{2.78}$$

- $i = 3$:

$$\begin{aligned} a_{31} &= l_{31} l_{11}^* \\ a_{32} &= l_{31} l_{21}^* + l_{32} l_{22}^* \\ a_{33} &= l_{31} l_{31}^* + l_{32} l_{32}^* + l_{33} l_{33}^* \end{aligned} \tag{2.79}$$

and so forth. Note that because the matrix is Hermitian, $a_{11} = a_{11}^*$ and so a_{11} must be real, implying that the equation $a_{11} = l_{11} l_{11}^* = ||l_{11}||^2$ indeed makes sense.

Let's now choose $l_{11} = \sqrt{a_{11}}$. We can then move to the next equation, and write successively

$$\begin{aligned} l_{21} &= \frac{a_{21}}{l_{11}} \\ l_{22} = l_{22}^* &= \sqrt{a_{22} - l_{21} l_{21}^*} \end{aligned} \tag{2.80}$$

then

$$\begin{aligned} l_{31} &= \frac{a_{31}}{l_{11}} \\ l_{32} &= \frac{a_{32} - l_{31} l_{21}^*}{l_{22}} \\ l_{33} = l_{33}^* &= \sqrt{a_{33} - l_{31} l_{31}^* - l_{32} l_{32}^*} \end{aligned} \tag{2.81}$$

etc...

This can always be done, which provides us with a straightforward algorithm to construct the Cholesky factorization. Note that many version of this algorithm exist, which merely reorder some of these operations in different ways (cf textbook, Numerical Recipes, etc). They appear to be roughly equivalent in terms of time and stability. The one presented below is reasonably simple to understand, and merely corresponds to working column by column, first calculating the diagonal element in each column, that is, calculating

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk} l_{jk}^*} \tag{2.82}$$

then working on the column below that element, in which

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}^*}{l_{jj}} \tag{2.83}$$

The elements of **L** are simply stored by over-writing the lower triangular elements of **A**.

Algorithm: Cholesky factorization (decomposition):

```
![loop over columns]
```

```
do $j = 1$ to $m$
    ![Calculate new diagonal element]
    do $k = 1$ to $j - 1$
        $a_{jj} = a_{jj} - a_{jk}a^*_{jk}$
    enddo
    $a_{jj} = \sqrt{a_{jj}}$
    ![Calculate elements below diagonal]
    do $i = j + 1$ to $m$
        do $k = 1$ to $j - 1$
            $a_{ij} = a_{ij} - a_{ik}a^*_{jk}$
        enddo
        $a_{ij} = a_{ij}/a_{jj}$
    enddo
enddo
```

Note: There is a number of facts about the CF algorithm that make it very attractive and popular for symmetric positive definite matrices:

- No pivoting is required for numerical stability.

- Only about $n^3/6$ multiplications and a similar number of additions are required, which makes it about twice as fast as the standard Gaussian elimination.

- The construction requires calculating the square roots of m numbers. If the matrix $\mathbf{A}$ is Hermitian positive definite, these numbers are all positive so the square root is well-defined. On the other hand, if one of these is not positive, then this means the original matrix was not positive definite – this is often used as a test to determine the positive-definiteness of a matrix.

As in the case of LU decomposition, the Cholesky decomposition stores a partially inverted matrix $\mathbf{A}$ (so to speak), so solutions to the problem $\mathbf{Ax} = \mathbf{b}$ can be obtained quickly for different RHS $\mathbf{b}$ as they become known. To do so, we need to solve the problem $\mathbf{LL}^*\mathbf{x} = \mathbf{b}$, which can be decomposed into two steps:

- a forward substitution step of the form $\mathbf{Ly} = \mathbf{b}$, which gives the vector $\mathbf{y}$

- a backsubstitution step of the form $\mathbf{L}^*\mathbf{x} = \mathbf{y}$, which gives the vector $\mathbf{x}$.

Algorithm: Cholesky backsubstitution :

```
![Forward substitution, solving $\mathbf{Ly} = \mathbf{b}$
do $i = 1$ to $m$
    sum = $b_i$
    do $j = 1$ to $i - 1$
        sum = sum $-y_j l_{ij}$
    enddo
    $y_i =$ sum/$l_{ii}$
enddo

![Backward substitution, solving $\mathbf{L}^*\mathbf{x}=\mathbf{y}$
do $i=m$ to 1
    if $l^*_{ii}=0$ then
        stop
    endif
    do $k=i+1$ to $m$
        $y_i = y_i - l^*_{ki}x_k$
    enddo
    $x_i = y_i/l^*_{ii}$
enddo
```

Note that this algorithm preserves **b** and returns a separate solution **x**. It is also possible to write it in such a way as to overwrite **b** with the solution on exit.

9. Some notions of stability for numerical linear algebra

Chapters 14 and 15 from the textbook

In this section, we now introduce the subtle but very important concepts of accuracy and stability in numerical linear algebra, where any algorithm put forward needs to be examined in the light of the potential errors arising from floating point arithmetic.

Let's first introduce some notations. An algorithm in linear algebra usually takes a quantity such as scalar, vector or a matrix, denoted X, and applies to it a series of functions or transformations to arrive at another quantity, scalar, vector, or matrix, denoted Y. In theory, this algorithm is exact, but its numerical implementation is not because of rounding errors and floating point arithmetic. Let's call f the exact theoretical algorithm (so $Y = f(X)$), and $\tilde{f}$ the approximate one. Let's also call $\tilde{X}$ the numerically-approximated input value (which may contain roundoff errors), and $\tilde{Y}$ is the numerically-approximated value of the exact solution (assuming the latter is somehow known).

Definition: A round-off error is defined by the difference between an exact and an approximated solutions.

We know that the relative roundoff errors are smaller or equal to machine accuracy, so by definition

$$\frac{||\tilde{X} - X||}{||X||} < \epsilon_{\text{mach}}, \tag{2.84}$$

$$\frac{||\tilde{Y} - Y||}{||Y||} < \epsilon_{\text{mach}} \tag{2.85}$$

However, except for the most trivial algorithms, there is no guarantee that

$$\tilde{f}(\tilde{X}) = \tilde{Y} \tag{2.86}$$

which means that, even though (2.85) is true, there is no guarantee that

$$\frac{||\tilde{f}(\tilde{X}) - f(X)||}{||f(X)||} < \epsilon_{\text{mach}} \tag{2.87}$$

should be. In fact, this is very rarely true! So the question is, can we nevertheless estimate under which circumstances the numerical solution $\tilde{f}(\tilde{X})$ is indeed close to $f(X)$?

Definition: A numerical algorithm $\tilde{f}$ for a problem f is called **accurate** if, for each possible input value X, we have

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = O(\epsilon_{\text{mach}}) \tag{2.88}$$

for small enough ϵ_{mach}.

The symbol O is a notation that means *of order of*, and has a very strict mathematical definition which is explored in detail in AMS212B and AMS213B. For the purpose of this class, it is sufficient to interpret this to mean that there exists a positive constant C such that

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} < C\epsilon_{\text{mach}} \tag{2.89}$$

for small enough ϵ_{mach} and for all X. Note that if $||f(X)|| \to 0$, this can and should be re-interpreted as $||\tilde{f}(X) - f(X)|| < C\epsilon_{\text{mach}}||f(X)||$. In words, this implies that $\tilde{f}(X) - f(X)$ decays to zero *at least as fast* as $f(X)$ does. Note that nothing in this definition requires C to be of order unity; as we shall see, sometimes C can be very large, so the concept of accuracy takes a different meaning in the strict mathematical sense and in the everyday-life sense.

Accuracy is usually quite difficult to prove directly (though see later for an important theorem on the topic). In fact, algorithms applied to input matrices

X that are ill-conditioned cannot be accurate. For this reason, another more practical concept is that of stability.

Definition: An algorithm $\tilde{f}$ for a problem f is **stable** if for each possible input X,

$$\frac{||\tilde{f}(X) - f(\tilde{X})||}{||f(\tilde{X})||} = O(\epsilon_{\text{mach}}) \tag{2.90}$$

$$\text{for some } \tilde{X} \text{ with } \frac{||\tilde{X} - X||}{||X||} = O(\epsilon_{\text{mach}}) \tag{2.91}$$

As described in the book, such *a stable numerical algorithm gives nearly the right answer for nearly the right input.*

Example: The unpivoted algorithm for Gaussian elimination is a clear example of an **unstable** algorithm. In the case of the input matrix

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{2.92}$$

there is no nearby value $\mathbf{X}'$ that would let the algorithm return *nearly the right answer.* Indeed, as long as the 0 is there in the diagonal, the algorithm fails because of division by zero, and even if we try to consider instead the matrix

$$\mathbf{X}' = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \tag{2.93}$$

with $\epsilon = O(\epsilon_{\text{mach}})$, to avoid the problem, we saw that the answer will still be $O(1)$ away from the true answer.

Directly proving the stability of an algorithm can be quite difficult but there is another concept that is both stronger (i.e. that implies stability), and often simpler to prove, that of **backward stability.**

Definition: An algorithm $\tilde{f}$ for a problem f is **backward stable** if, for each possible input X,

$$\tilde{f}(X) = f(\tilde{X}) \text{ for some } \tilde{X} \text{ with } \frac{||\tilde{X} - X||}{||X||} = O(\epsilon_{\text{mach}}). \tag{2.94}$$

To paraphrase the textbook again, *a backward stable numerical algorithm gives exactly the right answer for nearly the right input.* Note that backward stability trivially implies stability, but the converse is not true (stable algorithms are not all backward stable). Also note that, because all norms are equivalent within a factor unity, the definitions of stable and backward stable hold regardless of the norm used – hence we can prove stability or backward stability using whichever one is the easiest to use for any given algorithm.

Backward stability is often easier to demonstrate than stability because it is relatively easy to show using some of the fundamental properties of floating point arithmetic that any of the four basic operations $+$, $-$, $\times$ and $\div$ are all backward stable – namely, given any input vector $(x, y)^T$, the numerical algorithms that compute respectively $x + y$, $x - y$, xy and x/y are backward stable (as long as the quantity is defined). For details, examples, and more, see the textbook. In particular, read the section on the proof of backward stability for the backsubstitution algorithm (Chapter 17), which should illustrate exactly how tricky and detailed stability analysis needs to be in order to be rigorous.

Because stability/instability proofs are so cumbersome, we will never attempt to prove stability in this course, but will merely use well-known results from the literature, namely:

- The backsubstitution algorithm for Gaussian elimination is backward stable.
- Gaussian elimination or LU factorization without pivoting is neither backward stable nor stable.
- Gaussian elimination or LU factorization with partial pivoting is *theoretically* backward stable, but can be prone to extremely large errors for large, poorly conditioned matrices (see Chapter 22 of the textbook and examples).
- Cholesky factorization and backsubstitution are backward stable.

So what can be said about accuracy, which was the original question posed in this section? As it turns out, the accuracy of an algorithm can be estimated as long as the algorithm has already been shown to be backward stable, with the following theorem:

Theorem: Suppose a backward stable algorithm $\tilde{f}$ is applied to solve a problem $f(X) = Y$ with a relative condition number $\kappa(X)$ that is known. Then the relative errors satisfy

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = O(\kappa(X)\epsilon_{\text{mach}}) \tag{2.95}$$

To prove the theorem, notice that since $\tilde{f}$ is backward stable, $\tilde{f}(X) = f(\tilde{X})$ for some $\tilde{X}$ such that

$$\frac{||\delta X||}{||X||} = O(\epsilon_{\text{mach}}), \tag{2.96}$$

where $\delta X = \tilde{X} - X$. If we let $\delta f = f(\delta X)$, the relative errors can be written as

$$\frac{||\tilde{f}(X) - f(X)||}{||f(X)||} = \frac{||f(\tilde{X}) - f(X)||}{||f(X)||} = \frac{||\delta f||}{||f(X)||} \tag{2.97}$$

as long as Eq.(2.96) is true. Using the definition of $\kappa(X)$, we get

$$\frac{||\delta f||}{||f(X)||}\frac{||X||}{||\delta X||} \leq \kappa(X), \tag{2.98}$$

which yields, together with Eq. (2.96),

$$\frac{||\delta f||}{||f(X)||} \leq \kappa(X)\frac{||\delta X||}{||X||} \leq C\kappa(X)\epsilon_{\text{mach}}, \tag{2.99}$$

for some $C > 0$. This prove the theorem. □

Hence, as long as $\kappa(X)$ is bounded for all X, the theorem guarantees accuracy of the algorithm, and provides upper bounds on the relative error. If $\kappa(X)$ is not globally bounded, the theorem can nevertheless be used to estimate the accuracy of the algorithm for a particular input X. This shows once again that, regardless of the quality of an algorithm, little can be done to get an accurate answer if the original problem itself is ill-conditioned.

Finally, we saw that for both matrix multiplication $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ and for the solution of the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ (i.e. $f(\mathbf{b}) = \mathbf{A}^{-1}\mathbf{b}$, given $\mathbf{b}$), the relative condition number of the problem was bounded by the condition number of the matrix $\mathbf{A}$. This then implies that the accuracy of a backward stable algorithm can be estimated with

$$\frac{||\tilde{f}(\tilde{X}) - f(X)||}{||f(X)||} \leq O(\kappa(\mathbf{A})\epsilon_{\text{mach}}) \tag{2.100}$$

This bound therefore applies to any matrix multiplication $\mathbf{A}\mathbf{x} = \mathbf{b}$ to given an estimate of the error on $\mathbf{b}$, and to applications of LU factorization and Gaussian elimination for the solution of linear systems, as well as to the use of the Cholesky factorization for the same purpose, to give an estimate of the error on the solution $\mathbf{x}$.

Chapter 3

Solutions of overdetermined linear problems (Least Square problems)

1. Over-constrained problems

See Chapter 11 from the textbook

1.1. Definition

In the previous chapter, we focused on solving well-defined linear problems defined by m linear equations for m unknowns, put into a compact matrix-vector form $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ an $m \times m$ square matrix, and $\mathbf{b}$ and $\mathbf{x}$ $m-$long column vectors. We focussed on using *direct methods* to seek *exact* solutions to such well-defined linear systems, which exist whenever $\mathbf{A}$ is nonsingular. We will revisit these problems later this quarter when we learn about iterative methods.

In this chapter, we look at a more general class of problems defined by so-called **overdetermined** systems – systems with a larger numbers of equations (m) than unknowns (n): this time, we have $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ an $m \times n$ matrix with $m > n$, $\mathbf{x}$ an n-long column vector and $\mathbf{b}$ an $m-$long column vector. This time there generally are no exact solutions to the problem. Rather we now want to find *approximate solutions* to overdetermined linear systems which minimize the residual error

$$E = ||\mathbf{r}|| = ||\mathbf{b} - \mathbf{Ax}|| \tag{3.1}$$

using some norm. The vector $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ is called the **residual** vector.

Any choice of norm would generally work, although, in practice, we prefer to use the Euclidean norm (i.e., the 2-norm) which is more convenient for numerical purposes, as they provide well-established relationships with the inner product and orthogonality, as well as its smoothness and convexity (we shall see later). For this reason, the numerical solution of overdetermined systems is usually called the **Least Square solution**, since it minimizes the sum of the square of the coefficients of the residual vector, $r_i = (\mathbf{b} - \mathbf{Ax})_i$ for $i = 1 \dots m$.

Remark: You will sometimes find references to least squares problems as

$$\mathbf{Ax} \cong \mathbf{b} \tag{3.2}$$

in order to explicitly reflect the fact that $\mathbf{x}$ is *not* the exact solution to the overdetermined system, but rather is an approximate solution that minimizes the Euclidean norm of the residual. □

1.2. Overdetermined System

The question that naturally arises is then "what makes us to consider an overdetermined system?" Let us consider some possible situations.

Example: Suppose we want to know monthly temperature distribution in Santa Cruz. We probably would not make one single measurement for each month and consider it done. Instead, we would need to take temperature readings over many years and average them. From this procedure, what we are going to make is a table of 'typical' temperatures from January to December, based on vast observational data, which often times even include unusual temperature readings that deviate from the 'usual' temperature distributions. This example illustrates a typical overdetermined system: we need to determine one representative meaningful temperature for each month (i.e., one unknown temperature for each month) based on many numbers (thus overdetermined) of collected sample data including sample noises (due to measurement failures/errors, or unusually cold or hot days – data deviations, etc.). □

Example: Early development of the method of least squares was due largely to Gauss, who used it for solving problems in astronomy, particularly determining the orbits of celestial bodies such as asteroids and comets. The least squares method was used to smooth out any observational errors and to obtain more accurate values for the orbital parameters. □

Example: A land surveyor is to determine the heights of three hills above some reference point. Sighting from the reference point, the surveyor measures their respective heights to be

$$h_1 = 1237ft., h_2 = 1942ft., \text{ and } h_3 = 2417ft. \tag{3.3}$$

And the surveyor makes another set of relative measurements of heights

$$h_2 - h_1 = 711ft., h_3 - h_1 = 1177ft., \text{ and } h_3 - h_2 = 475ft. \tag{3.4}$$

The surveyor's observation can be written as

$$\mathbf{Ax} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \cong \begin{bmatrix} 1237 \\ 1941 \\ 2417 \\ 711 \\ 1177 \\ 475 \end{bmatrix} = \mathbf{b}. \tag{3.5}$$

It turns out that the approximate solution becomes (we will learn how to solve this soon)

$$\mathbf{x}^T = [h_1, h_2, h_3] = [1236, 1943, 2416], \tag{3.6}$$

which differ slightly from the three initial height measurements, representing a compromise that best reconciles the inconsistencies resulting from measurement errors. □

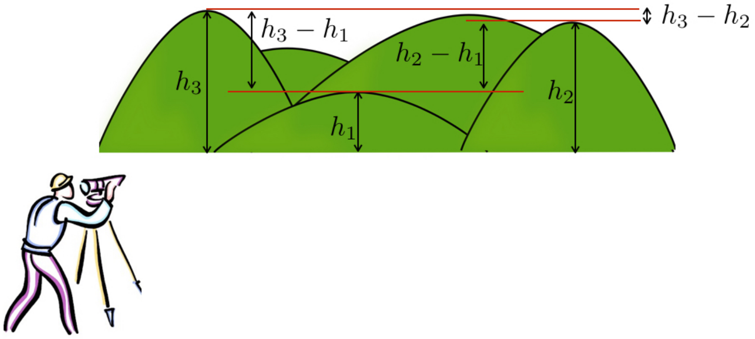


Figure 1. A math-genius land surveyor obtains three hills' height by computing the least squares solution to the overdetermined linear system.

1.3. Examples of applications of overdetermined systems

One of the most common standard applications that give rise to an overdetermined system is that of *data fitting*, or *curve fitting*. The problem can be illustrated quite simply for any function from $\mathbb{R}$ to $\mathbb{R}$ as

- Given m data points $(x_i, y_i), i = 1, \ldots, m$
- Given a function $y = f(x; \mathbf{a})$ where $\mathbf{a} = (a_1, \cdots, a_n)^T$ is a vector of n unknown parameters
- The goal is to find for which vector $\mathbf{a}$ the function f best fits the data in the least square sense, i.e. that minimizes

$$E^2 = \sum_{i=1}^{m} \Big(y_i - f(x_i, \mathbf{a}) \Big)^2. \tag{3.7}$$

The problem can be quite difficult to solve when f is a nonlinear function of the unknown parameters. However, if f uses a linear combination of the coefficients a_i this problem simply reduces to an overdetermined linear problem. These are called **linear data fitting** problems.

Definition: A data fitting problem is **linear** if f is linear in $\mathbf{a} = [a_1, \cdots, a_n]^T$, although f could be nonlinear in x.

Note that the example above can also easily be generalized to multivariate functions. Let's see a few examples of linear fitting problems.

1.3.1. Linear Regression Fitting a linear function through a set of points is a prime example of linear regression.

In the single-variable case, this requires finding the coefficients a and b of the linear function $f(x) = ax+b$ that best fits a set of data points $(x_i, y_i), i = 1, \dots, m$. This requires solving the overdetermined system

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \\ x_m & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{3.8}$$

For multivariate problems, we may for instance want to fit the linear function $y = f(\mathbf{x}) = a_0 + a_1 x_1 + \cdots + a_{n-1} x_{n-1}$ through the m data points with coordinates $(x_1^{(i)}, x_2^{(i)}, \dots, x_{n-1}^{(i)}, y^{(i)})$ for $i = 1, \dots, m$. In that case, we need to solve the overdetermined system

$$\begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_{n-1}^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_{n-1}^{(2)} \\ \vdots & & & & \\ 1 & x_1^{(m)} & x_2^{(m)} & \dots & x_{n-1}^{(m)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \tag{3.9}$$

for the n parameters $a_0, \dots, a_{n-1}$.

1.3.2. Polynomial fitting Polynomial fitting, in which we try to fit a polynomial function

$$f(x; \mathbf{a}) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \tag{3.10}$$

to a set of m points (x_i, y_i) is a also a linear data fitting problem (since f depends linearly on the coefficients of $\mathbf{a}$). Finding the best fit involves solving the overconstrained problem

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & & & & \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{3.11}$$

for the n parameters $a_0, \dots, a_{n-1}$. The matrix formed in the process is of a particularly well-known type called a **Vandermonde matrix**.

Example: Consider five given data points, $(t_i, y_i), 1 \le i \le 5$, and a data fitting using a quadratic polynomial. This overdetermined system can be written as, using a Vandermonde matrix,

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cong \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \mathbf{b}. \tag{3.12}$$

The problem is to find the best possible values of $\mathbf{x} = [x_1, x_2, x_3]^T$ which minimizes the residual $\mathbf{r}$ in l^2-sense:

$$||\mathbf{r}||_2^2 = ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 = \sum_{i=1}^{5} \Big(y_i - (x_1 + x_2 t_i + x_3 t_i^2) \Big)^2 \tag{3.13}$$

Such an approximating quadratic polynomial is plotted as a smooth curve in blue in Fig. 2, together with m given data points denoted as red dots. □

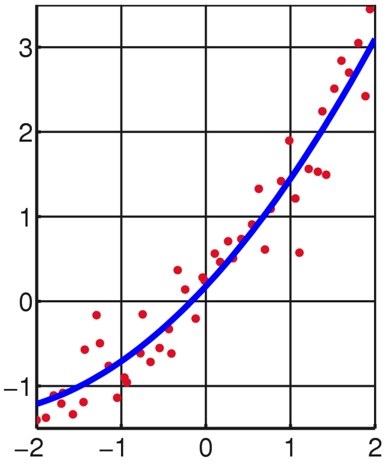


Figure 2. The result of fitting a set of data points (t_i, y_i) with a quadratic function, $f(x, \mathbf{a}) = a_0 + a_1 x + a_2 x^2$. Image source: Wikipedia

Remark: In statistics, the method of least squares is also known as *regression analysis.* □

1.3.3. A nonlinear fitting problem: Fitting a function of the kind

$$f(t, \mathbf{a}) = a_1 e^{a_2 t} + a_2 e^{a_3 t} + \cdots + a_{n-1} e^{a_n t} \tag{3.14}$$

is not a linear data fitting problem. This must be treated using **nonlinear least square** methods, beyond the scope of this course.

Note that in all the examples above, we were able to frame the problem in the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the vector of real parameters (called $\mathbf{a}$ above) of size n, $\mathbf{A}$ is a real matrix, of size $m \times n$ and $\mathbf{b}$ is a real vector of size m.

2. Solution of Least Squares Problems using the Cholesky decomposition

As discussed earlier, a Least Squares problem can be viewed as a minimization problem on the norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ over all possible values of $\mathbf{x}$. This is essentially a problem in multivariate calculus! To solve it, first note that minimizing the norm or its square is the same thing. Then we write the square of the Euclidean norm of $\mathbf{r}$ as an inner product:

$$E^2 = ||\mathbf{r}||^2 = \mathbf{r}^T\mathbf{r} = (\mathbf{b} - \mathbf{Ax})^T(\mathbf{b} - \mathbf{Ax}) \tag{3.15}$$

Expanding this, we get

$$\begin{aligned} E^2 &= \mathbf{b}^T\mathbf{b} - (\mathbf{Ax})^T\mathbf{b} - \mathbf{b}^T(\mathbf{Ax}) + (\mathbf{Ax})^T(\mathbf{Ax}) && (3.16)\\ &= \mathbf{b}^T\mathbf{b} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} - \mathbf{b}^T\mathbf{Ax} + \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} && (3.17) \end{aligned}$$

Minimizing the residual implies we need to find the solution $\mathbf{x}$ that satisfies

$$\frac{\partial E^2}{\partial x_i} = 0, \text{ for } i = 1, \dots, n. \tag{3.18}$$

Writing E^2 in component form, we have

$$\begin{aligned} E^2 &= \sum_i b_i^2 - \sum_{i,j} x_i a_{ji} b_j - \sum_{i,j} b_i a_{ij} x_j + \sum_{ijk} x_i a_{ji} a_{jk} x_k && (3.19)\\ &= \sum_i b_i^2 - 2\sum_{i,j} x_i a_{ji} b_j + \sum_{ijk} x_i a_{ji} a_{jk} x_k && (3.20) \end{aligned}$$

It is fairly easy to verify that

$$\frac{\partial E^2}{\partial x_i} = -2\sum_j a_{ji} b_j + 2\sum_{jk} a_{ji} a_{jk} x_k = 2(\mathbf{A}^T\mathbf{Ax})_i - 2(\mathbf{A}^T\mathbf{b})_i \tag{3.21}$$

so setting this to zero for all i requires

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}. \tag{3.22}$$

These new equations are usually referred to as the **normal equations** associated with the overdetermined problem $\mathbf{Ax} = \mathbf{b}$. The reason for this nomenclature will be clarified shortly.

This time, since $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{A}^T\mathbf{A}$ is a square $n \times n$ matrix which is usually much smaller than the original matrix $\mathbf{A}$. Similarly, $\mathbf{A}^T\mathbf{b}$ is now a

$n-$long vector, much smaller than the original $\mathbf{b}$. In addition, it is easy to show that as long as $\mathbf{A}$ is full rank (i.e. rank n), $\mathbf{A}^T\mathbf{A}$ is positive definite. Indeed, let $\tilde{\mathbf{A}} = \mathbf{A}^T\mathbf{A}$, and let's compute

$$\mathbf{y}^T\tilde{\mathbf{A}}\mathbf{y} = \mathbf{y}^T\mathbf{A}^T\mathbf{A}\mathbf{y} = (\mathbf{A}\mathbf{y})^T(\mathbf{A}\mathbf{y}) = ||\mathbf{A}\mathbf{y}||^2 \tag{3.23}$$

which is always greater than zero for any input vector $\mathbf{y}$ (unless $\mathbf{y} = 0$) since $\mathbf{A}$ is assumed to be full rank.

Positive definiteness is good news because it means that we can use the Cholesky decomposition to solve the problem. Since we saw that the Cholesky decomposition is stable, we therefore know that we can stably find solutions to Least Square problems using the following steps. Given the overconstrained problem $\mathbf{A}\mathbf{x} = \mathbf{b}$,

- Form the normal equation $\tilde{\mathbf{A}}\mathbf{x} = \tilde{\mathbf{b}}$ where $\tilde{\mathbf{A}} = \mathbf{A}^T\mathbf{A}$ and $\tilde{\mathbf{b}} = \mathbf{A}^T\mathbf{b}$.
- Solve the normal equation using the Cholesky decompositon and backsubstitution, for the unknown vector $\mathbf{x}$.

and voila! Examples of application of this method for data fitting will be explored in Homework 3.

Among all the methods for the solution on unconstrained linear systems (see later for other ones) this is definitely the fastest. However, it is also the least accurate of all methods (even though it is stable). That's because

$$\text{cond}(\mathbf{A}^T\mathbf{A}) = \text{cond}(\mathbf{A})^2, \tag{3.24}$$

so if $\mathbf{A}$ is relatively poorly conditioned (large cond($\mathbf{A}$)) then $\tilde{\mathbf{A}} = \mathbf{A}^T\mathbf{A}$ is *very* poorly conditioned, and from a numerical point of view can even be singular even if the original problem is not.

Example: Consider

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \epsilon & 0 \\ 0 & \epsilon \end{bmatrix}, \tag{3.25}$$

where $0 < \epsilon < \sqrt{\epsilon_{\text{mach}}}$. Forming $\tilde{\mathbf{A}}$ we get

$$\tilde{\mathbf{A}} = \mathbf{A}^T\mathbf{A} = \begin{bmatrix} 1+\epsilon^2 & 1 \\ 1 & 1+\epsilon^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{3.26}$$

in floating point arightmetic, which is a singular matrix. □

Even though this example is rather extreme, poor conditioning is unfortunately very common in data sets that involve points with vastly varying values of x, *especially* for polynomial fitting and their associated Vandermonde matrices. For this reason, other methods were later developed that do not involve forming the normal equations in the first place, but instead, working directly with the original matrix $\mathbf{A}$.

3. Towards better algorithms for Least Square problems

3.1. A geometric interpretation of the Least Square Problem

To see how one may go about constructing such methods, let's look back at the normal equations and re-write them as

$$\mathbf{A}^T(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0 \Leftrightarrow \mathbf{A}^T\mathbf{r} = 0 \tag{3.27}$$

which is also equivalent to saying that

$$\mathbf{a}_i^T\mathbf{r} = 0 \text{ for all } i = 1, \dots, n, \tag{3.28}$$

where $\mathbf{a}_i$ are column vectors of $\mathbf{A}$ (or $\mathbf{a}_i^T$ are row vectors of $\mathbf{A}^T$). In other words, the normal equations simply state that the solution that minimizes the error E is the one for which the residual $\mathbf{r}$ is orthogonal to the column vectors of $\mathbf{A}$.

While this may seem to be just an odd coincidence at first, of course it isn't. This result actually has a very simple geometric interpretation. Consider indeed that, in solving the original problem $\mathbf{A}\mathbf{x} \cong \mathbf{b}$, the vector $\mathbf{b}$ has dimension m, while the range of the matrix $\mathbf{A}$ only has dimension $n < m$. The vector $\mathbf{y} = \mathbf{A}\mathbf{x}$, by definition, is a linear combination of the column vectors of $\mathbf{A}$, and therefore lies in the span of $\mathbf{A}$. But in general $\mathbf{b}$ does not lie in span($\mathbf{A}$) because its dimension is m and is larger than the dimension of span($\mathbf{A}$). We therefore have the situation illustrated in the figure below. In this figure, we see that the vector $\mathbf{r}$ joins the tip of $\mathbf{b}$ to the tip of $\mathbf{y}$, and so it is shortest when $\mathbf{r}$ is perpendicular to the plane spanned by $\mathbf{A}$. This picture can be expanded to more than 3 dimensions, and shows that to find the vector $\mathbf{x}$ for which $||\mathbf{r}|| = ||\mathbf{b} - \mathbf{A}\mathbf{x}||$ is minimal, we simply have to ensure that $\mathbf{r}$ is orthogonal to the span of $\mathbf{A}$, which requires that it must be orthogonal to every single column vector of $\mathbf{A}$.

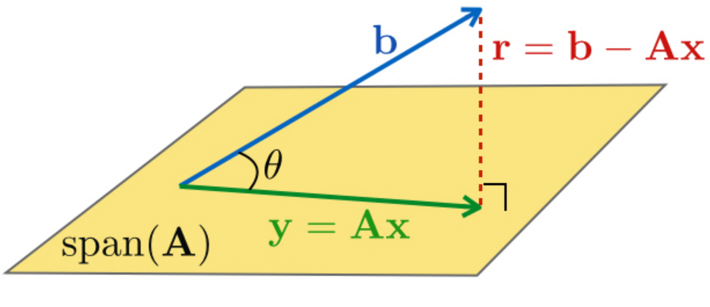

$\mathbf{b}$
$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
θ
$\mathbf{y} = \mathbf{A}\mathbf{x}$
span($\mathbf{A}$)

Figure 3. Geometric interpretation of linear squares problem.

More importantly, this diagram also illustrates that finding the vector $\mathbf{A}\mathbf{x}$ for which $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ is orthogonal to the span of $\mathbf{A}$ is equivalent to finding the

orthogonal projection of $\mathbf{b}$ *onto* the span of $\mathbf{A}$. If that projection operator, $\mathbf{P}_A$, is known, then we just have to solve

$$\mathbf{A}\mathbf{x} = \mathbf{P}_A\mathbf{b} \tag{3.29}$$

to find $\mathbf{x}$. Note that this time, $\mathbf{A}\mathbf{x}$ and $\mathbf{P}_A\mathbf{b}$ are both vectors of length equal to the rank of $\mathbf{A}$ (n, if $\mathbf{A}$ is full rank) so this equation forms an exact linear system. This idea leads to a new method of solution of Least Square problems, namely that of orthogonal projection. To construct it we first need to make a little detour to learn about orthogonal projectors.

3.2. Orthogonal Projectors

See Chapter 6 of the textbook

Definition: A square matrix $\mathbf{P}$ (of size $m \times m$) is said to be a **projector** if it is **idempotent**, that is,

$$\mathbf{P}^2 = \mathbf{P}. \tag{3.30}$$

Definition: If a projector $\mathbf{P}$ is also symmetric, $\mathbf{P}^T = \mathbf{P}$, then it is called an **orthogonal projector**. Note that an orthogonal projector is not necessarily an orthogonal matrix (this can be confusing!)

Orthogonal projectors behave exactly as you would imagine from their names: they project vectors onto some subspace, orthogonally to that subspace. And the subspace in question is simply the space spanned by the column vectors of $\mathbf{P}$. To see all of this, first note that because $\mathbf{P}^2 = \mathbf{P}$, then

$$\mathbf{P}\mathbf{p}_i = \mathbf{p}_i \text{ for } i = 1, \ldots, k \tag{3.31}$$

where $\mathbf{p}_i$ are the column vectors of $\mathbf{P}$, and $\text{rank}(\mathbf{P}) = k \leq m$. Hence, any vector $\mathbf{x}$ that lies in the subspace spanned by the columns of $\mathbf{P}$, which can be written as $\mathbf{x} = \sum_i \alpha_i \mathbf{p}_i$, is invariant by application of $\mathbf{P}$ since $\mathbf{P}\mathbf{x} = \sum_i \alpha_i \mathbf{P}\mathbf{p}_i = \sum_i \alpha_i \mathbf{p}_i = \mathbf{x}$. Meanwhile, suppose instead that we have a vector $\mathbf{x}$ that is *orthogonal* to the span of $\mathbf{P}$, then by definition we have $\mathbf{x}^T\mathbf{p}_i = \mathbf{p}_i^T\mathbf{x} = 0$ for all i. If we now calculate $\mathbf{P}\mathbf{x}$ we find that

$$(\mathbf{P}\mathbf{x})_i = (\mathbf{P}^T\mathbf{x})_i = \mathbf{p}_i^T\mathbf{x} = 0 \text{ for all } i = 1, \ldots, k \tag{3.32}$$

as required. Hence, the projector $\mathbf{P}$ behaves as expected, leaving any vector in the span of $\mathbf{P}$ invariant, but projecting any vector orthogonal to the span of $\mathbf{P}$ onto 0.

Example:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.33}$$

is an orthogonal projector that maps all vectors in $\mathbb{R}^3$ onto the x-y plane, while keeping those vectors in the x-y plane unchanged:

$$\mathbf{P}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}, \tag{3.34}$$

and

$$\mathbf{P}\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}, \text{ while } \mathbf{P}\begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.35}$$

It is easy to check $\mathbf{P}^2 = \mathbf{P}$:

$$\mathbf{P}^2\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{P}\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}. \tag{3.36}$$

□

Note: Given an orthogonal projector $\mathbf{P}$ we can also define

$$\mathbf{P}_\perp = \mathbf{I} - \mathbf{P} \tag{3.37}$$

which can be shown to be an orthogonal projector onto span$(\mathbf{P})^\perp$, the orthogonal complement of span$(\mathbf{P})$. Then, we can express any vector $\mathbf{x} \in \mathbb{R}^m$ as a sum

$$\mathbf{x} = \Big(\mathbf{P} + (\mathbf{I} - \mathbf{P})\Big)\mathbf{x} = \mathbf{P}\mathbf{x} + \mathbf{P}_\perp\mathbf{x}. \tag{3.38}$$

□

We will now use these definitions to go back to the problem discussed at the end of the previous section, namely how to construct $\mathbf{P}_A$, an orthogonal projector onto the span of any given $m \times n$ matrix $\mathbf{A}$.

Let us now consider how we can make use of the concept of the orthogonal projector $\mathbf{P}_A$ onto span$(\mathbf{A})$ to help understanding in solving the overdetermined system $\mathbf{A}\mathbf{x} \cong \mathbf{b}$. For notational simplicity, let us denote $\mathbf{P} = \mathbf{P}_A$ for now. First, by definition, we get

$$\mathbf{P}\mathbf{A} = \mathbf{A}, \mathbf{P}_\perp\mathbf{A} = \mathbf{0}. \tag{3.39}$$

Then we have

$$\begin{aligned} ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 &= ||\mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x}) + \mathbf{P}_\perp(\mathbf{b} - \mathbf{A}\mathbf{x})||_2^2 \\ &= ||\mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x})||_2^2 + ||\mathbf{P}_\perp(\mathbf{b} - \mathbf{A}\mathbf{x})||_2^2 \text{ (by Pythagorean Theorem)} \\ &= ||\mathbf{P}\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 + ||\mathbf{P}_\perp\mathbf{b}||_2^2. \end{aligned} \tag{3.40}$$

Therefore, we see that the least squares solution is given by the solution $\mathbf{x}$ that satisfies the first term in the last relation, which is the solution to the overdetermined linear system,

$$\mathbf{Ax} = \mathbf{Pb}. \tag{3.41}$$

This is an intuitively clear result and is shown in Fig. 3 that the orthogonal projection $\mathbf{Pb}$ of $\mathbf{b}$ gives $\mathbf{y} \in \text{span}(\mathbf{A})$, the closest vector to $\mathbf{b}$.

Remember that we wish to transform our given overdetermined $m \times n$ system into an $n \times n$ square system, so that we can use the techniques we learned in Chapter 2. Assuming $\text{rank}(\mathbf{A}) = n$, there are two ways to construct orthogonal projectors (i.e., symmetric and idempotent) *explicitly*, which allow us to have transformation into the square system:

- $\mathbf{P} = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$.
- $\mathbf{P} = \mathbf{QQ}^T$, where $\mathbf{Q}$ is an $m \times n$ matrix whose columns form an orthonormal bases (such $\mathbf{Q}$ is said to be *orthogonal* and satisfies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) for $\text{span}(\mathbf{A})$. Obviously, this gives $\text{span}(\mathbf{Q}) = \text{span}(\mathbf{A})$.

First of all, we see that both choices of $\mathbf{P}$ easily satisfy $\mathbf{P}^T = \mathbf{P}$ and $\mathbf{P}^2 = \mathbf{P}$. Also, after substituting $\mathbf{P}$ into Eq. 3.41, one can respectively obtain the following square systems:

- $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ (note here we used $\mathbf{A}^T\mathbf{P} = \mathbf{A}^T\mathbf{P}^T = (\mathbf{PA})^T = \mathbf{A}^T$)
- $\mathbf{Q}^T\mathbf{Ax} = \mathbf{Q}^T\mathbf{b}$ (note here we used $\mathbf{Q}^T\mathbf{P} = \mathbf{Q}^T\mathbf{QQ}^T = \mathbf{Q}^T$)

Notice that the first transformation – the easier construction of the two – results in the system of normal equations which we already showed there are some associated numerical issues. The second orthogonal transformation, however, will provide us with a very useful idea on accomplishing so called the *QR factorization* as will be shown in Section 5.

4. Invariant Transformations

We will now focus on examining several methods for transforming an overdetermined $m \times n$ linear least squares problem $\mathbf{Ax} \cong \mathbf{b}$ into an $n \times n$ square linear system $\mathbf{A}'\mathbf{x} = \mathbf{b}'$ which leaves $\mathbf{x}$ unchanged and which we already know how to solve using the methods of Chapter 2.

In seeking for invariant transformations we keep in mind that the sequence of problem transformation we would like to establish is:

$$\text{rectangular} \rightarrow \text{square} \rightarrow \text{triangular}$$

Note: The second transformation (square to triangular) is what we already have learned in Chapter 2; while we now try to learn the first transformation (rectangular to square) in this chapter. □

4.1. Normal Equations

With rank($\mathbf{A}$) $= n$ we already have seen several times the $n \times n$ symmetric positive definite system of normal equations

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \tag{3.42}$$

has the invariant property of preserving the same solution $\mathbf{x}$ as the $m \times n$ least squares problem $\mathbf{A}\mathbf{x} \cong \mathbf{b}$.

As discussed, we could theoretically pursuit to use the Cholesky factorization,

$$\mathbf{A}^T\mathbf{A} = \mathbf{L}\mathbf{L}^T, \tag{3.43}$$

followed by solving the forward-substitution first $\mathbf{L}\mathbf{y} = \mathbf{A}^T\mathbf{b}$, and then the backward-substitution later $\mathbf{L}^T\mathbf{x} = \mathbf{y}$. But we've seen there are numerical accuracy and stability issues that are related to floating-point arithmetics as well as condition number squaring effects. In this reason, we do not use the normal equations in practice.

4.2. Orthogonal Transformations

In view of the potential numerical difficulties with the normal equations approach, we need an alternative that does not require $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$. In this alternative, we expect a more numerically robust type of transformation.

Recall that in Chapter 2 we did use a similar trick to introduce transformations to a simpler system which was a triangular system. Can we use the same triangular form for the current purpose? The answer is no simply because such a triangular transformation does not preserve what we want to preserve in the least squares problems now, the l^2-norm.

What kind other transformation then preserve the norm, at the same time, without changing the solution $\mathbf{x}$? Hinted by the previous section, we see that an orthogonal transformation given by $\mathbf{Q}$, where $\mathbf{Q}$ is a orthogonal real square matrix, i.e., $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ (in other words, each column of $\mathbf{Q}$ is orthonormal basis) would be a good candidate.

The norm-preserving property of $\mathbf{Q}$ can be easily shown:

$$||\mathbf{Q}\mathbf{x}||_2^2 = (\mathbf{Q}\mathbf{x})^T\mathbf{Q}\mathbf{x} = \mathbf{x}^T\mathbf{Q}^T\mathbf{Q}\mathbf{x} = \mathbf{x}^T\mathbf{x} = ||\mathbf{x}||_2^2. \tag{3.44}$$

Similarly, we also have

$$||\mathbf{Q}^T\mathbf{x}||_2^2 = (\mathbf{Q}^T\mathbf{x})^T\mathbf{Q}\mathbf{x} = \mathbf{x}^T\mathbf{Q}\mathbf{Q}^T\mathbf{x} = \mathbf{x}^T\mathbf{x} = ||\mathbf{x}||_2^2. \tag{3.45}$$

Remark: Orthogonal matrices are of great importance in many areas of numerical computation because of their norm-preserving property. With this property, the magnitude of errors will remain the same without any amplification. Thus, for example, one can use orthogonal transformations to solve square linear systems which will *not* require the need for pivoting for numerical stability. Although it looks very attractive the orthogonalization process is significantly more expensive computationally then the standard Gaussian elimination, so their superior numerical properties come at a price. □

4.3. Triangular Least Squares

As we now prepared ourselves with a couple of transformations that preserves the least squares solution, we are further motivated to seek for a more simplified system where a least squares problem can be solved in an easier way. As seen in the square linear systems in Chapter 2, we consider least squares problems with an upper triangular matrix of the form:

$$\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \cong \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix} = \mathbf{c}, \tag{3.46}$$

where $\mathbf{R}$ is an $n \times n$ upper triangular matrix, $\mathbf{x}$ an n-vector. The right hand side vector $\mathbf{c}$ is partitioned accordingly into an n-vector $\mathbf{c}_1$ and an $(m-n)$-vector $\mathbf{c}_2$.

We see that the least squares residual is given by

$$||\mathbf{r}||_2^2 = ||\mathbf{c}_1 - \mathbf{R}\mathbf{x}||_2^2 + ||\mathbf{c}_2||_2^2, \tag{3.47}$$

which tells us that the the least squares solution $\mathbf{x}$ satisfies

$$\mathbf{R}\mathbf{x} = \mathbf{c}_1, \tag{3.48}$$

which is solvable by back-substitution. The minimum residual then becomes

$$||\mathbf{r}||_2^2 = ||\mathbf{c}_2||_2^2. \tag{3.49}$$

4.4. QR Factorization

Let us now combine the two nice techniques, the orthogonal transformation and the triangular least squares, into one, and we call this method the *QR factorization*. The QR factorization method writes $m \times n$ $(m > n)$ matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}, \tag{3.50}$$

where $\mathbf{Q}$ is an $m \times m$ orthogonal matrix and $\mathbf{R}$ is an $n \times n$ upper triangular matrix.

This QR factorization transforms the *linear squares problem* $\mathbf{A}\mathbf{x} \cong \mathbf{b}$ into a *triangular least squares problem*. Do they both have the same solution? To see this, we check:

$$\begin{aligned}
||\mathbf{r}||_2^2 &= ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2^2 && (3.51)\\
&= ||\mathbf{b} - \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\mathbf{x}||_2^2 && (3.52)\\
&= ||\mathbf{Q}^T(\mathbf{b} - \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\mathbf{x})||_2^2 && (3.53)\\
&= ||\mathbf{Q}^T\mathbf{b} - \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\mathbf{x}||_2^2 && (3.54)\\
&= ||\mathbf{c}_1 - \mathbf{R}\mathbf{x}||_2^2 + ||\mathbf{c}_2||_2^2, && (3.55)
\end{aligned}$$

where the transformed right hand side

$$\mathbf{Q}^T\mathbf{b} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}, \tag{3.56}$$

with n-vector $\mathbf{c}_1$ and an $(m-n)$-vector $\mathbf{c}_2$. As in the previous section, we make the same conclusion on $\mathbf{x}$ that satisfies:

$$\mathbf{Rx} = \mathbf{c}_1, \tag{3.57}$$

which is solvable by back-substitution, and the minimum residual is

$$||\mathbf{r}||_2^2 = ||\mathbf{c}_2||_2^2. \tag{3.58}$$

We will study how to compute this QR factorization in the next section.

5. The QR factorization for Least Square problems

5.1. Preliminaries

See Chapter 7 of the textbook

Definition: A reduced QR factorization of a full rank $m \times n$ $(m > n)$ matrix $\mathbf{A}$ expresses $\mathbf{A}$ as

$$\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}} \tag{3.59}$$

where $\hat{\mathbf{Q}}$ is an $m \times n$ matrix whose column vectors are orthonormal, and $\hat{\mathbf{R}}$ is an $n \times n$ upper triangular matrix.

Definition: A full QR factorization of a full rank $m \times n$ $(m > n)$ matrix $\mathbf{A}$ expresses $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{QR} \tag{3.60}$$

where $\mathbf{Q}$ is an $m \times m$ orthogonal matrix (unitary, if $\mathbf{A}$ is complex) and $\mathbf{R}$ is an $m \times n$ upper triangular matrix (i.e. a matrix whose entries r_{ij} are 0 if $i > j$).

The two definitions are related in the sense that $\hat{\mathbf{Q}}$ forms the first n column-vectors of $\mathbf{Q}$, and $\hat{\mathbf{R}}$ forms the first n rows of $\mathbf{R}$. The last $m-n$ columns of $\mathbf{Q}$ are filled with vectors that are orthogonal to each other and to the span of $\hat{\mathbf{Q}}$, while the last $m-n$ rows of $\mathbf{R}$ are just filled with zeros. We therefore have

$$\mathbf{Q} = \left(\hat{\mathbf{Q}}\middle|\mathbf{q}_{n+1} \quad \dots \quad \mathbf{q}_m\right) \text{ and } \mathbf{R} = \left(\frac{\hat{\mathbf{R}}}{\mathbf{0}}\right) \tag{3.61}$$

As it turns out, knowing $\hat{\mathbf{Q}}$ is all we need to construct the projector $\mathbf{P}_A$. That's because it can be shown that $\mathbf{P} = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T$ is indeed a projector on the subspace spanned by $\mathbf{A}$, hence $\mathbf{P}_A = \mathbf{P}$. To do this we must show that

- The span of $\mathbf{P}$ is the same as the span of $\mathbf{A}$

- $\mathbf{P}^2 = \mathbf{P}$
- $\mathbf{P}^T = \mathbf{P}$.

Proving the second and third statements is very easy:

$$\mathbf{P}^T = (\hat{\mathbf{Q}}\hat{\mathbf{Q}}^T)^T = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T \tag{3.62}$$

$$\mathbf{P}^2 = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\hat{\mathbf{Q}}^T = \hat{\mathbf{Q}}\mathbf{I}\hat{\mathbf{Q}}^T = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T = \mathbf{P} \tag{3.63}$$

because $\hat{\mathbf{Q}}^T\hat{\mathbf{Q}}$ is an $n \times n$ identity matrix by the orthonormality of the columns of $\hat{\mathbf{Q}}$.

To show that the span of $\mathbf{P}$ is the same as a span of $\mathbf{A}$ is a little trickier, but not much. We can first show that the span of $\hat{\mathbf{Q}}$ lies in the span of $\mathbf{A}$ by noting that $\hat{\mathbf{Q}} = \mathbf{A}\hat{\mathbf{R}}^{-1}$, so the column vectors of $\hat{\mathbf{Q}}$, $\mathbf{q}_i = \mathbf{A}(\hat{\mathbf{R}}^{-1})_i$ (where $(\hat{\mathbf{R}}^{-1})_i$ are the column vectors of $\hat{\mathbf{R}}^{-1}$), are necessarily a linear combination of the column vectors of $\mathbf{A}$ (see first lecture). Then, since the $\mathbf{q}_i$ vectors are mutually orthogonal, they are also linearly independent. Finally, since there are n of them, the rank of $\hat{\mathbf{Q}}$ is n, which is the same as the rank of $\mathbf{A}$, so the span of $\hat{\mathbf{Q}}$ is *equal* to the span of $\mathbf{A}$. A series of very similar arguments can then be applied to show that the span of $\mathbf{P}$ is equal to the span of $\hat{\mathbf{Q}}$, and therefore to the span of $\mathbf{A}$.

This shows that $\mathbf{P} = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T$ is the orthogonal projector onto the span of $\mathbf{A}$, namely $\mathbf{P}_A$, that we were looking for earlier. We then write

$$\mathbf{A}\mathbf{x} = \mathbf{P}_A\mathbf{b} = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T\mathbf{b} \rightarrow \hat{\mathbf{Q}}^T\mathbf{A}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b} \rightarrow \hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b} \rightarrow \hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b} \tag{3.64}$$

where we repeatedly used the fact that $\hat{\mathbf{Q}}^T\hat{\mathbf{Q}} = \mathbf{I}$. In other words, an alternative algorithm for solving the overdetermined system $\mathbf{A}\mathbf{x} = \mathbf{b}$ involves

- Finding the reduced QR factorization of $\mathbf{A}$, such that $\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$
- Solving the exact system $\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$

Note that $\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$ is an exact system since $\hat{\mathbf{R}}$ is $n \times n$ and $\mathbf{x}$ and $\hat{\mathbf{Q}}^T\mathbf{b}$ are both $n-$long vectors. Also, since $\hat{\mathbf{R}}$ is upper triangular, the second step boils down to a very basic back-substitution step! The crux of the method is therefore not step 2, but step 1, the QR factorization.

To understand how to construct a QR factorization, it is worth trying to interpret its meaning geometrically. A simple way of seing what the reduced QR factorization does is to note that it constructs an orthonormal basis for the space spanned by the column vectors of $\mathbf{A}$, namely the basis formed by the column vectors of $\hat{\mathbf{Q}}$. Furthermore, since

$$\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}} \rightarrow \mathbf{a}_j = \hat{\mathbf{Q}}\mathbf{r}_j \tag{3.65}$$

This explicitly writes each vector $\mathbf{a}_j$ in this new orthonormal basis, and the components of the vector $\mathbf{r}_j$ are the coordinates of $\mathbf{a}_j$ in the new basis.

There are a number of commonly used methods to construct the reduced QR factorization of the matrix $\mathbf{A}$. The two we will see here are

- Gram-Schmidt orthogonalization,
- Householder transformation (based on elementary reflectors).

There is another popular method that uses so-called Givens transformations, which is based on plane rotations. This method can be found in many numerical linear algebra text books but we will not cover it here. Finally, in all that follows we now assume that $\mathbf{A}$ is real, although the various methods are very easy to generalize for complex matrices using unitary transformations instead of orthogonal ones.

5.2. QR decomposition using Gram-Schmidt Orthogonalization

See Chapters 8 and 11

The most straightforward method for computing the QR factorization is *Gram-Schmidt orthogonalization.* It works directly with the expression $\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$, expressing it in vector form, and progressively solving for the columns of $\hat{\mathbf{Q}}$ and the elements of $\hat{\mathbf{R}}$. To see how, note first that $\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$ is equivalent to the system of vector equations

$$\begin{aligned} \mathbf{a}_1 &= r_{11}\mathbf{q}_1 \\ \mathbf{a}_2 &= r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 \\ &\vdots \\ \mathbf{a}_i &= \sum_{k=1}^{i} r_{ki}\mathbf{q}_k \end{aligned} \tag{3.66}$$

This shows that

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||} \text{ and } r_{11} = ||\mathbf{a}_1|| \tag{3.67}$$

Next, we have

$$\mathbf{q}_2 = \frac{\mathbf{a}_2 - r_{12}\mathbf{q}_1}{r_{22}} \tag{3.68}$$

Once $\mathbf{a}_2 - r_{12}\mathbf{q}_1$ is known we can calculate $r_{22} = ||\mathbf{a}_2 - r_{12}\mathbf{q}_1||$ to normalize $\mathbf{q}_2$. The value of r_{12} if found by requiring the orthogonality of $\mathbf{q}_2$ with $\mathbf{q}_1$: we need $\mathbf{q}_1^T\mathbf{q}_2 = 0$ so $r_{12} = \mathbf{q}_1^T\mathbf{a}_2$ since $\mathbf{q}_1$ is normalized. This shows that

$$\mathbf{q}_2 = \frac{\mathbf{a}_2 - (\mathbf{q}_1^T\mathbf{a}_2)\mathbf{q}_1}{r_{22}} = \frac{\mathbf{a}_2 - \mathbf{q}_1\mathbf{q}_1^T\mathbf{a}_2}{r_{22}} = \frac{(\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^T)\mathbf{a}_2}{r_{22}} \tag{3.69}$$

Note that in the last expression above, it is easy to show, based on what we learned about projectors earlier, that $\mathbf{q}_1\mathbf{q}_1^T$ is an orthogonal projector onto the vector $\mathbf{q}_1$, that we shall call $\mathbf{P}_1$. This will come in handy later when interpreting the process geometrically.

At the i-th step, we then have

$$\mathbf{q}_i = \frac{\mathbf{a}_i - \sum_{k=1}^{i-1} r_{ki}\mathbf{q}_k}{r_{ii}} \tag{3.70}$$

and requiring as before orthogonality of $\mathbf{q}_i$ with all previously determined vectors we get $r_{ki} = \mathbf{q}_k^T \mathbf{a}_i$. This can be used to form $\mathbf{a}_i - \sum_{k=1}^{i-1} r_{ki}\mathbf{q}_k$, and then $\mathbf{q}_i$ after normalization. And as before, we can also interpret

$$\mathbf{a}_i - \sum_{k=1}^{i-1} r_{ki}\mathbf{q}_k = (\mathbf{I} - \sum_{k=1}^{i-1} \mathbf{P}_k)\mathbf{a}_i \tag{3.71}$$

where $\mathbf{P}_k = \mathbf{q}_k\mathbf{q}_k^T$ is a projector onto $\mathbf{q}_k$.

We can write the so-called basic Gram-Schmidt orthogonalization algorithm as follows:

Algorithm: Basic Gram-Schmidt orthogonalization:

```
do $i = 1$ to $n$
    [!loop over columns of $\mathbf{A}$ and $\hat{\mathbf{Q}}$]
    $\mathbf{q}_i = \mathbf{a}_i$
    do $k = 1$ to $i - 1$
        $r_{ki} = \mathbf{q}_k^T \mathbf{a}_i$
        [!Form the coefficients of $\hat{\mathbf{R}}$]
        $\mathbf{q}_i = \mathbf{q}_i - r_{ki}\mathbf{q}_k$ [!Compute $\mathbf{q}_i$]
    enddo
    $r_{ii} = ||\mathbf{q}_i||$
    if $r_{ii} = 0$ stop
    $\mathbf{q}_i = \mathbf{q}_i / r_{ii}$ [!Normalize $\mathbf{q}_i$]
enddo
```

In this algorithm we treated $\mathbf{a}_i$ and $\mathbf{q}_i$ separately for clear exposition purpose, but they can be shared in the same storage, with new $\mathbf{q}_i$ gradually replacing the $\mathbf{a}_i$ □

While we derived the algorithm from mathematical considerations, it has a very simple geometrical interpretation, illustrated in Figure 4.

Indeed, the first $\mathbf{q}_1$ is selected to be the unit vector in the direction of $\mathbf{a}_1$, and successive vectors $\mathbf{q}_i$ are constructed using the orthogonal projectors $\mathbf{P}_k$ to be orthogonal to the previous ones. For instance, we see that $(\mathbf{I} - \mathbf{P}_1)\mathbf{a}_2$ constructs a vector that is perpendicular to $\mathbf{q}_1$ while lying in the plane spanned by $\mathbf{a}_1$ and $\mathbf{a}_2$, and by successive application of the algorithm, $(\mathbf{I} - \mathbf{P}_1 - \mathbf{P}_2 - \cdots - \mathbf{P}_{i-1})\mathbf{a}_i$ constructs a vector that is perpendicular to the span of $\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{i-1}\}$ while lying in the span of $\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{i-1}, \mathbf{q}_i\}$

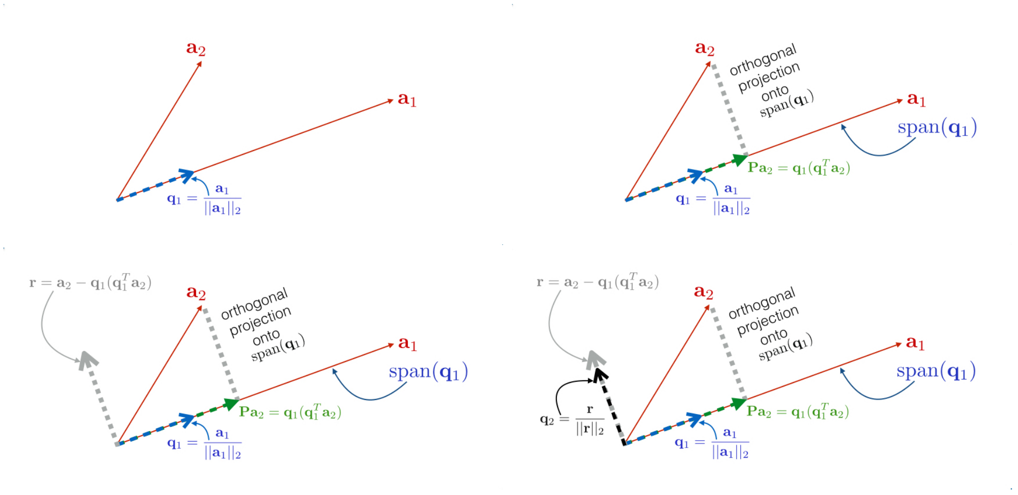

$\mathbf{a}_2$
$\mathbf{a}_1$
$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||_2}$
$\mathbf{a}_2$
orthogonal projection onto span($\mathbf{q}_1$)
$\mathbf{a}_1$
span($\mathbf{q}_1$)
$\mathbf{P}\mathbf{a}_2 = \mathbf{q}_1(\mathbf{q}_1^T\mathbf{a}_2)$
$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||_2}$
$\mathbf{r} = \mathbf{a}_2 - \mathbf{q}_1(\mathbf{q}_1^T\mathbf{a}_2)$
$\mathbf{a}_2$
orthogonal projection onto span($\mathbf{q}_1$)
$\mathbf{a}_1$
span($\mathbf{q}_1$)
$\mathbf{P}\mathbf{a}_2 = \mathbf{q}_1(\mathbf{q}_1^T\mathbf{a}_2)$
$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||_2}$
$\mathbf{r} = \mathbf{a}_2 - \mathbf{q}_1(\mathbf{q}_1^T\mathbf{a}_2)$
$\mathbf{a}_2$
orthogonal projection onto span($\mathbf{q}_1$)
$\mathbf{a}_1$
span($\mathbf{q}_1$)
$\mathbf{q}_2 = \frac{\mathbf{r}}{||\mathbf{r}||_2}$
$\mathbf{P}\mathbf{a}_2 = \mathbf{q}_1(\mathbf{q}_1^T\mathbf{a}_2)$
$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||_2}$

Figure 4. Geometrical interpretation of Gram-Schmidt orthogonalization

As it turns out, it can be shown that this Gram-Schmidt algorithm is unfortunately numerically unstable (see below for more on this). However there is a remarkably simple fix to that problem – simply switch the order of the operations, leading to the **Modified Gram-Schmidt algorithm**:

Algorithm: Modified Gram-Schmidt orthogonalization:

```
do $i = 1$ to $n$
    [!First initialize temporary vectors $\mathbf{v}_i$ as the $\mathbf{a}_i$]
    $\mathbf{v}_i = \mathbf{a}_i$
enddo
do $i = 1$ to $n$
    $r_{ii} = ||\mathbf{v}_i||$
    $\mathbf{q}_i = \mathbf{v}_i / r_{ii}$ [!Create $\mathbf{q}_i$ normalizing $\mathbf{v}_i$ ]
    do $k = i + 1$ to $n$
    [!Apply simple projector to all the vectors $\mathbf{v}_k$ for $k > i$]

        $r_{ik} = \mathbf{q}_i^T \mathbf{v}_k$
        $\mathbf{v}_k = \mathbf{v}_k - r_{ik}\mathbf{q}_i$
    enddo
enddo
```

As before, we can alternatively store the $\mathbf{q}$ vectors into the columns of $\mathbf{A}$ as the algorithm proceeds. With a bit of work, one can show that the operations performed end up being *exactly* the same as in the basic Gram-Schmidt algorithm, but their order is different, and the new ordering stabilizes the algorithm. The modified algorithm has therefore become the standard in commercial packages.

It is also interesting to note that this modified algorithm can be viewed as the successive multiplication of the matrix $\mathbf{A}$ from the right by upper triangular matrices $\mathbf{R}_i$, as

$$\mathbf{A}\mathbf{R}_1\mathbf{R}_2\ldots\mathbf{R}_n = \hat{\mathbf{Q}} \text{ where } \mathbf{R}_i = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \frac{1}{r_{ii}} & -\frac{r_{i,i+1}}{r_{ii}} & \cdots & -\frac{r_{i,n}}{r_{ii}} \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix} \tag{3.72}$$

where the coefficients r_{ij} were defined earlier. Since the multiplication of upper triangular matrices is another upper triangular one, and since the inverse of an upper triangular matrix is also upper triangular, we then have

$$\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}} \text{ where } \hat{\mathbf{R}} = (\mathbf{R}_1\mathbf{R}_2\ldots\mathbf{R}_n)^{-1} \tag{3.73}$$

For this reason, the procedure is sometimes referred to as **triangular orthogonalization** (i.e. multiply by triangular matrices to form an orthogonal one).

Note that in practice, however, it is rather difficult to find examples of commonly occurring matrices for which an instability genuinely manifests itself in the basic algorithm. Recall that for instability to occur an algorithm must return an answer whose error does *not* scale with machine accuracy. See the textbook Chapter 9 for such an example. What is much more common, however, is to note that *both* Gram-Schmidt-based QR decomposition algorithms (standard and modified) suffer from the accumulation of round-off errors, and this manifests itself in two ways when applied to poorly conditioned matrices:

- $||\mathbf{A} - \hat{\mathbf{Q}}\hat{\mathbf{R}}|| \gg \epsilon_{\text{mach}}$ (loss of accuracy)
- $||\hat{\mathbf{Q}}^T\hat{\mathbf{Q}} - \mathbf{I}|| \gg \epsilon_{\text{mach}}$ (loss of orthogonality)

Let us see a simple example of this.

Example: Consider the general 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{3.74}$$

Applying either of the Gram-Schmidt algorithm (whose steps are exactly the same for a 2×2 matrix), we have

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \rightarrow r_{11} = \sqrt{a^2+c^2} \rightarrow \mathbf{A} = \begin{pmatrix} \frac{a}{\sqrt{a^2+c^2}} & b \\ \frac{c}{\sqrt{a^2+c^2}} & d \end{pmatrix} \tag{3.75}$$

then

$$r_{12} = \frac{ab+cd}{\sqrt{a^2+c^2}} \rightarrow \mathbf{A} = \begin{pmatrix} \frac{a}{\sqrt{a^2+c^2}} & \frac{c(bc-ad)}{a^2+c^2} \\ \frac{c}{\sqrt{a^2+c^2}} & \frac{a(ad-bc)}{a^2+c^2} \end{pmatrix} \tag{3.76}$$

We easily recognize $D = ad - bc$ appear in the second column vector. Let us assume, for the sake of simplicity, that $D > 0$ (a similar line of argument applies if $D < 0$). We then finally normalize the second vector to get

$$r_{22} = \frac{D}{\sqrt{a^2+c^2}} \rightarrow \mathbf{A} = \frac{1}{\sqrt{a^2+c^2}} \begin{pmatrix} a & -c \\ c & a \end{pmatrix} \tag{3.77}$$

It is trivial to verify that, in this exact case, the column vectors of $\mathbf{A}$ are indeed orthogonal. In addition, we can also check easily that $\mathbf{A} - \mathbf{QR} = 0$.

However, what happens when the matrix is nearly singular? In that case, recall that $D \simeq 0$ even when a, b, c and d themselves are of order unity. We also saw that, when performing floating point operations, the absolute error on the subtraction of two numbers is equal to machine error times the size of the largest number. In the calculations of the two components of the second column vector, the error on $D = ad - bc$ can therefore be as large as D if the latter is close to zero, and so the result would be an approximate matrix

$$\mathbf{A} = \begin{pmatrix} \frac{a}{\sqrt{a^2+c^2}} & \frac{c(-D+\epsilon_1)}{a^2+c^2} \\ \frac{c}{\sqrt{a^2+c^2}} & \frac{a(D+\epsilon_2))}{a^2+c^2} \end{pmatrix} \tag{3.78}$$

where ϵ_1 and ϵ_2 are order machine precision (if a, b, c and d are order unity). If $D = O(\epsilon_1) = O(\epsilon_2)$ as well, the second column can be quite far from being orthogonal to the first. In addition, r_{22} will be of the same order as D, ϵ_1 and ϵ_2, while r_{11} is of order a and/or c. The matrix $\mathbf{R}$ will then be very poorly conditioned, with a condition number $O(D^{-1})$. The relative error on the matrix multiplication $\mathbf{QR}$ is therefore of order $D^{-1}\epsilon_{\text{mach}} \gg \epsilon_{\text{mach}}$. □

The lack of both accuracy and orthogonality in the calculation of the QR decomposition naturally carry over to the original Least Square problem we were trying to solve. Recall that, if $\hat{\mathbf{Q}}\hat{\mathbf{R}}$ is known, then we can solve the Least Square problem $\mathbf{Ax} = \mathbf{b}$ by the two-step method $\hat{\mathbf{Q}}^T\hat{\mathbf{Q}}\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$, and then solve the upper-triangular $n \times n$ problem $\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$. The first step uses the fact that $\hat{\mathbf{Q}}^T\hat{\mathbf{Q}} = \mathbf{I}$, so large errors may appear if this is no longer true because of loss of orthogonality. Compounding on this, the relative error of the second step is proportional to the condition number of $\hat{\mathbf{R}}$, which is large if the problem is poorly conditioned.

Luckily, the orthogonality problem of step 1 can be solved, even if the accuracy problem of step 2 cannot (for poorly conditioned matrices). Doing so requires a completely different approach to orthogonalization, using so-called Householder transformations.

Example: Let us consider to solve the same least squares problem of the land surveyors example given in Eq. 3.5, using Gram-Schmidt orthogonalization. The first step is to normalize the first column of **A**:

$$r_{11} = ||\mathbf{a}_1||_2 = 1.7321, \quad \mathbf{q}_1 = \frac{\mathbf{a}_1}{r_{11}} = \begin{bmatrix} 0.5774 \\ 0 \\ 0 \\ -0.5774 \\ -0.5774 \\ 0 \end{bmatrix}. \tag{3.79}$$

Calculating orthogonalization processes and subtractions in **Step 2** and **Step 3**, we first obtain

$$r_{12} = \mathbf{q}_1^T \mathbf{a}_2 = -0.5774, \quad r_{13} = \mathbf{q}_1^T \mathbf{a}_3 = -0.5774, \tag{3.80}$$

where $\mathbf{a}_2$ and $\mathbf{a}_3$ are the second and the third column of **A**. Continuing the remaining procedures in **Step 2** and **Step 3** we get:

$$\mathbf{r}_2 = \mathbf{a}_2 - r_{12}\mathbf{q}_1 = \begin{bmatrix} 0.3333 \\ 1 \\ 0 \\ 0.6667 \\ -0.3333 \\ -1 \end{bmatrix}, \quad \mathbf{r}_3 = \mathbf{a}_3 - r_{13}\mathbf{q}_1 = \begin{bmatrix} 0.3333 \\ 0 \\ 1 \\ -0.3333 \\ 0.6667 \\ 1 \end{bmatrix}. \tag{3.81}$$

The resulting transformed matrix now has $\mathbf{q}_1$ for its first column, together with $\mathbf{r}_2$ and $\mathbf{r}_3$ for the unnormalized second and the third columns:

$$\begin{bmatrix} 0.5774 & 0.3333 & 0.3333 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -0.5774 & 0.6667 & -0.3333 \\ -0.5774 & -0.3333 & 0.6667 \\ 0 & -1 & 1 \end{bmatrix} \tag{3.82}$$

Let us abuse our naming strategy and let $\mathbf{a}_2$ and $\mathbf{a}_3$ be the second and the third columns of the new matrix now. Normalizing the second column gives

$$r_{22} = ||\mathbf{a}_2||_2 = 1.6330, \quad \mathbf{q}_2 = \frac{\mathbf{a}_2}{r_{22}} = \begin{bmatrix} 0.2041 \\ 0.6124 \\ 0 \\ 0.4082 \\ -0.2041 \\ -0.6124 \end{bmatrix}. \tag{3.83}$$

If we evaluate the orthogonalization of the second column against the third column, we get:

$$r_{23} = \mathbf{q}_2^T \mathbf{a}_3 = -0.8165, \tag{3.84}$$

and hence we further obtain yet another residual vector

$$\mathbf{r}_3 = \mathbf{a}_3 - r_{23}\mathbf{q}_2 = \begin{bmatrix} 0.5 \\ 0.5 \\ 1 \\ 0 \\ 0.5 \\ 0.5 \end{bmatrix}. \tag{3.85}$$

We now form another transformed matrix which has $\mathbf{q}_2$ for its second orthonormal column and $\mathbf{r}_3$ for its unnormalized third column:

$$\begin{bmatrix} 0.5774 & 0.2041 & 0.5 \\ 0 & 0.6124 & 0.5 \\ 0 & 0 & 1 \\ -0.5774 & 0.4082 & 0 \\ -0.5774 & -0.2041 & 0.5 \\ 0 & -0.6124 & 0.5 \end{bmatrix}. \tag{3.86}$$

Finally we normalize the last column, again abusing our naming strategy and let the third column be $\mathbf{a}_3$:

$$r_{33} = ||\mathbf{a}_3||_2 = 1.4142, \quad \mathbf{q}_3 = \frac{\mathbf{a}_3}{r_{33}} = \begin{bmatrix} 0.3536 \\ 0.3536 \\ 0.7071 \\ 0 \\ 0.3536 \\ 0.3536 \end{bmatrix}, \tag{3.87}$$

and by replacing the last column of the last transformed matrix with $\mathbf{q}_3$ results in the final transformed matrix

$$\begin{bmatrix} 0.5774 & 0.2041 & 0.3536 \\ 0 & 0.6124 & 0.3536 \\ 0 & 0 & 0.7071 \\ -0.5774 & 0.4082 & 0 \\ -0.5774 & -0.2041 & 0.3536 \\ 0 & -0.6124 & 0.3536 \end{bmatrix}. \tag{3.88}$$

Now collecting entries r_{ij} of $\mathbf{R}$, we form

$$\mathbf{R} = \begin{bmatrix} 1.7321 & -0.5774 & -0.5774 \\ & 1.6330 & -0.8165 \\ & & 1.4142 \end{bmatrix}, \tag{3.89}$$

which altogether provides the QR factorization:

$$\mathbf{A} = \mathbf{QR} = \begin{bmatrix} 0.5774 & 0.2041 & 0.3536 \\ 0 & 0.6124 & 0.3536 \\ 0 & 0 & 0.7071 \\ -0.5774 & 0.4082 & 0 \\ -0.5774 & -0.2041 & 0.3536 \\ 0 & -0.6124 & 0.3536 \end{bmatrix} \begin{bmatrix} 1.7321 & -0.5774 & -0.5774 \\ & 1.6330 & -0.8165 \\ & & 1.4142 \end{bmatrix}. \tag{3.90}$$

Computing the right hand side transformation, $\mathbf{Q}^T\mathbf{b}$, we obtain

$$\mathbf{Q}^T\mathbf{b} = \begin{bmatrix} -376 \\ 1200 \\ 3417 \end{bmatrix} = \mathbf{c}_1. \tag{3.91}$$

This allows us to solve the upper triangular system $\mathbf{Rx} = \mathbf{c}_1$ by back-substitution, resulting in the same solution as before:

$$\mathbf{x}^T = [1236, 1943, 2416]. \tag{3.92}$$

□

5.3. QR decomposition using Householder Transformations

See Chapter 10 of the textbook

5.3.1. Householder matrices Given a vector $\mathbf{v}$ with $||\mathbf{v}|| = 1$, its corresponding Householder matrix $\mathbf{H}$ is

$$\mathbf{H} = \mathbf{I} - 2\mathbf{v}\mathbf{v}^T \tag{3.93}$$

We see that from the definition, $\mathbf{H}$ is both

- symmetric, i.e., $\mathbf{H}^T = \mathbf{H}$, and
- orthogonal, i.e., $\mathbf{H}^T\mathbf{H} = \mathbf{I}$, hence $\mathbf{H}^T = \mathbf{H}^{-1}$.

The geometrical interpretation of matrix $\mathbf{H}$ is actually quite simple: it is a **reflection** across the plane perpendicular to $\mathbf{v}$. To see this, note that for any vector parallel to $\mathbf{v}$ (namely $\mathbf{w} = \alpha\mathbf{v}$),

$$\mathbf{Hw} = \alpha\mathbf{Hv} = \alpha(\mathbf{v} - 2\mathbf{v}\mathbf{v}^T\mathbf{v}) = -\alpha\mathbf{v} = -\mathbf{w} \tag{3.94}$$

while for any vector $\mathbf{w}$ perpendicular to $\mathbf{v}$, we have

$$\mathbf{Hw} = \mathbf{w} - 2\mathbf{v}\mathbf{v}^T\mathbf{w} = \mathbf{w} \tag{3.95}$$

These two properties define an orthogonal reflection about a plane: any vector parallel to the plane exactly reflected ($\mathbf{w} \to -\mathbf{w}$), while vectors perpendicular to that plane remain invariant .

Remark: First recall that the orthogonal projector onto span($\mathbf{v}$) is given by

$$\mathbf{P} = \mathbf{v}(\mathbf{v}^T\mathbf{v})^{-1}\mathbf{v}^T = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}. \tag{3.96}$$

Also the orthogonal complement projector onto span$(\mathbf{v})^\perp$ is given by

$$\mathbf{P}_\perp = \mathbf{I} - \mathbf{P} = \mathbf{I} - \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}. \tag{3.97}$$

This projector $\mathbf{P}_\perp$ gives the projection of $\mathbf{a}$ onto the hyperplane,

$$\mathbf{P}_\perp \mathbf{a} = (\mathbf{I} - \mathbf{P})\mathbf{a} = \mathbf{a} - \mathbf{v}\frac{\mathbf{v}^T\mathbf{a}}{\mathbf{v}^T\mathbf{v}}, \tag{3.98}$$

which is only the half way through to the desired location, the first coordinate axis in the current example.

In order to reach the first coordinate axis we therefore need to go twice as far, which gives us our final form of $\mathbf{H}$:

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}. \tag{3.99}$$

The full design construction is illustrated in Fig. 5, where the hyperplane is given by $\text{span}(\mathbf{v})^\perp = \{\mathbf{x} : \mathbf{v}^T\mathbf{x} = \mathbf{0}\}$, for some $\mathbf{v} \neq \mathbf{0}$.

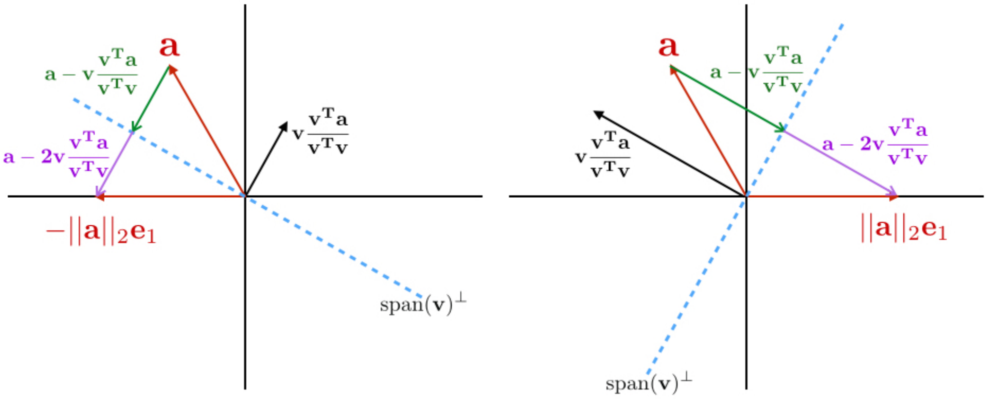

$\mathbf{a}$
$\mathbf{a} - \mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$\mathbf{a} - 2\mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$\mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$-||\mathbf{a}||_2\mathbf{e}_1$
$\text{span}(\mathbf{v})^\perp$
$\mathbf{a}$
$\mathbf{a} - \mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$\mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$\mathbf{a} - 2\mathbf{v}\frac{\mathbf{v}^\mathbf{T}\mathbf{a}}{\mathbf{v}^\mathbf{T}\mathbf{v}}$
$||\mathbf{a}||_2\mathbf{e}_1$
$\text{span}(\mathbf{v})^\perp$

Figure 5. Geometric interpretation of Householder transformation as reflection.

Note: We should make a quick comment on the choice of signs of $\alpha = \pm||\mathbf{a}||_2$ now. Depending the choice of the sign, we get the closer transformation $-||\mathbf{a}||_2\mathbf{e}_1$ from $\mathbf{a}$ (shown in the left panel in Fig. 5), or the farther one $||\mathbf{a}||_2\mathbf{e}_1$ from $\mathbf{a}$ (shown in the right panel in Fig. 5). Both choices should work just fine in principle, however, we know from experience that dealing with subtraction that results in small in magnitude, i.e., $\mathbf{v} = \mathbf{a} - \alpha\mathbf{e}_1$, is prone to numerical errors due to finite-precision arithmetic. In this reason, we prefer to choose the sign for α that yields the point on the first coordinate axis as farther away as possible from $\mathbf{a}$. □

5.3.2. Relationship between Householder transformations and QR decomposition. A QR factorization based on Householder transformations is sometimes known as the **orthogonal triangularization** of $\mathbf{A}$, by contrast with the Gram-Schmidt algorithm which performs a triangular orthogonalization (see above). The terminology implies that $\mathbf{A}$ is now gradually transformed into an upper triangular matrix $\mathbf{R}$ by successive application of orthogonal transformations, of the kind

$$\mathbf{Q}_n\mathbf{Q}_{n-1}\ldots\mathbf{Q}_1\mathbf{A} = \mathbf{R} \tag{3.100}$$

The product of orthogonal matrices $\mathbf{Q}_n\mathbf{Q}_{n-1}\ldots\mathbf{Q}_1$ is also orthogonal, so if we call it $\mathbf{Q}^T$, we then have

$$\mathbf{A} = \mathbf{Q}\mathbf{R} \text{ where } \mathbf{Q}^T = \mathbf{Q}_n\mathbf{Q}_{n-1}\ldots\mathbf{Q}_1 \tag{3.101}$$

Note that the Householder QR decomposition can only perform *full QR decompositions* which is why we have now begun to refer to the matrices as $\mathbf{Q}$ and $\mathbf{R}$ as defined in the last lecture, with $\mathbf{Q}$ an $m\times m$ matrix, and $\mathbf{R}$ an $m\times n$ matrix.

Based on this idea, we want to find a way of gradually zeroing out the subdiagonal elements of $\mathbf{A}$, which is superficially similar to what Gaussian elimination or LU decomposition do. The important difference however is that these operations must be done by orthogonal transformations! So how do we construct them? Here again, having a geometric mind can really help.

In what follows, it will be useful to remember that orthogonal operations on vectors are norm-preserving, since $||\mathbf{Q}\mathbf{x}|| = \sqrt{(\mathbf{Q}\mathbf{x})^T\mathbf{Q}\mathbf{x}} = \sqrt{\mathbf{x}^T\mathbf{x}} = ||\mathbf{x}||$, using the fact that $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. In the first step of transforming $\mathbf{A}$ into an upper triangular matrix, we want to zero out all the entries below the first one. Geometrically speaking, this involves creating an orthogonal transformation that takes the first column vector $\mathbf{a}_1$, and returns a vector $\mathbf{Q}_1\mathbf{a}_1 = \pm||\mathbf{a}_1||\mathbf{e}_1$, i.e. with the same norm but pointing in the $\mathbf{e}_1$ direction. Since orthogonal transformations are either rotations or reflections, we see that a simple way of constructing $\mathbf{Q}_1$ is to construct the relevant plane that reflects $\mathbf{a}_1$ onto $\pm||\mathbf{a}_1||\mathbf{e}_1$, as shown in Figure 6.

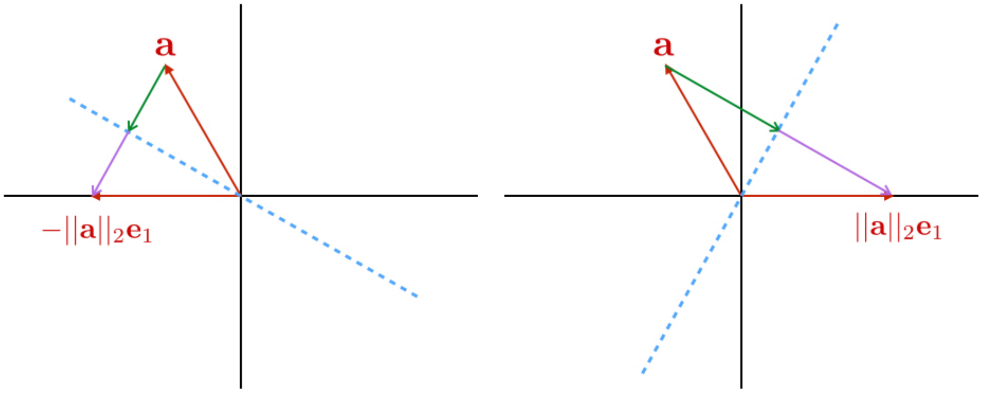

$\mathbf{a}$
$-||\mathbf{a}||_2\mathbf{e}_1$
$\mathbf{a}$
$||\mathbf{a}||_2\mathbf{e}_1$

Figure 6. Geometric interpretation of Householder transformation $\mathbf{H}$ as reflection. The transformation operator $\mathbf{H}$ is represented in two successive transformations denoted as the green arrow, followed by the purple arrow, across the hyperplane in dashed pale-blue line.

The left panel in Fig. 6 shows the principle of the method that reflects the given vector $\mathbf{a}_1$ to produce $-||\mathbf{a}_1||\mathbf{e}_1$. The reflection is established by bisecting the angle between $\mathbf{a}_1$ and the the first coordinate axis. Obviously, the norm is well preserved in this reflection.

Clearly, another transformation is also available, as indicated on the right panel in Fig. 6, where in this case the vector $\mathbf{a}_1$ is reflected onto $||\mathbf{a}_1||\mathbf{e}_1$.

Both choices of reflection should work just fine in principle, but the first can be shown to be more prone to numerical errors due to finite-precision arithmetic than the second. For this reason, we prefer to choose the sign ± that is equal to **minus** the sign of $\mathbf{a}_1^T\mathbf{e}_1$ (i.e. $-\text{sign}(a_{11})$).

Geometrically speaking, we therefore see that we need a Householder transformation (an orthogonal reflection across a plane) $\mathbf{H}_1$, such that

$$\mathbf{H}_1\mathbf{a}_1 = -s_1\mathbf{e}_1 \tag{3.102}$$

where $s_1 = \text{sign}(a_{11})||\mathbf{a}_1||$ is a signed norm of $\mathbf{a}_1$. The correct vector $\mathbf{v}_1$ that can perform this transformation is found by solving

$$(\mathbf{I} - 2\mathbf{v}_1\mathbf{v}_1^T)\mathbf{a}_1 = -s_1\mathbf{e}_1 \tag{3.103}$$

and requiring that $\mathbf{v}_1$ be normalized. The first condition implies that

$$\mathbf{v}_1 = \frac{\mathbf{a}_1 + s_1\mathbf{e}_1}{2\mathbf{v}_1^T\mathbf{a}_1} \tag{3.104}$$

which looks awkward because of the term in the denominator. But since this term is just a constant, and since we require $\mathbf{v}_1$ to be normalized, we then simply have

$$\mathbf{v}_1 = \frac{\mathbf{a}_1 + s_1\mathbf{e}_1}{||\mathbf{a}_1 + s_1\mathbf{e}_1||} \tag{3.105}$$

which can now be constructed easily once $\mathbf{a}_1$ is known. Note that, effectively, we have

$$\mathbf{v}_1 = (a_{11} + s_1, a_{21}, a_{31}, \dots, a_{m1})^T/||\mathbf{v}_1|| \tag{3.106}$$

To summarize, we can zero out the elements below the diagonal in the first column of $\mathbf{a}_1$ simply by constructing $\mathbf{v}_1$, then applying the corresponding Householder reflection $\mathbf{H}_1$ to $\mathbf{A}$. The effect of $\mathbf{H}_1$ on the other columns of $\mathbf{A}$ on the other hand is benign (i.e. does not do anything special).

Next, we look at the second column. This time, we effectively want to transform $\mathbf{a}_2$ into a vector that lies in the plane formed by $\mathbf{e}_1$ and $\mathbf{e}_2$, *without moving the transformed vector* $\mathbf{a}_1$ *away from* $\mathbf{e}_1$. For $\mathbf{e}_1$ to be invariant by transformation $\mathbf{H}_2$, $\mathbf{e}_1$ must lie in the plane of reflection, and therefore be perpendicular to the vector $\mathbf{v}_2$. Hence $\mathbf{v}_2$ must have a null first entry. By analogy with the construction of $\mathbf{v}_1$, and following this last remark, we therefore try

$$\mathbf{v}_2 = (0, a_{22} + s_2, a_{32}, \dots, a_{m2})^T/||\mathbf{v}_2|| \tag{3.107}$$

and $\mathbf{H}_2 = \mathbf{I} - 2\mathbf{v}_2\mathbf{v}_2^T$, where

$$s_2 = \text{sign}(a_{22}) \left(\sum_{k=2}^{m} a_{k2}^2 \right)^{1/2} \tag{3.108}$$

It can be verified that $\mathbf{H}_2$ indeed zeroes out the subdiagonal elements of the second column of $\mathbf{A}$, without modifying the first column. More generally, we then have at the j-th step, $\mathbf{H}_j = \mathbf{I} - 2\mathbf{v}_j\mathbf{v}_j^T$ where

$$\mathbf{v}_j = (0, \dots, 0, a_{jj} + s_j, a_{j+1,j}, \dots, a_{mj})^T / ||\mathbf{v}_j|| \tag{3.109}$$

and

$$s_j = \text{sign}(a_{jj}) \left(\sum_{k=j}^{m} a_{kj}^2 \right)^{1/2} \tag{3.110}$$

After n applications of the algorithm, we have

$$\mathbf{H}_n \dots \mathbf{H}_1 \mathbf{A} = \mathbf{R} \tag{3.111}$$

where each $\mathbf{H}_j$ is orthogonal by construction, as required, and $\mathbf{Q}^T = \mathbf{H}_n \dots \mathbf{H}_1$.

This then suggests the following QR decomposition algorithm:

Algorithm: Householder QR factorization algorithm:

```
do $j = 1$ to $n$
    ![loop over columns]
    $s_j = \text{sign}(a_{jj})\sqrt{\sum_{i=j}^{m} a_{ij}^2}$
    !  [compute signed norm]
    $\mathbf{v}_j = [0, \cdots 0, a_{jj} + s_j, a_{j+1,j}, \cdots, a_{mj}]^T$
    $\mathbf{v}_j = \mathbf{v}_j / ||\mathbf{v}_j||$
    !  [compute Householder vector and normalize it]
    $\mathbf{A} = \mathbf{A} - 2\mathbf{v}_j\mathbf{v}_j^T\mathbf{A}$  ![Update $\mathbf{A}$]
enddo
```

This algorithm gradually transforms $\mathbf{A}$ into $\mathbf{R}$, but does not compute nor save $\mathbf{Q}$. This turns out not to be needed as long as the $\mathbf{v}_j$ vectors are saved. Indeed, should we ever want to compute the action of $\mathbf{Q}$ or $\mathbf{Q}^T$ onto a vector $\mathbf{x}$, we simply have to remember their definitions, and the fact that each $\mathbf{H}_j$ matrix is symmetric and orthogonal at the same time. This implies

$$\begin{aligned} \mathbf{Q}^T\mathbf{x} &= \mathbf{H}_n \dots \mathbf{H}_1 \mathbf{x} = (\mathbf{I} - 2\mathbf{v}_n\mathbf{v}_n^T) \dots (\mathbf{I} - 2\mathbf{v}_1\mathbf{v}_1^T)\mathbf{x} \\ \mathbf{Q}\mathbf{x} &= (\mathbf{H}_n \dots \mathbf{H}_1)^{-1}\mathbf{x} = \mathbf{H}_1^{-1} \dots \mathbf{H}_n^{-1}\mathbf{x} = \mathbf{H}_1 \dots \mathbf{H}_n \mathbf{x} \\ &= (\mathbf{I} - 2\mathbf{v}_1\mathbf{v}_1^T) \dots (\mathbf{I} - 2\mathbf{v}_n\mathbf{v}_n^T)\mathbf{x} \end{aligned} \tag{3.112}$$

In each case, this implies repeated application of $\mathbf{H}_j$ to a vector, which can easily done as

$$\mathbf{H}_j\mathbf{x} = \mathbf{x} - 2\mathbf{v}_j(\mathbf{v}_j^T\mathbf{x}). \tag{3.113}$$

There are several options to save the vectors $\mathbf{v}_j$. One can, as in the example above, save them in a separate matrix $\mathbf{V}$ whose columns are $\mathbf{v}_j$. This is very simple, but not very memory efficient nor computationally efficient. A much more efficient way it to save the elements of $\mathbf{v}_j$ whose indices range from $j+1$ to m, in the column elements of the matrix $\mathbf{A}$ that have just been zeroed out. We then simply have to decide what to do with the diagonal elements of $\mathbf{A}$: we can either (i) choose to put all the elements of $\mathbf{R}$ including the diagonal elements in the corresponding upper triangular region of $\mathbf{A}$ and save the j-th elements of the $\mathbf{v}_j$, i.e., $\mathbf{v}_{jj} = a_{jj} + s_j$ vectors in a separate array, or (ii) the converse, namely to put the element $a_{jj} + s_j$ in diag($\mathbf{A}$) and return the diagonal elements of $\mathbf{R}$ (namely each $-s_j$) as a separate array. Some codes choose the first option (c.f. LAPACK) some choose the second (c.f. Numerical recipes), so it's really important to read the code description!

Finally, it is worth noting that the Householder QR algorithm is backward stable (see Chapter 16), with all the properties that this implies. The orthogonality of $\mathbf{Q}$ is guaranteed to be close to machine accuracy, but roundoff errors in $\mathbf{Q}$ and $\mathbf{R}$ mean that the reconstruction $\mathbf{QR}$ is not necessarily that close to $\mathbf{A}$ for poorly conditioned problem.

5.3.3. Solving a Least-Square problem using the QR Householder decomposition. While the Gram-Schmidt algorithm returns the reduced QR decomposition $\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$, which can then easily be used to solve the $n \times n$ exact problem $\hat{\mathbf{R}} = \hat{\mathbf{Q}}^T\mathbf{b}$, the Householder method returns the full QR decompostion $\mathbf{A} = \mathbf{QR}$. While the equality $\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$ is well-defined in the sense that it equates two $m-$long vectors, the problem appears ill-posed because there are only n unknowns (the n components of $\mathbf{x}$). This mismatch, however, does not matter as long as we remember that $\mathbf{R}$ contains $\hat{\mathbf{R}}$ in its first n rows, and only zeros afterwards, while the matrix $\mathbf{Q}$ contains $\hat{\mathbf{Q}}$ in its first columns, and other orthogonal vectors after that. This means that we can recover the original $\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$ problem by looking *only* at the first n lines of the problem $\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$ and ignoring the other lines.

Because the QR method using Householder transformation guarantees the orthogonality of $\mathbf{Q}$ to within machine accuracy, it is vastly preferred over Gram-Schmidt orthogonalization when used in the context of Least-Square problems. So one may wonder why ever use the Gram-Schmidt method? The answer to that question lies in the fact that the Householder algorithm is inherently serial and cannot be parallelized: each column j must be zeroed out before work can be done on column $j+1$. This is not true of *modified* Gram-Schmidt algorithm, where the entire matrix $\mathbf{A}$ is orthogonalized at the same time by the iterative process, rather than doing so one vector at a time. Some parallel QR algorithms therefore work with the Gram-Schmidt method instead, when it is perceived that the loss of accuracy is an acceptable price to pay for a vast gain in time.

Example: Let us now solve the land surveyor's least squares problem given by the system in Eq. 3.5 using Householder QR factorization:

$$\mathbf{Ax} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \cong \begin{bmatrix} 1237 \\ 1941 \\ 2417 \\ 711 \\ 1177 \\ 475 \end{bmatrix} = \mathbf{b}. \tag{3.114}$$

Recall that the solution we assumed to know was given by

$$\mathbf{x}^T = [h_1, h_2, h_3] = [1236, 1943, 2416], \tag{3.115}$$

and let's see if we get this solution indeed.

The first Householder step is to construct the Householder vector $\mathbf{v}_1$ that annihilates the subdiagonal entries of the first column $\mathbf{a}_1$ of $\mathbf{A}$ with, in this case, $s_1 = ||\mathbf{a}_1||_2 = \sqrt{3} \cong 1.7321$,

$$\mathbf{v}_1 = \mathbf{a}_1 + s_1\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1.7321 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2.7321 \\ 0 \\ 0 \\ -1 \\ -1 \\ 0 \end{bmatrix}. \tag{3.116}$$

Applying the resulting $\mathbf{H}_1$ to the first column gives

$$\mathbf{H}_1\mathbf{a}_1 = \mathbf{a}_1 - 2\mathbf{v}_1\frac{\mathbf{v}_1^T\mathbf{a}_1}{\mathbf{v}_1^T\mathbf{v}_1} = \begin{bmatrix} -1.7321 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.117}$$

Applying $\mathbf{H}_1$ to the second and third columns and also the right hand side vector $\mathbf{b}$ in a similar way gives, respectively:

$$\mathbf{H}_1\mathbf{a}_2 = \mathbf{a}_2 - 2\mathbf{v}_1\frac{\mathbf{v}_1^T\mathbf{a}_2}{\mathbf{v}_1^T\mathbf{v}_1} = \begin{bmatrix} 0.5774 \\ 1 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix}, \tag{3.118}$$

$$\mathbf{H}_1\mathbf{a}_3 = \mathbf{a}_3 - 2\mathbf{v}_1\frac{\mathbf{v}_1^T\mathbf{a}_3}{\mathbf{v}_1^T\mathbf{v}_1} = \begin{bmatrix} 0.5774 \\ 0 \\ 1 \\ -0.2113 \\ 0.7887 \\ 1 \end{bmatrix}, \tag{3.119}$$

and

$$\mathbf{H}_1\mathbf{b} = \mathbf{b} - 2\mathbf{v}_1 \frac{\mathbf{v}_1^T\mathbf{b}}{\mathbf{v}_1^T\mathbf{v}_1} = \begin{bmatrix} 376 \\ 1941 \\ 2417 \\ 1026 \\ 1492 \\ 475 \end{bmatrix}. \tag{3.120}$$

Putting all things together, we get

$$\mathbf{H}_1\mathbf{A} = \begin{bmatrix} -1.7321 & 0.5774 & 0.5774 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0.7887 & -0.2113 \\ 0 & -0.2113 & 0.7887 \\ 0 & -1 & 1 \end{bmatrix}, \quad \mathbf{H}_1\mathbf{b} = \begin{bmatrix} 376 \\ 1941 \\ 2417 \\ 1026 \\ 1492 \\ 475 \end{bmatrix}. \tag{3.121}$$

Next, we compute the second Householder vector $\mathbf{v}_2$ that annihilates the subdiagonal entries of the second column of $\mathbf{H}_1\mathbf{A}$ and leave the first entry 0.5774 unchanged (i.e., we choose all entries of $\mathbf{a}_2$ except for the first entry 0.5774):

$$\mathbf{a}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix}. \tag{3.122}$$

Then, with $s_2 = ||\mathbf{a}_2||_2 = 1.6330$, we get

$$\mathbf{v}_2 = \mathbf{a}_2 + s_2\mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1.6330 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 2.6330 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix}. \tag{3.123}$$

Now we apply $\mathbf{H}_2 = \mathbf{I} - 2\mathbf{v}_2 \frac{\mathbf{v}_2^T}{\mathbf{v}_2^T\mathbf{v}_2}$ onto the second and the third columns of $\mathbf{H}_1\mathbf{A}$ as well as $\mathbf{H}_1\mathbf{b}$, where the actual operation begins from the second entry in both columns, keeping the first entries unchanged. Writing these out explicitly, we get:

$$\mathbf{H}_2 \begin{bmatrix} 0.5774 \\ 1 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix} = \left(\mathbf{I} - 2\mathbf{v}_2 \frac{\mathbf{v}_2^T}{\mathbf{v}_2^T\mathbf{v}_2}\right) \begin{bmatrix} 0.5774 \\ 1 \\ 0 \\ 0.7887 \\ -0.2113 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.5774 \\ -1.6330 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{3.124}$$

$$\mathbf{H}_2 \begin{bmatrix} 0.5774 \\ 0 \\ 1 \\ -0.2113 \\ 0.7887 \\ 1 \end{bmatrix} = \left(\mathbf{I} - 2\mathbf{v}_2 \frac{\mathbf{v}_2^T}{\mathbf{v}_2^T \mathbf{v}_2}\right) \begin{bmatrix} 0.5774 \\ 0 \\ 1 \\ -0.2113 \\ 0.7887 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5774 \\ 0.8165 \\ 1 \\ 0.0333 \\ 0.7232 \\ 0.6899 \end{bmatrix}, \tag{3.125}$$

so that we have:

$$\mathbf{H}_2\mathbf{H}_1\mathbf{A} = \begin{bmatrix} -1.7321 & 0.5774 & 0.5774 \\ 0 & -1.6330 & 0.8165 \\ 0 & 0 & 1 \\ 0 & 0 & 0.0333 \\ 0 & 0 & 0.7232 \\ 0 & 0 & 0.6899 \end{bmatrix}, \quad \mathbf{H}_2\mathbf{H}_1\mathbf{b} = \begin{bmatrix} 376 \\ -1200 \\ 2417 \\ 85 \\ 1744 \\ 1668 \end{bmatrix}. \tag{3.126}$$

The last step now uses the third Householder vector $\mathbf{v}_3$ for annihilating the subdiagonal entries of the third column of $\mathbf{H}_2\mathbf{H}_1\mathbf{A}$ by considering

$$\mathbf{a}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0.0333 \\ 0.7232 \\ 0.6899 \end{bmatrix}. \tag{3.127}$$

This gives $s_3 = ||\mathbf{a}_3||_2 = 1.4142$ and hence we get

$$\mathbf{v}_3 = \mathbf{a}_3 + s_3\mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0.0333 \\ 0.7232 \\ 0.6899 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1.4142 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2.4142 \\ 0.0332 \\ 0.7231 \\ -0.6899 \end{bmatrix}. \tag{3.128}$$

Applying the final Householder transformation with $\mathbf{H}_3 = \mathbf{I} - 2\mathbf{v}_3 \frac{\mathbf{v}_3^T}{\mathbf{v}_3^T \mathbf{v}_3}$ to the third column of $\mathbf{H}_2\mathbf{H}_1\mathbf{A}$ gives

$$\mathbf{H}_3\mathbf{H}_2\mathbf{H}_1\mathbf{A} = \begin{bmatrix} -1.7321 & 0.5774 & 0.5774 \\ 0 & -1.6330 & 0.8165 \\ 0 & 0 & -1.4142 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}, \tag{3.129}$$

and

$$\mathbf{H}_3\mathbf{H}_2\mathbf{H}_1\mathbf{b} = \begin{bmatrix} 376 \\ -1200 \\ -3417 \\ 5 \\ 3 \\ 1 \end{bmatrix} = \mathbf{Q}^T\mathbf{b} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}. \tag{3.130}$$

We now can solve the upper triangular system $\mathbf{R}\mathbf{x} = \mathbf{c}_1$ by back-substitution to obtain

$$\mathbf{x}^T = [h_1, h_2, h_3] = [1236, 1943, 2416]. \tag{3.131}$$

Finally, the minimum residual is given by $||\mathbf{r}||_2^2 = ||\mathbf{c}_2||_2^2 = 35$. □

Chapter 4

Eigenvalue Problems

In this chapter we now look at a third class of important problems in Numerical Linear Algebra, which consist in finding the eigenvalues and eigenvectors of a given $m \times m$ matrix $\mathbf{A}$, if and when they exist. As discussed in Chapter 1, numerical methods for finding eigenvalues and eigenvectors differ significantly from what one may do analytically (i.e. construction and solution of the characteristic polynomial). Instead, eigenvalue algorithms are always based on **iterative** methods.

In what follows, we first illustrate how very simply iterative methods can actually work to find specific eigenvalues and eigenvectors of a matrix $\mathbf{A}$. For simplicity, these methods assume the matrix $\mathbf{A}$ is real and symmetric, so the eigenvalues are real and symmetric too, and the eigenvectors are orthogonal. Later, we relax these conditions to construct **eigenvalue revealing** algorithms that can find all the eigenvalues, real or complex, of any matrix $\mathbf{A}$.

Before we proceed, however, let's see a few example of applied mathematical problems where we want to find the eigenvalues of a matrix.

1. Eigenvalue problems in applied mathematics

The following examples are very basic examples that come up in simple ODE and PDE problems, that you may encounter in AM 212A and AM 214 for instance. Other more complex examples come up all the time in fluid dynamics, control theory, etc.

1.1. A simple Dynamical Systems problem

Consider the set of m nonlinear autonomous ODEs for m variables written as

$$\dot{x}_i = f_i(\mathbf{x}) \text{ for } i = 1 \dots m \tag{4.1}$$

where $\mathbf{x} = (x_1, x_2, \dots, x_m)^T$, and the functions f_i are any nonlinear function of the coefficients of $\mathbf{x}$. Suppose a fixed point of this system is known, for which $f_i(\mathbf{x}_\star) = 0$ for all i. Then, to study the stability of this fixed point, we consider a small displacement $\boldsymbol{\epsilon}$ away from it such that

$$f_i(\mathbf{x}_\star + \boldsymbol{\epsilon}) = f_i(\mathbf{x}_\star) + \sum_{j=1}^{m} \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}_\star} \epsilon_j = \sum_{j=1}^{m} \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}_\star} \epsilon_j \tag{4.2}$$

The ODE system becomes, near the fixed point,

$$\dot{\epsilon}_i = \sum_{j=1}^{m} \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}_\star} \epsilon_j \text{ for } i = 1 \dots m \tag{4.3}$$

or in other words

$$\dot{\boldsymbol{\epsilon}} = \mathbf{J}\boldsymbol{\epsilon} \tag{4.4}$$

where $\mathbf{J}$ is the Jacobian matrix of the original system at the fixed point. This is a simple linear system now, and we can look for solutions of the kind $\epsilon_i \propto e^{\lambda t}$, which implies solving for the value(s) of λ for which

$$\mathbf{J}\boldsymbol{\epsilon} = \lambda\boldsymbol{\epsilon} \tag{4.5}$$

If any of the eigenvalues λ has a positive real part, then the fixed point is unstable.

1.2. A simple PDE problem

Suppose you want to solve the diffusion equation problem

$$\frac{\partial f}{\partial t} = \frac{\partial}{\partial x}\left[D(x)\frac{\partial f}{\partial x}\right] \tag{4.6}$$

This problem is slightly more complicated than usual because the diffusion coefficient is a function of x. The first step would consist in looking for separable solutions of the kind

$$f(x,t) = A(x)B(t) \tag{4.7}$$

where it is easy to show that

$$\frac{dB}{dt} = -\lambda B \tag{4.8}$$

$$\frac{d}{dx}\left[D(x)\frac{dA}{dx}\right] = -\lambda A \tag{4.9}$$

where, on physical grounds, we can argue that $\lambda \geq 0$. If the domain is periodic, say of period 2π, we can expand the solution $A(x)$ and the diffusion coefficient $D(x)$ in Fourier modes as

$$A(x) = \sum_m a_m e^{imx} \text{ and } D(x) = \sum_m d_m e^{imx} \tag{4.10}$$

where the $\{d_m\}$ are known, but the $\{a_m\}$ are not. The equation for A becomes

$$\frac{d}{dx}\left[\sum_n d_n e^{inx} \sum_m i m a_m e^{imx}\right] = -\lambda \sum_k a_k e^{ikx} \tag{4.11}$$

and then

$$\sum_{m,n} m(m+n) d_n a_m e^{i(m+n)x} = \lambda \sum_k a_k e^{ikx} \tag{4.12}$$

Projecting onto the mode e^{ikx} we then get

$$\sum_m mkd_{k-m}a_m \equiv \sum_m B_{km}a_m = \lambda a_k \tag{4.13}$$

or, in other words, we have another matrix eigenvalue problem $\mathbf{Bv} = \lambda\mathbf{v}$ where the coefficients of the matrix $\mathbf{B}$ were given above, and the elements of $\mathbf{v}$ are the Fourier coefficients $\{a_n\}$. The solutions to that problem yield both the desired λs and the eigenmodes $A(x)$, which can then be used to construct the solution to the PDE.

Many other examples of eigenvalue problems exist. You are unlikely to go through your PhD without having to solve at least one!

1.3. Localizing Eigenvalues: Gershgorin Theorem

For some purposes it suffices to know crude information on eigenvalues, instead of determining their values exactly. For example, we might merely wish to know rough estimations of their *locations*, such as bounding circles or disks. The simplest such "bound" can be obtained as

$$\rho(\mathbf{A}) \leq ||\mathbf{A}||. \tag{4.14}$$

This can be easily shown if we take λ to be $|\lambda| = \rho(\mathbf{A})$, and if we let $\mathbf{x}$ be an associated eigenvector $||\mathbf{x}|| = 1$ (recall we can always normalize eigenvectors!). Then

$$\rho(\mathbf{A}) = |\lambda| = ||\lambda\mathbf{x}|| = ||\mathbf{Ax}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}|| = ||\mathbf{A}||. \tag{4.15}$$

A more accurate way of locating eigenvalues is given by *Gershgorin's Theorem* which is stated as the following:

Theorem: (Gershgorin's Theorem) Let $\mathbf{A} = \{a_{ij}\}$ be an $n \times n$ matrix and let λ be an eigenvalue of $\mathbf{A}$. Then λ belongs to one of the circles Z_i given by

$$Z_k = \{z \in \mathbb{R} \text{ or } \mathbb{C} : |z - a_{kk}| \leq r_k\}, \tag{4.16}$$

where

$$r_k = \sum_{j=1, j\neq k}^{n} |a_{kj}|, k = 1, \cdots, n. \tag{4.17}$$

Moreover, if m of the circles form a connected set S, disjoint from the remaining $n - m$ circles, then S contains exactly m of the eigenvalues of $\mathbf{A}$, counted according to their algebraic multiplicity.

Proof: Let $\mathbf{Ax} = \lambda\mathbf{x}$. Let k be the subscript of a component of $\mathbf{x}$ such that $|x_k| = \max_i |x_i| = ||\mathbf{x}||$, then we see that the k-th component satisfies

$$\lambda x_k = \sum_{j=1}^{n} a_{kj}x_j, \tag{4.18}$$

so that

$$(\lambda - a_{kk})x_k = \sum_{j=1, j\neq k}^{n} a_{kj}x_j. \tag{4.19}$$

Therefore

$$|\lambda - a_{kk}| \cdot |x_k| \leq \sum_{j=1, j\neq k} |a_{kj}| \cdot |x_j| \leq \sum_{j=1, j\neq k} |a_{kj}| \cdot ||x||. \tag{4.20}$$

□

Example: Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -4 \end{bmatrix}. \tag{4.21}$$

Then the eigenvalues must be contained in the circles

$$Z_1 : \quad |\lambda - 4| \leq 1 + 0 = 1, \tag{4.22}$$

$$Z_2 : \quad |\lambda| \leq 1 + 1 = 2, \tag{4.23}$$

$$Z_3 : \quad |\lambda + 4| \leq 1 + 1 = 2. \tag{4.24}$$

Note that Z_1 is disjoint from $Z_2 \cup Z_3$, therefore there exists a single eigenvalue in Z_1. Indeed, if we compute the true eigenvalues, we get

$$\lambda(\mathbf{A}) = \{-3.76010, -0.442931, 4.20303\}. \tag{4.25}$$

□

2. Invariant Transformations

As before we seek for a simpler form whose eigenvalues and eigenvectors are determined in easier ways. To do this we need to identify what types of transformations leave eigenvalues (or eigenvectors) unchanged or easily recoverable, and for what types of matrices the eigenvalues (or eigenvectors) are easily determined.

- Shift: A shift subtracts a constant scalar σ from each diagonal entry of a matrix, effectively shifting the origin.

  $$\mathbf{Ax} = \lambda\mathbf{x} \Longrightarrow (\mathbf{A} - \sigma\mathbf{I})\mathbf{x} = (\lambda - \sigma)\mathbf{x}. \tag{4.26}$$

  Thus the eigenvalues of the matrix $\mathbf{A} - \sigma\mathbf{I}$ are translated, or shifted, from those of $\mathbf{A}$ by σ, but the eigenvectors are unchanged.

- Inversion: If $\mathbf{A}$ is nonsingular and $\mathbf{Ax} = \lambda\mathbf{x}$ with $\mathbf{x} \neq \mathbf{0}$, then

  $$\mathbf{A}^{-1}\mathbf{x} = \frac{1}{\lambda}\mathbf{x}. \tag{4.27}$$

  Thus the eigenvalues of $\mathbf{A}^{-1}$ are reciprocals of the eigenvalues of $\mathbf{A}$, and the eigenvectors are unchanged.

- Powers: Raising power of a matrix also raises the same power of the eigenvalues, but keeps the eigenvectors unchanged.

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \Longrightarrow \mathbf{A}^2\mathbf{x} = \lambda^2\mathbf{x} \Longrightarrow \cdots \Longrightarrow \mathbf{A}^k\mathbf{x} = \lambda^k\mathbf{x}. \tag{4.28}$$

- Polynomials: More generally, if

$$p(t) = c_0 + c_1 t + c_2 t^2 + \cdots c_k t^k \tag{4.29}$$

  is a polynomial of degree k, then we define

$$p(\mathbf{A}) = c_0\mathbf{I} + c_1\mathbf{A} + c_2\mathbf{A}^2 + \cdots c_k\mathbf{A}^k. \tag{4.30}$$

  Now if $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ then $p(\mathbf{A})\mathbf{x} = p(\lambda)\mathbf{x}$.

- Similarity: We already have seen this in Eq. 1.39 – Eq. 1.43.

- Transpose: $\mathbf{A}$ and $\mathbf{A}^T$ have the same eigenvalues. This can be proved by realizing that

$$(\mathbf{A} - \lambda\mathbf{I})^T = \mathbf{A}^T - \lambda\mathbf{I}. \tag{4.31}$$

  Since determinant is invariant under matrix transposes (i.e., $\det(\mathbf{B}) = \det(\mathbf{B}^T)$ for any matrix $\mathbf{B}$), we get

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \det\left((\mathbf{A} - \lambda\mathbf{I})^T\right) = \det\left(\mathbf{A}^T - \lambda\mathbf{I}\right), \tag{4.32}$$

  which implies that $\mathbf{A}^T - \lambda\mathbf{I}$ and $\mathbf{A} - \lambda\mathbf{I}$ have the same characteristic polynomial.

3. Iterative ideas

See Chapter 27 from the textbook

In this section, as discussed above, all matrices are assumed to be real and symmetric.

3.1. The Power Iteration

We can very easily construct a simple algorithm to reveal the eigenvector corresponding to the largest eigenvalue of a matrix. To do so, we simply apply the matrix $\mathbf{A}$ over, and over, and over again on any initial seed vector $\mathbf{x}$. By the properties of the eigenvalues and eigenvectors of real, symmetric matrices, we know that the eigenvectors $\{\mathbf{v}_i\}$, for $i = 1 \dots m$, form an orthogonal basis in which the vector $\mathbf{x}$ can be written as

$$\mathbf{x} = \sum_{i=1}^{m} \alpha_i \mathbf{v}_i \tag{4.33}$$

Then

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{m} \lambda_i \alpha_i \mathbf{v}_i \Rightarrow \mathbf{A}^n\mathbf{x} = \sum_{i=1}^{m} \lambda_i^n \alpha_i \mathbf{v}_i \tag{4.34}$$

If we call the eigenvalue with the largest norm λ_1, then

$$\mathbf{A}^n\mathbf{x} = \lambda_1^n \sum_{i=1}^{m} \left(\frac{\lambda_i}{\lambda_1}\right)^n \alpha_i \mathbf{v}_i \tag{4.35}$$

where, by construction, $|\lambda_i/\lambda_1| < 1$ for $i > 1$. As $n \to \infty$, all but the first term in that sum tend to zero, which implies that

$$\lim_{n\to\infty} \mathbf{A}^n\mathbf{x} = \lim_{n\to\infty} \lambda_1^n \alpha_1 \mathbf{v}_1 \tag{4.36}$$

which is aligned in the direction of the first eigenvector $\mathbf{v}_1$. In general, we see that the iteration yields a sequence $\{\mathbf{x}^{(n)}\} = \{\mathbf{A}^n\mathbf{x}\}$ converges to the eigenvector $\mathbf{v}_1$ with normalization

$$\mathbf{x}_n \equiv \frac{\mathbf{x}^{(n)}}{||\mathbf{x}^{(n)}||} \approx \frac{\lambda_1^n \alpha_1 \mathbf{v}_1}{||\lambda_1^n \alpha_1 \mathbf{v}_1||} = \pm\mathbf{v}_1. \tag{4.37}$$

To approximate the corresponding value λ_1, we compute

$$\lambda^{(n)} = \mathbf{x}_n^T \mathbf{A}\mathbf{x}_n \approx (\pm\mathbf{v}_1)^T \mathbf{A}(\pm\mathbf{v}_1) = (\pm\mathbf{v}_1)^T \lambda_1(\pm\mathbf{v}_1) = \lambda||\mathbf{v}_1||^2 = \lambda_1. \tag{4.38}$$

Because of the rapid increase in the norm of the vector $\mathbf{A}^n\mathbf{x}$, it is preferable to normalize the vector obtained at each iteration before applying $\mathbf{A}$ again. But the result remains otherwise unchanged. The Power Iteration algorithm is therefore simply

Algorithm: Power Iteration algorithm:

```
x=arbitrary nonzero vector with $||\mathbf{x}|| = 1$
do while $||\mathbf{r}|| >$ desired accuracy
    $\mathbf{y} = \frac{\mathbf{Ax}}{||\mathbf{Ax}||}$ !  [Calculate new normalized vector]
    $\mathbf{r} = \mathbf{y} - \mathbf{x}$ !  [Calculate difference between old and new]
    $\mathbf{x} = \mathbf{y}$ !  [Replace old by new ]
enddo
```

This algorithm takes any initial vector, and gradually rotates it until it points in the direction of $\mathbf{v}_1$ (unless by some weird and unlikely coincidence that vector is orthogonal to $\mathbf{v}_1$). It is clear from its construction that the convergence rate depends on the ratios $|\lambda_i/\lambda_1|$. If they are all small, then the algorithm converges rapidly, but otherwise the convergence can be painfully slow. In addition, this doesn't tell us what the corresponding eigenvalue is, nor what any of the other eigenvectors and eigenvalues are! But at the very least, it illustrates how simple iterations of a most basic algorithm can reveal something about the eigenspace and spectrum of a matrix. Let us now build on this method gradually.

Example: Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \tag{4.39}$$

and let's compute the dominant (i.e., largest in magnitude) eigenvalue λ_1 by power iteration. We start with an initial vector, say,

$$\mathbf{x}_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}^T. \tag{4.40}$$

We can analytically evaluate the two eigenvalues of $\mathbf{A}$ which are $\lambda_1 = 2$ and $\lambda_2 = 4$. The corresponding eigenvectors are $v_1 = \frac{1}{\sqrt{2}}[-1, \ \ 1]^T$ and $v_2 = \frac{1}{\sqrt{2}}[1, \ \ 1]^T$, respectively.

Table 1. Power iteration for the dominant eigenvalue. The normalization is obtained with l^2 norm.

k	$\mathbf{x}_k = (x_1, x_2)_k$	$\lambda^{(k)}$
0	(0, 1)	N/A
1	(0.316227766016838, 0.948683298050514)	3.600000000000000
2	(0.514495755427527, 0.857492925712544)	3.882352941176471
3	(0.613940613514921, 0.789352217376326)	3.969230769230770
4	(0.661621637086846, 0.749837855365093)	3.992217898832684
5	(0.684675461664049, 0.728848072093987)	3.998048780487805
6	(0.695973285238859, 0.718067675246442)	3.999511837930193
7	(0.701561099839534, 0.712609306136219)	3.999877937137626
8	(0.704339271660849, 0.709863501242503)	3.999969482887529
9	(0.705724367193419, 0.708486497789088)	3.999992370634572

We see from Table 1 that the k-th lambda value $\lambda^{(k)}$ converges to the dominant eigenvalue $\lambda_2 = 4$. □

Remark: Power iteration works well in practice, but it can fail for a number of reasons:

- If the initial guess vector $\mathbf{x} = \mathbf{x}_0$ has no component in the dominant eigenvector $\mathbf{v}_1$ that corresponds to λ_1, i.e., $\alpha_1 = 0$ in Eq. (4.33), then the iteration doesn't work. One does not need to worry about this case too much as the probability this may happen is extremely unlikely if $\mathbf{x}_0$ is randomly chosen. Moreover, in practice, roundoff errors naturally help introduce such a component.

- If there are multiple eigenvalues whose modulus are the same and maximum, the iteration may converge to a vector that is a linear combination of the corresponding multiple eigenvectors.

- For a real matrix and real initial vector, the iteration cannot converge to a complex vector.

□

3.2. The Inverse Iteration

The Inverse Iteration is a pretty powerful method that addresses two of the problems of the Power Iteration. It accelerates the convergence, and can find eigenvectors other than the one corresponding to the fastest-growing eigenvalue. There is a price to pay, however, which is that we have to have an estimate of the eigenvalue corresponding to the eigenvector we are looking for.

The Inverse Iteration algorithm is based on the realization that for any μ that is not exactly equal to one of the eigenvalues, then

- The eigenvectors of $(\mathbf{A}-\mu\mathbf{I})^{-1}$ are the same as the eigenvectors of $\mathbf{A}$
- The eigenvalues of $(\mathbf{A}-\mu\mathbf{I})^{-1}$ corresponding to each eigenvector $\mathbf{v}_i$ are $(\lambda_i-\mu)^{-1}$ where λ_i is the eigenvalue of $\mathbf{A}$ corresponding to $\mathbf{v}_i$.

To prove these statements, we start with

$$\begin{aligned} \mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i \Rightarrow (\mathbf{A}-\mu\mathbf{I})\mathbf{v}_i = (\lambda_i-\mu)\mathbf{v}_i \\ \Rightarrow \mathbf{v}_i = (\lambda_i-\mu)(\mathbf{A}-\mu\mathbf{I})^{-1}\mathbf{v}_i \Rightarrow (\mathbf{A}-\mu\mathbf{I})^{-1}\mathbf{v}_i = \frac{1}{\lambda_i-\mu}\mathbf{v}_i \end{aligned} \quad (4.41)$$

We can now use this result to our advantage. Suppose that we have a rough estimate μ of *any arbitrary* (as opposed to the largest) eigenvalue λ_i. Then, by construction, $\frac{1}{\lambda_i-\mu}$ is the largest eigenvalue of $(\mathbf{A}-\mu\mathbf{I})^{-1}$. We can then do a Power Iteration on the matrix $\mathbf{B}=(\mathbf{A}-\mu\mathbf{I})^{-1}$ to find the corresponding eigenvector $\mathbf{v}_i$, which happens to be an eigenvector of both $\mathbf{A}$ and $(\mathbf{A}-\mu\mathbf{I})^{-1}$.

Algorithm: Inverse Iteration algorithm

```
Initialize $\mu$
$\mathbf{B}=(\mathbf{A}-\mu\mathbf{I})^{-1}$
do while $||\mathbf{r}||>$ desired accuracy
    $\mathbf{y}=\frac{\mathbf{Bx}}{||\mathbf{Bx}||}$  !  [Calculate new normalized vector]
    $\mathbf{r}=\mathbf{y}-\mathbf{x}$  !  [Calculate difference between old and new]
    $\mathbf{x}=\mathbf{y}$  !  [Replace old by new ]
enddo
```

The disadvantage of this algorithm is that it requires a good estimate of a certain eigenvalue to get good convergence on its corresponding eigenvector. In what follows, we now learn a new trick to speed up convergence *very* significantly.

Example: Consider again the matrix

$$\mathbf{A}=\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \quad (4.42)$$

and let's find $\lambda_1=2$ this time using inverse iteration. We take the same initial vector

$$\mathbf{x}_0=\begin{bmatrix} 0 & 1 \end{bmatrix}^T . \quad (4.43)$$

Table 2. Inverse iteration for the smallest eigenvalue with different values of shift factor $\mu = 1$ and 1.9. The normalization is obtained with L_2 norm.

k	$\mathbf{x}_k = (x_1, x_2)_k$	$\lambda^{(k)}$ with $\mu = 1.0$
0	(0, 1)	N/A
1	(-0.447213595499958, 0.894427190999916)	2.200000000000000
2	(-0.624695047554424, 0.780868809443030)	2.024390243902439
3	(-0.680451099367278, 0.732793491626299)	2.002739726027397
4	(-0.698323852075327, 0.715781948377211)	2.000304785126485
5	(-0.704190913994984, 0.710010673614777)	2.000033869602032
6	(-0.706136148677213, 0.708076083151601)	2.000003763345764
7	(-0.706783384584535, 0.707430029950121)	2.000000418150229
8	(-0.706998998735688, 0.707214547210912)	2.000000046461145
9	(-0.707070855527723, 0.707142705020206)	2.000000005162351
k	$\mathbf{x}_k = (x_1, x_2)_k$	$\lambda^{(k)}$ with $\mu = 1.9$
0	(0, 1)	N/A
1	(-0.672672793996313, 0.739940073395944)	2.004524886877828
2	(-0.705501550645646, 0.708708375875853)	2.000010283728057
3	(-0.707030423886725, 0.707183130241776)	2.000000023319231
4	(-0.707103145311582, 0.707110417042818)	2.000000000052878
5	(-0.707106608050068, 0.707106954322984)	2.000000000000120
6	(-0.707106772941954, 0.707106789431141)	2.000000000000000
7	(-0.707106780793948, 0.707106781579147)	2.000000000000000
8	(-0.707106781167852, 0.707106781205243)	2.000000000000000
9	(-0.707106781185657, 0.707106781187438)	2.000000000000000

As shown in Table 2 the k-th iterated eigenvalue $\lambda^{(k)}$ converges to $\lambda_1 = 2$ of the matrix $\mathbf{A}$. It is also shown that the iteration with $\mu = 1.9$ which is closer to $\lambda_1 = 2$ is faster than than the case with with $\mu = 1$, converging in 6 steps.
□

What does this tell us? This suggests that we could now use Gershgorin theorem to estimate the locations of target eigenvalues, and choose μ accordingly in order to accelerate the search for *all* eigenvalues.

3.3. The Rayleigh Quotient iteration

3.3.1. The Rayleigh Quotient. A Rayleigh Quotient is a very useful concept that comes up both in Linear Algebra and in methods for PDEs (AM 212A). As we shall see, it can help estimate an eigenvalue, as long as the eigenvector is known.

Definition: Given an $m \times m$ matrix $\mathbf{A}$, the Rayleigh Quotient is a function from $\mathbb{R}^m$ to $\mathbb{R}$ which, for each vector $\mathbf{x}$, returns the scalar function

$$r(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \tag{4.44}$$

It is very easy to show that if $\mathbf{x}$ is the eigenvector $\mathbf{v}_i$ of matrix $\mathbf{A}$, then $r(\mathbf{v}_i) = \lambda_i$, where λ_i is the eigenvalue corresponding to $\mathbf{v}_i$.

What is much more interesting, however, is that the eigenvectors are actually fixed points of the function $r(\mathbf{x})$, and to be precise, they are all *local minima* of this function. To prove that they are fixed points, let's evaluate ∇r:

$$\nabla r = \left(\frac{\partial r}{\partial x_1}, \dots, \frac{\partial r}{\partial x_m}\right)^T \tag{4.45}$$

Each individual component of this vector is

$$\begin{aligned}
\frac{\partial r}{\partial x_i} &= \frac{\partial}{\partial x_i}\frac{\sum_{j,k} x_j A_{jk} x_k}{\sum_j x_j^2} && (4.46)\\
&= \frac{1}{||\mathbf{x}||^2}\frac{\partial}{\partial x_i}\sum_{j,k} x_j A_{jk} x_k - \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{||\mathbf{x}||^4}\frac{\partial}{\partial x_i}\sum_j x_j^2 && (4.47)\\
&= \frac{1}{||\mathbf{x}||^2}\left(\sum_k A_{ik}x_k + \sum_j A_{ji}x_j\right) - \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{||\mathbf{x}||^4}(2x_i) && (4.48)
\end{aligned}$$

Recalling that we are working with real symmetric matrices, $A_{ji} = A_{ij}$ so $\sum_k A_{ik}x_k = \sum_j A_{ji}x_j$. This implies

$$\nabla r = \frac{2}{\mathbf{x}^T\mathbf{x}}\left(\mathbf{A}\mathbf{x} - r(\mathbf{x})\mathbf{x}\right) \tag{4.49}$$

This is true in general, but if $\mathbf{x}$ is one of the eigenvectors, then $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$, and $r(\mathbf{v}_i) = \lambda_i$. As a result

$$\nabla r(\mathbf{v}_i) = 0 \tag{4.50}$$

□

Showing that the Rayleigh quotient actually has a local minimum at these points (rather than a maximum or a stationary point) is a little more involved, but can be done. The implications of this conclusion are quite deep, because they imply that if we have an *estimate* for the eigenvector $\mathbf{v}_i$, i.e, we know that $\mathbf{v} = \mathbf{v}_i + \boldsymbol{\epsilon}$ where $||\boldsymbol{\epsilon}||$ is small, then a Taylor expansion of r near $\mathbf{v}_i$ shows that

$$r(\mathbf{v}) = r(\mathbf{v}_i + \boldsymbol{\epsilon}) = r(\mathbf{v}_i) + \boldsymbol{\epsilon}\cdot\nabla r + O(||\boldsymbol{\epsilon}||^2) = \lambda_i + O(||\boldsymbol{\epsilon}||^2) \tag{4.51}$$

or in other words, if we have an estimate for an eigenvector $\mathbf{v}_i$ that is accurate to within $O(\epsilon)$, then we can get an estimate for its corresponding eigenvalue λ_i that is accurate within $O(\epsilon)^2$! This seems almost too good to be true, but there are no tricks here – it really works.

3.3.2. The Rayleigh Quotient iteration Based on what we just found, there is a very simple way of accelerating the Inverse Iteration algorithm significantly, noting that the Inverse Iteration can help converge to an eigenvector knowing an eigenvalue, while calculation of the Rayleigh Quotient helps converge to an eigenvalue given an eigenvector

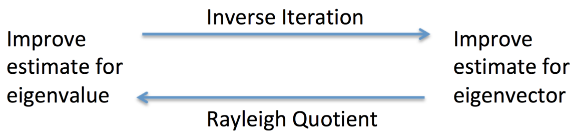

Inverse Iteration
Improve estimate for eigenvalue
Improve estimate for eigenvector
Rayleigh Quotient

Figure 1. The Rayleigh-Quotient Iteration

The algorithm itself is extremely simple:

Algorithm: Rayleigh Quotient Iteration algorithm

```
Initialize guess eigenvector v, with $||\mathbf{v}|| = 1$
Initialize $\lambda = r(\mathbf{v})$
do while $||\mathbf{r}|| >$ desired accuracy
    Solve $(\mathbf{A} - \lambda\mathbf{I})\mathbf{y} = \mathbf{v}$ !  [One step of inverse iteration]
    $\mathbf{y} = \frac{\mathbf{y}}{||\mathbf{y}||}$ !  [Calculate new normalized vector]
    $\mathbf{r} = \mathbf{y} - \mathbf{v}$
    $\lambda = r(\mathbf{y})$ !  [Calculate new eigenvalue]
    $\mathbf{v} = \mathbf{y}$ !  [Replace old by new ]
enddo
```

The convergence of this algorithm is phenomenally fast (see examples), and the algorithm converges to the eigenvector that is the generally the closest in direction to the initial guess $\mathbf{v}$. A theorem (that we will not prove) establishes the following:

Theorem: The Rayleigh Quotient Iteration converges to an eigenvector/eigenvalue pair $(\lambda_i, \mathbf{v}_i)$ for almost any possible initial guess (i.e. for all initial guesses except a set of measure 0). When it converges, then the convergence is ultimately cubic in the sense that, after a certain initial number of iterations,

- $||\mathbf{v}^{(n+1)} - \mathbf{v}_i|| = O(||\mathbf{v}^{(n)} - \mathbf{v}_i||^3)$
- $|\lambda^{(n+1)} - \lambda_i| = O(|\lambda^{(n)} - \lambda_i|^3)$

where $\lambda^{(n)}$ is the value of λ after n iterations, and $\mathbf{v}^{(n)}$ is the vector $\mathbf{v}$ after n iterations.

Furthermore, one can easily find many of the eigenvectors simply by trying different initial vectors $\mathbf{v}$ on the unit sphere. In fact, one can work on an entire basis at the same time using this algorithm, to converge to all of the eigenvectors at once (see next lecture on how to do this).

4. The QR algorithm without shifts

See Chapter 28 of the textbook

In this section again we shall assume that the matrix $\mathbf{A}$ is $m \times m$, real and symmetric.

4.1. Simultaneous Iterations algorithm

Suppose we now want to construct an algorithm that can find *all* the eigenvectors and eigenvalues of the matrix $\mathbf{A}$ at the same time. A naive approach, based on the Power Iteration[1], would simply be to apply the matrix $\mathbf{A}$ repeatedly to a set of initially orthogonal vectors (chosen so to make sure they span the whole space and are linearly independent), such as the column vectors of $\mathbf{I}$ for instance. But that doesn't quite work: to see why, let's look at an example in 2D. Let $\mathbf{x}_1$ and $\mathbf{x}_2$ be the two initial vectors, which can both be written in the basis of the eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$:

$$\begin{aligned}\mathbf{x}_1 &= \alpha_{11}\mathbf{v}_1 + \alpha_{12}\mathbf{v}_2 \\ \mathbf{x}_2 &= \alpha_{21}\mathbf{v}_1 + \alpha_{22}\mathbf{v}_2\end{aligned} \tag{4.52}$$

After applying $\mathbf{A}$ once, we get

$$\begin{aligned}\mathbf{A}\mathbf{x}_1 &= \alpha_{11}\lambda_1\mathbf{v}_1 + \alpha_{12}\lambda_2\mathbf{v}_2 \\ \mathbf{A}\mathbf{x}_2 &= \alpha_{21}\lambda_1\mathbf{v}_1 + \alpha_{22}\lambda_2\mathbf{v}_2\end{aligned} \tag{4.53}$$

and after n times, when n is very large,

$$\begin{aligned}\mathbf{A}^n\mathbf{x}_1 &= \alpha_{11}\lambda_1^n\mathbf{v}_1 + \alpha_{12}\lambda_2^n\mathbf{v}_2 \simeq \alpha_{11}\lambda_1^n\mathbf{v}_1 + \dots \\ \mathbf{A}^n\mathbf{x}_2 &= \alpha_{21}\lambda_1^n\mathbf{v}_1 + \alpha_{22}\lambda_2^n\mathbf{v}_2 \simeq \alpha_{21}\lambda_1^n\mathbf{v}_1 + \dots\end{aligned} \tag{4.54}$$

so unfortunately *both* vectors turn into the direction of $\mathbf{v}_1$! However, if we remember that the set of eigenvectors of a real symmetric matrix is orthogonal, we can find $\mathbf{v}_2$ simply by requiring that it should be orthogonal to $\mathbf{v}_1$, once the latter is known.

In more than two dimensions it's more complicated, since there is an infinite number of ways of constructing an orthogonal basis whose first vector is known (simply rotating the axes...). However, if we orthogonalize the basis of vectors $\{\mathbf{x}_i\}$ at *every step*, we effectively force $\mathbf{x}_2$ to be orthogonal to $\mathbf{x}_1$, so its evolution becomes dominated by the eigenvector with the next largest eigenvalue, $\mathbf{v}_2$,

[1] In the next lecture, we will improve on this to use the Rayleigh Quotient Iteration, but for now, this is much easier to understand.

and similarly we force $\mathbf{x}_3$ to be orthogonal to both $\mathbf{x}_1$ and $\mathbf{x}_2$, so its evolution becomes dominated by the eigenvector with the *next* largest eigenvalue, $\mathbf{v}_3$, etc.. As a result, the basis of vectors $\{\mathbf{x}_i\}$ gradually rotates *as a whole*, axis by axis, until it becomes the basis $\{\mathbf{v}_i\}$.

Conveniently, we have *just* learned a technique to orthogonalize a basis: this is the QR decomposition, which takes a set of vectors (listed as column vectors of a matrix $\mathbf{A}$) and returns $\mathbf{QR} = \mathbf{A}$, in which the column vectors of $\mathbf{Q}$ form an orthonormal basis whose first vector is aligned with $\mathbf{a}_1$, whose second vector lies in the plane spanned by $\mathbf{a}_1$ and $\mathbf{a}_2$, etc.. We can then combine the QR algorithm and the Power Iteration algorithm into what is called the **Simultaneous Iterations** algorithm.

Algorithm: Simultaneous Iterations Algorithm:

```
$\mathbf{Q} = \mathbf{I}$ !  [Initialize matrix]
do while error remains too large
    $\mathbf{Z} = \mathbf{AQ}$ !  [Multiply $\mathbf{Q}$ by $\mathbf{A}$]
    $\mathbf{QR} = \mathbf{Z}$ !  [Compute QR factorization of $\mathbf{Z}$, get new $\mathbf{Q}$]
enddo
```

Note: The convergence condition *while the error remains too large* can be interpreted in different ways, but can for instance involve computing the eigenvalue estimate at step n, and then again at step $n+1$, and requiring that the change in these estimates between the two steps be smaller than a small number, to be decided by the user.

After a sufficient number of iterations, this algorithm gradually transforms the matrix $\mathbf{Q}$ into the matrix of eigenvectors $\mathbf{V} = [\mathbf{v}_1|\mathbf{v}_2|\dots|\mathbf{v}_m]$. This then also implies that

$$\lim_{n\to\infty} \mathbf{Q}^{(n)T}\mathbf{A}\mathbf{Q}^{(n)} = \mathbf{V}^T\mathbf{A}\mathbf{V} = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{pmatrix} \equiv \mathbf{D}_\lambda \tag{4.55}$$

where $\mathbf{Q}^{(n)}$ is the $\mathbf{Q}$ matrix at step n.

Note that since we are orthogonalizing the matrix $\mathbf{Z}$ at every step (and not $\mathbf{A}^n$), it is not entirely obvious that $\mathbf{Q}^{(n)}$ should necessarily contain an orthonormal basis associated with $\mathbf{A}^n\mathbf{I}$, which is what we were aiming to create. We can however prove this easily by showing that

$$\mathbf{A}^n\mathbf{I} = \mathbf{Q}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)}\dots\mathbf{R}^{(1)} \tag{4.56}$$

where $\mathbf{R}^{(n)}$ is the R-factor in the QR factorization at step n.

Proof: We can easily prove this statement using recursion. Given the algorithm above, at the first step, we have $\mathbf{Q}^{(1)}\mathbf{R}^{(1)} = \mathbf{AI}$, which is in the correct form.

Then, if we assume that, at step $n-1$, $\mathbf{A}^{n-1}\mathbf{I} = \mathbf{Q}^{(n-1)}\mathbf{R}^{(n-1)} \ldots \mathbf{R}^{(1)}$, then

$$\begin{aligned} \mathbf{A}^n\mathbf{I} &= \mathbf{A}\mathbf{A}^{n-1}\mathbf{I} = \mathbf{A}\mathbf{Q}^{(n-1)}\mathbf{R}^{(n-1)} \ldots \mathbf{R}^{(1)} \qquad (4.57) \\ &= \mathbf{Z}^{(n)}\mathbf{R}^{(n-1)} \ldots \mathbf{R}^{(1)} = \mathbf{Q}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)} \ldots \mathbf{R}^{(1)} \qquad (4.58) \end{aligned}$$

as required. Since the product of upper triangular matrices is another upper triangular matrix, we have therefore shown that the expression

$$\mathbf{A}^n = \mathbf{Q}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)} \ldots \mathbf{R}^{(1)} \qquad (4.59)$$

is in fact a QR factorization of $\mathbf{A}^n$, and the factor $\mathbf{Q}^{(n)}$ therefore contains a set of orthonormal vectors forming a basis for the span of $\mathbf{A}^n$, whose first element is parallel to the first element of $\mathbf{A}^n$. □

The convergence rate of this algorithm is similar to that of the basic Power Iteration algorithm, and is therefore not that great. It is interesting to note, however that it converges on the first and last eigenvalues at roughly the same rate, but slower on intermediate eigenvalues. This result may be surprising at first but will be clarified in the next lecture. We will also see the next lecture how to improve convergence! In the meantime, we will now learn another algorithm that is equivalent in all respects to the Simultaneous Iterations algorithm, but looks a lot simpler (if that is even possible), and is remarkably elegant. This is the QR algorithm (not to be mixed up with QR decompositions, which it uses).

4.2. The QR algorithm without shifts

The QR algorithm goes as follows:

Algorithm: QR algorithm:

```
do while error remains too large
    $\mathbf{QR} = \mathbf{A}$ !  [Compute QR factorization of $\mathbf{A}$]
    $\mathbf{A} = \mathbf{RQ}$ !  [Replace $\mathbf{A}$ by product $\mathbf{RQ}$]
enddo
```

It is insanely simple: just compute a QR factorization of $\mathbf{A}$, then recompute the product[2] switching $\mathbf{R}$ and $\mathbf{Q}$. Then repeat with the new matrix.

[2] Of course, creating $\mathbf{R}$ and $\mathbf{Q}$ and then multiplying them is not very efficient. It is best to work only with the Householder vectors returned by the QR factorization scheme, see previous lecture.

We are now about to prove that this algorithm is entirely equivalent to the Simultaneous Iterations algorithm, and in particular, that

- This algorithm gradually transforms the initial matrix $\mathbf{A}$ into a diagonal matrix containing the eigenvalues in its diagonal, ordered in norm from largest to smallest
- The matrix $\mathbf{Q}$ of the QR factorization of $\mathbf{A}$ gradually transforms into the matrix that contains all orthogonal and normalized eigenvectors of $\mathbf{A}$

Proof: To help with the proof, we will label the matrices with superscripts (n) to describe their values at the n-th iteration. Also, the matrices of the QR factorization associated with the QR algorithms will be marked as $\mathbf{Q}$ and $\mathbf{R}$, and those associated with the SI algorithm will be marked as $\check{\mathbf{Q}}$ and $\check{\mathbf{R}}$. The initial values are written with superscript (0). In these notations, the two algorithms become:

- QR algorithm: $\mathbf{A}^{(0)} = \mathbf{A}$, $\mathbf{Q}^{(n)}\mathbf{R}^{(n)} = \mathbf{A}^{(n-1)}$ and $\mathbf{A}^{(n)} = \mathbf{R}^{(n)}\mathbf{Q}^{(n)}$.
- SI algorithm: $\check{\mathbf{Q}}^{(0)} = \mathbf{I}$, $\mathbf{Z} = \mathbf{A}\check{\mathbf{Q}}^{(n-1)}$ and $\check{\mathbf{Q}}^{(n)}\check{\mathbf{R}}^{(n)} = \mathbf{Z}$.

We now prove a number of intermediate results:

1. $\mathbf{R}^{(n)} = \check{\mathbf{R}}^{(n)}$
2. $\check{\mathbf{Q}}^{(n)} = \mathbf{Q}^{(1)} \dots \mathbf{Q}^{(n)}$
3. $\mathbf{A}^{(n)} = \check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n)}$
4. $\mathbf{A}^{n} = \check{\mathbf{Q}}^{(n)}\mathbf{R}^{(n)} \dots \mathbf{R}^{(1)} = \mathbf{Q}^{(1)} \dots \mathbf{Q}^{(n)}\mathbf{R}^{(n)} \dots \mathbf{R}^{(1)}$

To do so, we use recursion, i.e., first verify that it is true at step 0, then show that it is true at step (n) provided it is true at step $(n-1)$.

At step 0: Since $\check{\mathbf{R}}^{(0)}$, $\check{\mathbf{Q}}^{(0)}$ $\mathbf{R}^{(0)}$ and $\mathbf{Q}^{(0)}$ are not needed, we simply define them to be the identity. Then we trivially have proved 1., 2., 3. and 4. (assuming that we do not evaluate the products in 2. and 4. since $n = 0$, and merely assume the right-hand-sides are the identity.

At step (n), assuming 1., 2., 3. and 4. hold at all steps until $(n-1)$:

While analysing the SI algorithm we have already shown that

$$\mathbf{A}^{n} = \check{\mathbf{Q}}^{(n)}\check{\mathbf{R}}^{(n)}\check{\mathbf{R}}^{(n-1)} \dots \check{\mathbf{R}}^{(1)} \tag{4.60}$$

which, using 1. also implies $\mathbf{A}^{n} = \check{\mathbf{Q}}^{(n)}\check{\mathbf{R}}^{(n)}\mathbf{R}^{(n-1)} \dots \mathbf{R}^{(1)}$.

But from the SI algorithm, we have $\check{\mathbf{Q}}^{(n)}\check{\mathbf{R}}^{(n)} = \mathbf{A}\check{\mathbf{Q}}^{(n-1)}$, which from 3. can be written as $\mathbf{A}\check{\mathbf{Q}}^{(n-1)} = \check{\mathbf{Q}}^{(n-1)}\mathbf{A}^{(n-1)}$ (recalling that the Q matrices are all orthogonal), so

$$\begin{aligned} \mathbf{A}^{n} &= \check{\mathbf{Q}}^{(n-1)}\mathbf{A}^{(n-1)}\mathbf{R}^{(n-1)} \dots \mathbf{R}^{(1)} \\ &= \check{\mathbf{Q}}^{(n-1)}\mathbf{Q}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)} \dots \mathbf{R}^{(1)} \\ = \mathbf{Q}^{(1)} \dots \mathbf{Q}^{(n-1)}\mathbf{Q}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)} \dots \mathbf{R}^{(1)} &= \check{\mathbf{Q}}^{(n)}\mathbf{R}^{(n)}\mathbf{R}^{(n-1)} \dots \mathbf{R}^{(1)} \end{aligned} \tag{4.61}$$

where we got to the second line using the QR algorithm, and to the last one using 4. at step $n-1$. This expression then proves 4 at step n. Identifying the two expressions for $\mathbf{A}^n$, we also prove 1., that $\mathbf{R}^{(n)} = \check{\mathbf{R}}^{(n)}$, and also 2., since

$$\check{\mathbf{Q}}^{(n)} = \check{\mathbf{Q}}^{(n-1)}\mathbf{Q}^{(n)} = \mathbf{Q}^{(1)} \dots \mathbf{Q}^{(n-1)}\mathbf{Q}^{(n)} \tag{4.62}$$

To prove the last remaining expression, namely 3. at step n, we begin with

$$\mathbf{A}^{(n)} = \mathbf{R}^{(n)}\mathbf{Q}^{(n)} = \mathbf{Q}^{(n)T}\mathbf{A}^{(n-1)}\mathbf{Q}^{(n)} \tag{4.63}$$

using both steps of the QR algorithm. Then we use $\check{\mathbf{R}}^{(n)} = \check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n-1)}$ from the SI algorithm, and $\mathbf{R}^{(n)} = \mathbf{Q}^{(n)T}\mathbf{A}^{(n-1)}$ from the QR algorithm. Since $\check{\mathbf{R}}^{(n)} = \mathbf{R}^{(n)}$, we can then identify

$$\check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n-1)} = \mathbf{Q}^{(n)T}\mathbf{A}^{(n-1)} \tag{4.64}$$

Using 2. this becomes

$$\check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n)} = \mathbf{Q}^{(n)T}\mathbf{A}^{(n-1)}\mathbf{Q}^{(n)} \tag{4.65}$$

which we can now use in the earlier expression

$$\mathbf{A}^{(n)} = \mathbf{Q}^{(n)T}\mathbf{A}^{(n-1)}\mathbf{Q}^{(n)} = \check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n)} \tag{4.66}$$

which proves 3.

Where does this leave us? Well, having proved that $\mathbf{A}^{(n)} = \check{\mathbf{Q}}^{(n)T}\mathbf{A}\check{\mathbf{Q}}^{(n)}$, and because the matrices $\check{\mathbf{Q}}^{(n)}$ from the SI algorithm are known to converge to the eigenvector matrix $\mathbf{V}$, we see that $\mathbf{A}^{(n)}$ converges to the eigenvalue matrix $\mathbf{D}_\lambda$. This completes the required proof! □

It is important to realize that, as is, the QR algorithm does *not* return the eigenvectors – merely the matrix $\mathbf{D}_\lambda$. However if the eigenvectors are required, then one can get them quite easily using the analogy with the SI algorithm given in 2 – since $\check{\mathbf{Q}}^{(n)}$ converges to the eigenvector matrix $\mathbf{V}$, we merely have to compute $\check{\mathbf{Q}}^{(n)}$. To do so, we use the following algorithm instead:

Algorithm: QR algorithm, returning eigenvectors:

```
$\mathbf{V} = \mathbf{I}$
do while error remains too large
    $\mathbf{QR} = \mathbf{A}$  !  [Compute QR factorization of $\mathbf{A}$]
    $\mathbf{A} = \mathbf{RQ}$  !  [Replace $\mathbf{A}$ by product $\mathbf{RQ}$]
    $\mathbf{V} = \mathbf{VQ}$  !  [Update the eigenvector matrix]
enddo
```

Note that since the QR algorithm is equivalent to the SI algorithm, its convergence is equally slow. In some extreme cases, the algorithm fails to converge altogether, as in the following example.

Example: Let

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{4.67}$$

Trivially, it's QR decomposition is

$$\mathbf{A} = \mathbf{QR} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{4.68}$$

since $\mathbf{A}$ is itself orthogonal. So reconstructing $\mathbf{RQ}$ for the next step, we get $\mathbf{RQ} = \mathbf{IQ} = \mathbf{A}$. And so forth. As a result, $\mathbf{A}$ never converges to its diagonal eigenvalue form. The latter is however well-defined: the matrix $\mathbf{A}$ has two eigenvalues, 1 and -1, with two distinct eigenvectors, $\mathbf{v}_1 = (1/\sqrt{2}, 1/\sqrt{2})^T$ and $\mathbf{v}_2 = (1/\sqrt{2}, -1/\sqrt{2})^T$, so there is nothing particularly special about $\mathbf{A}$. □

Aside from these extreme cases, however we may wonder whether it is possible to somehow create a simultaneous version of the Rayleigh Quotient iteration to greatly speed up convergence. The answer is yes, and leads to the QR algorithm with shifts, which (with a few added tricks) has become an industry standard.

Example: To illustrate the QR iteration we will apply it to the real symmetric matrix (an easy case!),

$$\mathbf{A} = \begin{bmatrix} 2.9766 & 0.3945 & 0.4198 & 1.1159 \\ 0.3945 & 2.7328 & -0.3097 & 0.1129 \\ 0.4198 & -0.3097 & 2.5675 & 0.6079 \\ 1.1159 & 0.1129 & 0.6079 & 1.7231 \end{bmatrix}, \tag{4.69}$$

which has eigenvalues $\lambda_1 = 4$, $\lambda_2 = 3$, $\lambda_3 = 2$, $\lambda_4 = 1$. Computing its QR factorization (e.g., using Householder or Gram-Schmidt factorizations) and then forming the reverse product, we obtain

$$\mathbf{A}^{(1)} = \begin{bmatrix} 3.7703 & 0.1745 & 0.5126 & -0.3934 \\ 0.1745 & 2.7675 & -0.3872 & 0.0539 \\ 0.5126 & -0.3872 & 2.4019 & -0.1241 \\ -0.3934 & 0.0539 & -0.1241 & 1.0603 \end{bmatrix}. \tag{4.70}$$

Most of the off-diagonal entries are now smaller in magnitude and the diagonal entries are somewhat closer to the eigenvalues. Continuing for a couple of more iterations, we obtain

$$\mathbf{A}^{(2)} = \begin{bmatrix} 3.9436 & 0.0143 & 0.3046 & 0.1038 \\ 0.0143 & 2.8737 & -0.3362 & -0.0285 \\ 0.3046 & -0.3362 & 2.1785 & 0.0083 \\ 0.1038 & -0.0285 & 0.0083 & 1.0042 \end{bmatrix}, \tag{4.71}$$

and

$$\mathbf{A}^{(3)} = \begin{bmatrix} 3.9832 & -0.0356 & 0.1611 & -0.0262 \\ -0.0356 & 2.9421 & -0.2432 & 0.0098 \\ 0.1611 & -0.2432 & 2.0743 & 0.0047 \\ -0.0262 & 0.0098 & 0.0047 & 1.0003 \end{bmatrix}. \tag{4.72}$$

The off-diagonal entries are now fairly small, and the diagonal entries are quite close to the eigenvalues. Only a few more iterations would be required to compute the eigenvalues to the full accuracy shown. □

5. Improvements to the QR algorithm

See Chapter 29 of the textbook

In this section, once again, we shall assume that the matrix **A** is $m \times m$, real and symmetric.

5.1. The QR algorithm with shifts

The convergence rate of the QR algorithm is, as discussed before, in principle determined by the ratio of the first to next eigenvalues; if that ratio is large, convergence is rapid, but if that ratio is close to one, it isn't. As discussed in the previous lecture, it is very tempting to try and use a trick similar to the Rayleigh Quotient Iteration to accelerate convergence, but we run into two problems when trying to do so:

- Starting from a single vector, using the Inverse Iteration with a well-chosen shift μ given by the Rayleigh Quotient can easily home in on the eigenvector whose eigenvalue is closest to μ. However, if we did simultaneous inverse iterations on a whole initial matrix **I** in the same way we did simultaneous power iterations on **I** in the last lecture, the same shift would apply to the whole matrix, but only helps the convergence of a single vector. We may worry that it might in fact slow down the convergence for the other vectors.

- In addition, while we proved that the QR algorithm is equivalent to performing Simultaneous Power Iterations, there is no immediately apparent way of generalizing it to be able to do Simultaneous Inverse Iterations (which is what would be needed here).

By some odd stroke of luck, however, both problems can be solved at the same time with the **QR algorithm with shifts**.

The principle behind the QR algorithm with shifts derives from a very important property the Simultaneous Iterations Algorithm (which, as you recall, is equivalent to the QR algorithm).

Theorem: The Simultaneous Iterations Algorithm applied to an initial identity matrix $\mathbf{I}$, it is equivalent to an *unshifted simultaneous inverse power iteration* (i.e. a simultaneous power iteration of $\mathbf{A}^{-1}$) on a reverse permutation of the identity matrix $\mathbf{P}$.

The matrix $\mathbf{P}$ is simply a back-to-front identity matrix (shown here in 4-dimensional space for instance)

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} . \tag{4.73}$$

To see why this is the case recall that the SI algorithm writes $\mathbf{A}^n$ as

$$\mathbf{A}^n\mathbf{I} = \check{\mathbf{Q}}^{(n)}\mathbf{R} \text{ where } \mathbf{R} = \check{\mathbf{R}}^{(n)} \dots \check{\mathbf{R}}^{(1)} \tag{4.74}$$

showing that $\check{\mathbf{Q}}^{(n)}$ is the orthogonalized matrix $\mathbf{A}^n$, and the first column vector of $\check{\mathbf{Q}}^{(n)}$ is aligned with the first column vector of $\mathbf{A}^n$. That vector, by construction, gradually converges to the first eigenvector $\mathbf{v}_1$.

However, we can also take the inverse of this expression, and get

$$(\mathbf{A}^n)^{-1} = (\mathbf{A}^{-1})^n = (\check{\mathbf{Q}}^{(n)}\mathbf{R})^{-1} = \mathbf{R}^{-1}\check{\mathbf{Q}}^{(n)T} \tag{4.75}$$

Furthermore, remembering that $\mathbf{A}$ is symmetric, we know that its inverse is too (this is very easy to prove), so

$$(\mathbf{A}^{-1})^n = \mathbf{R}^{-1}\check{\mathbf{Q}}^{(n)T} = (\mathbf{R}^{-1}\check{\mathbf{Q}}^{(n)T})^T = \check{\mathbf{Q}}^{(n)}\mathbf{R}^{-T} \tag{4.76}$$

Now the matrix $\mathbf{R}^{-T}$ is *not* upper triangular – it is in fact lower triangular, but here is where the matrix $\mathbf{P}$ comes in. Noting that $\mathbf{P}^2 = \mathbf{I}$, we have

$$(\mathbf{A}^{-1})^n\mathbf{P} = [\check{\mathbf{Q}}^{(n)}\mathbf{P}][\mathbf{P}\mathbf{R}^{-T}\mathbf{P}] \tag{4.77}$$

This time, $\mathbf{P}\mathbf{R}^{-T}\mathbf{P}$ is upper triangular (think of flipping both the columns and the row of a lower triangular matrix with $\mathbf{P}$), so what we have here is an effective QR decomposition of $(\mathbf{A}^{-1})^n\mathbf{P}$. □

This shows that the matrix $\check{\mathbf{Q}}^{(n)}\mathbf{P}$ contains the orthogonalized basis of $(\mathbf{A}^{-1})^n\mathbf{P}$ in its column vectors, and that its first vector, which is actually the last column of $\check{\mathbf{Q}}^{(n)}$, is aligned with the first vector of $(\mathbf{A}^{-1})^n\mathbf{P}$. Since $(\mathbf{A}^{-1})^n\mathbf{P}$ is a simultaneous inverse iteration, its first vector picks up the direction of the *largest* eigenvalue of $\mathbf{A}^{-1}$. Since the eigenvalues of $\mathbf{A}^{-1}$ are $1/\lambda_i$ (if those of $\mathbf{A}$ are λ_i), then the largest eigenvalue of $\mathbf{A}^{-1}$ is $1/\lambda_m$. The simultaneous inverse iteration acting on $\mathbf{P}$ will therefore promote the eigenvector $\mathbf{v}_m$. This ultimately shows that the SI algorithm not only preferentially converges the first column of $\check{\mathbf{Q}}^{(n)}$ to the first eigenvector $\mathbf{v}_1$, it *also* preferentially converges the last column of $\check{\mathbf{Q}}^{(n)}$ into the last eigenvector $\mathbf{v}_m$, *at the same time!*. This now explains why the first and last eigenvalue/eigenvector pairs appear to converge faster than all the other ones.

The upshot of realizing this is that we can now shift the inverse iteration on the last column vector to accelerate it greatly, without affecting the convergence of the first eigenvector too much. This leads to the QR algorithm with shifts!

Algorithm: QR algorithm with shift (part 1):

```
do while error remains too large
    $\mu = a_{mm}$ !  [Select shift]
    $\mathbf{QR} = \mathbf{A} - \mu\mathbf{I}$ !  [Compute QR factorization of $\mathbf{A} - \mu\mathbf{I}$]
    $\mathbf{A} = \mathbf{RQ} + \mu\mathbf{I}$ !  [Replace $\mathbf{A}$ by $\mathbf{RQ} + \mu\mathbf{I}$]
enddo
```

Note that selecting $\mu = a_{mm}$ effectively uses the Rayleigh Quotient shift, since $a_{mm} = \mathbf{q}_m^T\mathbf{A}\mathbf{q}_m$ converges to the last eigenvalue (see the last lecture). This algorithm, therefore, performs one QR factorization of the shifted matrix $\mathbf{A}$, effectively corresponding to one step of shifted inverse simultaneous power iteration, and then reconstructs $\mathbf{A}$ as $\mathbf{RQ} + \mu\mathbf{I}$, thereby canceling the shift. This last step is crucial; otherwise, the matrix $\mathbf{A}$ would not converge to $\mathbf{D}_\lambda$ (recalling that the eigenvalues of the shifted matrix are not the same as the eigenvalues of the original matrix).

Reconstructing the eigenvectors, in this case, is also not very difficult, because we *still* have that $\check{\mathbf{Q}}^{(n)} = \mathbf{Q}^{(1)} \dots \mathbf{Q}^{(n)}$ converges to the eigenvector matrix $\mathbf{V}$, even though this time $\check{\mathbf{Q}}^{(n)}$ denotes $\mathbf{Q}$ at the $n-$th step of the shifted SI algorithm, and $\mathbf{Q}^{(n)}$ denotes $\mathbf{Q}$ at the $n-$th step of the shifted QR algorithm.[3] We can therefore proceed exactly as before.

Finally, it is worth noting that as in the case of the basic QR algorithm, there are some input matrices for which even the shifted algorithm does not converge. In fact, the matrix $\mathbf{A}$ discussed in the previous lecture has exactly the same problem with or without shift. For this reason, other shifting strategies are sometimes used (e.g. the Wilkinson shift, discussed in Chapter 29, or multiple shift strategies, which more easily generalize to non-symmetric matrices, etc..).

Example: To illustrate the QR algorithm with shifts, we repeat the previous example with the shift $\mu_k = a_{nn}^{(k-1)}$ at each iteration.

[3] The proof of this statement is just as nasty as the proof of the equivalence between non-shifted QR and SI, so we will not attempt to do it here.

Thus, with

$$\mathbf{A}^{(0)} = \begin{bmatrix} 2.9766 & 0.3945 & 0.4198 & 1.1159 \\ 0.3945 & 2.7328 & -0.3097 & 0.1129 \\ 0.4198 & -0.3097 & 2.5675 & 0.6079 \\ 1.1159 & 0.1129 & 0.6079 & 1.7231 \end{bmatrix}, \tag{4.78}$$

we take $\mu_1 = 1.7321$ as shift for the first iteration. Computing the QR factorization of the resulting shifted matrix $\mathbf{Q}^{(1)}\mathbf{R}^{(1)} = \mathbf{A}^{(0)} - \mu_1\mathbf{I}$, forming the reverse product, $\mathbf{R}^{(1)}\mathbf{Q}^{(1)}$, and then adding back to the shift, we get

$$\mathbf{A}^{(1)} = \begin{bmatrix} 3.8811 & -0.0178 & 0.2355 & 0.5065 \\ -0.0178 & 2.9528 & -0.2134 & -0.1602 \\ 0.2355 & -0.2134 & 2.0404 & -0.0951 \\ 0.5065 & -0.1602 & -0.0951 & 1.1253 \end{bmatrix}, \tag{4.79}$$

which is noticeably closer to diagonal form and to the correct eigenvalues then after one iteration of the unshifted algorithm. Our next shift is then $\mu_2 = 1.1253$, which gives

$$\mathbf{A}^{(2)} = \begin{bmatrix} 3.9946 & -0.0606 & 0.0499 & 0.0233 \\ -0.0606 & 2.9964 & -0.0882 & -0.0103 \\ 0.0499 & -0.0882 & 2.0081 & -0.0252 \\ 0.0223 & -0.0103 & -0.0252 & 1.0009 \end{bmatrix}. \tag{4.80}$$

The next shift, $\mu_3 = 1.0009$, is very close to an eigenvalue and gives

$$\mathbf{A}^{(3)} = \begin{bmatrix} 3.9980 & -0.0426 & 0.0165 & 0.0000 \\ -0.0426 & 3.0000 & -0.0433 & 0.0000 \\ 0.0165 & -0.0433 & 2.0020 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}. \tag{4.81}$$

Notice that the final iterated matrix $\mathbf{A}^{(3)}$ is very close to diagonal form. As expected for inverse iteration with a shift close to an eigenvalue, the smallest eigenvalue has been determined to the full accuracy shown. The last row of $\mathbf{A}^{(3)}$ is all zeros except for the diagonal one, so we can reduce the problem to the leading 3×3 submatrix for further iterations. Because the diagonal entries are already very close to the eigenvalues, only one or two additional iterations will be required to obtain full accuracy for the remaining eigenvaules. □

5.2. Deflation

One of the dirty secrets of the textbook is about what happens if you implement the QR with shifts *as is*. As it turns out, it's not very pretty and it's easy to see why. As the last eigenvalue converges extremely fast thanks to the shifts in the inverse iteration, the last row and last column of the matrix $\mathbf{A}$ rapidly become $(\epsilon_1, \epsilon_2, \dots, \epsilon_{m-1}, \lambda_m + \epsilon_m)$ where all the ϵ_i are order machine accuracy. With the shift in place, we then construct the matrix $\mathbf{A} - (\lambda_m + \epsilon_m)\mathbf{I}$ whose last row and last column are now entirely equal to the vector $\mathbf{0}$ within machine precision! The matrix $\mathbf{A} - \lambda_m\mathbf{I}$ is therefore singular, and this affects the precision of the

QR decomposition if we continue to work with it.

A nice solution to the problem is to stop working with the last eigenvalue/eigenvector pair as soon as we are happy with its convergence. This is the principle behind **deflation** – once an eigenvalue has converged, simply continue to work with the remaining submatrix. In other words, as soon as

$$\mathbf{A} \sim \begin{pmatrix} & \mathbf{B} & & \epsilon \\ & & & \epsilon \\ & & & \epsilon \\ \epsilon & \epsilon & \epsilon & \lambda_m \end{pmatrix} \tag{4.82}$$

continue applying the shifted QR algorithm to the sub-matrix **B**, where the shift is now the last element of **B**. Once done with **B**, copy its results back into **A**. Of course, this technique can be applied recursively, deflating **B** when its last eigenvalue has converged, and so forth.

Here is a shifted QR algorithm with basic deflation from the bottom up.

Algorithm: QR algorithm with shift and deflation:

```
do $j = m$ down to 2
    Allocate $\mathbf{B}$, $\mathbf{Q}$, $\mathbf{R}$ of size $j \times j$.
    $\mathbf{B} = \mathbf{A}(1:j, 1:j)$
    do while error on $\lambda_j$ remains too large
        $\mu = b_{jj}$ !  [Select shift]
        $\mathbf{QR} = \mathbf{B} - \mu\mathbf{I}$ !  [Compute QR factorization of $\mathbf{B} - \mu\mathbf{I}$]
        $\mathbf{B} = \mathbf{RQ} + \mu\mathbf{I}$ !  [Replace $\mathbf{B}$ by $\mathbf{RQ} + \mu\mathbf{I}$]
        $\mathbf{A}(1:j, 1:j) = \mathbf{B}$ !  [Replace $\mathbf{B}$ into $\mathbf{A}$]
    enddo
    Deallocate $\mathbf{B}$, $\mathbf{Q}$, $\mathbf{R}$
enddo
```

Working on copies of the matrix **A** in the inner loop guarantees the columns of **A** outside of **B** are left untouched. We then copy back **B** into **A** at every iteration, just to make sure **A** is kept updated at all times, so it is indeed updated when the loop exits. While this makes the algorithm easy to read, it is not very memory efficient, however. Other techniques can be used to do the same thing while working directly on **A** instead.

Recomputing the eigenvectors in this case is a little complicated, unfortunately, so we will leave this as a homework[4].

[4] Just kidding! If you need to compute the eigenvectors, use a commercial package.

5.3. The Hessenberg form

A final refinement of the QR algorithm consists in doing some preliminary work on the matrix $\mathbf{A}$ *prior* to the application of the QR algorithm, so that each QR decomposition and reconstruction is much cheaper to do. This involves putting $\mathbf{A}$ in the so-called **Hessenberg form**.

Putting a matrix $\mathbf{A}$ in **Hessenberg form** consists in finding an orthogonal transformation $\mathbf{P}$ such that $\mathbf{A} = \mathbf{P}\mathbf{A}_{\mathrm{H}}\mathbf{P}^T$, where $\mathbf{A}_{\mathrm{H}}$ is of the form

$$\mathbf{A}_{\mathrm{H}} = \begin{pmatrix} * & * & \dots & \dots & * & * \\ * & * & \dots & \dots & * & * \\ 0 & * & \dots & \dots & * & * \\ 0 & 0 & \ddots & & * & * \\ \vdots & \vdots & \ddots & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{pmatrix} \tag{4.83}$$

that is, a matrix whose coefficient ij is 0 if $i > j + 1$. Note that we are running out of letters here, so $\mathbf{P}$ is just a standard orthogonal matrix that has nothing to do with either a projection or with the reverse identity matrix introduced earlier.

Since $\mathbf{A} = \mathbf{P}\mathbf{A}_{\mathrm{H}}\mathbf{P}^T$ is a similarity transformation, the eigenvalues of $\mathbf{A}_{\mathrm{H}}$ are the same as those of the original matrix. The eigenvectors are different, but only in as much as they have gone through a change of base, which is very easy to reverse. As a result, finding the eigenvalues/eigenvectors of $\mathbf{A}$ is *almost* the same as finding the eigenvalues/eigenvectors of $\mathbf{A}_{\mathrm{H}}$.

What is the advantage of working with $\mathbf{A}_{\mathrm{H}}$ instead of $\mathbf{A}$? As it turns out, the QR algorithm applied to $\mathbf{A}_{\mathrm{H}}$ can be made *enormously* faster than the QR algorithm applied to $\mathbf{A}$. To understand why, first note that the Hessenberg form is *preserved* by the QR algorithm.

Proof: Let's first look at the QR decomposition, and assume we can write $\mathbf{A}_{\mathrm{H}} = \mathbf{QR}$. Our first claim is that $\mathbf{Q}$ is also in Hessenberg form. To show this, note that

$$(\mathbf{A}_H)_{ij} = \sum_{k=1}^{m} q_{ik} r_{kj} = \sum_{k=1}^{j} q_{ik} r_{kj} \tag{4.84}$$

because $\mathbf{R}$ is upper triangular. Then, the only way to guarantee that $(\mathbf{A}_H)_{ij} = 0$ if $i > j + 1$ is to have $q_{ik} = 0$ if $i > k + 1$.

Next, we have to show that the reconstruction $\mathbf{RQ}$ is also in Hessenberg form. This is easy, since

$$(\mathbf{RQ})_{ij} = \sum_{k=1}^{m} r_{ik} q_{kj} = \sum_{k=i+1}^{m} r_{ik} q_{kj} \tag{4.85}$$

Since $\mathbf{Q}$ is in Hessenberg form, then $q_{kj} = 0$ if $k > j + 1$ and so $(\mathbf{RQ})_{ij} = 0$ if $i > j + 1$, and is therefore also in Hessenberg form. $\square$

An even more interesting result is that the Hessenberg form of a symmetric matrix is itself symmetric (this is very easy to show), and must therefore be **tridiagonal**! Based on the result above, we therefore also know that the QR algorithm applied to a tridiagonal matrix returns a tridiagonal matrix. Since nearly all entries of a tridiagonal matrix are known zeros, they do not need to be computed, so the speed of each iteration of the QR algorithm can be increased enormously!

All of this leaves just one question – how to put $\mathbf{A}$ into Hessenberg form in the first place? As it turns out, it's very easy to do using the tools we have learned so far. Recalling that the matrix $\mathbf{P}$ has to be orthogonal, a good way to proceed is to apply successive orthogonal transformations to $\mathbf{A}$, to zero out the elements below the subdiagonal, until $\mathbf{A}$ is in Hessenberg form. We know how to do this – we just have to successively apply Householder transformations $\mathbf{H}_j$ where each of the $\mathbf{H}_j$ are created to zero out sub-subdiagonal elements instead of subdiagonal ones (see below on how to do that):

$$\mathbf{A} \to \mathbf{H}_m \dots \mathbf{H}_1 \mathbf{A} \equiv \mathbf{P}^T \mathbf{A} \tag{4.86}$$

Because we want to create a similarity transformation, however, we have to apply $\mathbf{P}$ on *both* sides of $\mathbf{A}$, which implies

$$\begin{aligned} \mathbf{A}_{\mathrm{H}} \quad &= \mathbf{P}^T\mathbf{A}\mathbf{P} = \mathbf{H}_m \dots \mathbf{H}_1 \mathbf{A} (\mathbf{H}_m \dots \mathbf{H}_1)^T \\ &= \mathbf{H}_m \dots \mathbf{H}_1 \mathbf{A} \mathbf{H}_1^T \dots \mathbf{H}_m^T = \mathbf{H}_m \dots \mathbf{H}_1 \mathbf{A} \mathbf{H}_1 \dots \mathbf{H}_m \end{aligned} \tag{4.87}$$

since each $\mathbf{H}_j$ is symmetric. One may therefore worry that applying $\mathbf{H}_j$ to the right of $\mathbf{A}$ at each iteration may repopulate the elements we just zeroed out by applying $\mathbf{H}_j$ to the left! As it turns out, this does not happen, because of the form of the matrix $\mathbf{H}_j$.

Indeed, by analogy with what we have learned so far, to zero out the sub-subdiagonals in the $j-$th column, we first compute

$$s_j = \mathrm{sign}(a_{j+1,j}) \sqrt{\sum_{k=j+1}^{m} a_{kj}^2} \tag{4.88}$$

Then create the Householder vector $\mathbf{v}_j = (0, \dots, 0, a_{j+1,j} + s_j, a_{j+2,j}, \dots, a_{mj})^T$ and normalize it, and finally, create $\mathbf{H}_j = \mathbf{I} - 2\mathbf{v}_j\mathbf{v}_j^T$. By construction, therefore, the first j elements of $\mathbf{v}_j$ are all zeros, which means that the first j rows and columns of $\mathbf{H}_j$ are equal to the first j rows and columns of the identity matrix. Because of this, applying $\mathbf{H}_j$ to the left of $\mathbf{A}$ only affects rows $j + 1$ and up, and applying it to the right only affects columns $j + 1$ and up – leaving the ones we have already worked on as they are!

We therefore have the following algorithm for reduction to Hessenberg form:

Algorithm: Reduction to Hessenberg form of a square matrix $\mathbf{A}$:

```
do $j = 1$ to $m-2$
    ![loop over columns]
    $s_j = \mathtt{sign}(a_{j+1,j})\sqrt{\sum_{i=j+1}^m a_{ij}^2}$
    !  [compute signed norm]
    $\mathbf{v}_j = [0, \cdots 0, a_{j+1,j} + s_j, a_{j+2,j}, \cdots, a_{mj}]^T$
    $\mathbf{v}_j = \mathbf{v}_j / ||\mathbf{v}_j||$
    !  [compute Householder vector and normalize it]
    $\mathbf{A} = \mathbf{A} - 2\mathbf{v}_j\mathbf{v}_j^T\mathbf{A}$  ![Update $\mathbf{A}$ from the left]
    $\mathbf{A} = \mathbf{A} - 2\mathbf{A}\mathbf{v}_j\mathbf{v}_j^T$  ![Update $\mathbf{A}$ from the right]
enddo
```

As expected, this looks very much like the Householder algorithm for QR decomposition, with only 2 modifications: (1) starting from the subdiagonal element $a_{j+1,j}$ in the sum involved in s_j, and in the vector $\mathbf{v}_j$, instead of the diagonal element a_{jj}, and (2) applying $\mathbf{H}_j$ from the right.

On exit, $\mathbf{A}$ should be in Hessenberg form if $\mathbf{A}$ is any normal nonsingular matrix, and in tridiagonal form if $\mathbf{A}$ is symmetric. Note that this algorithm does *not* return $\mathbf{P}$. This is fine if we do not want to compute the eigenvectors, but if we do, then we have to create $\mathbf{P}$ on the side and save it. As usual, there is a simple way (initialize $\mathbf{P} = \mathbf{I}$, then add the line $\mathbf{P} = \mathbf{H}_j\mathbf{P}$ at the end of the algorithm), and an efficient way (in which the sub-subdiagonal components of the Householder vectors are stored in the place of the zeros, and the subdiagonal elements of these vectors are returned in a separate array).

5.4. Summary

For real symmetric matrices, the combination of (1) reduction to Hessenberg form, (2) a QR algorithm taylor-made for tridiagonal matrices, (3) shifts and (4) deflation, transforms the basic QR algorithm into a blindly fast one, which is now the industry standard. The overall operation count is of order $\frac{2}{3}m^3$ multiplications, which is even less than computing the product of two matrices!

Because the algorithm only makes use of basic Householder transformations and simple matrix multiplications, it has been shown to be backward-stable when it converges. The computed eigenvalues have been shown to be accurate within a factor that is order machine accuracy times the norm of $\mathbf{A}$:

$$|\tilde{\lambda}_j - \lambda_j| = O(||\mathbf{A}||\epsilon_{\text{mach}}) \tag{4.89}$$

where $\tilde{\lambda}_j$ is the answer returned by the algorithm.

All that is left to do, now, is to see how one may generalize some of these algorithms to non-symmetric matrices!

6. Beyond real symmetric matrices

See Chapter 7 of Golub and Van Loan's Matrix Computations

The general case where matrices are neither real nor symmetric is much more complicated than what we have seen so far. Consider for instance what happens when the matrix $\mathbf{A}$ is real, but not symmetric. None of the nice properties of real symmetric matrices apply. In particular

- The eigenvalues may not be real (even though $\mathbf{A}$ is real)
- The eigenvectors may not be orthogonal
- Some eigenvalues could be defective so the basis of eigenvectors is not necessarily a complete basis.

and so forth. As a result, the QR algorithm no longer turns $\mathbf{A}$ into the eigenvalue matrix $\mathbf{D}_\lambda$. However, that does not mean the QR algorithm is now useless. On the contrary, as we shall see below.

Note that this final section of the Eigenvalue chapter is not meant to be rigorous or comprehensive. In particular, we will not prove anything, nor is it expected that you should be able to code any of the algorithms described. Instead, this is the point at which we shall start relying entirely on more sophisticated commercial algorithms to give us the desired answers.

6.1. The complex QR algorithm for Hermitian matrices

A very easy extension of the QR algorithm can be made for Hermitian matrices. Recall that a Hermitian matrix is such that

$$\mathbf{A} = \mathbf{A}^* \tag{4.90}$$

where $\mathbf{A}^*$ is the Hermitian conjugate of $\mathbf{A}$ (i.e. the complex conjugate of the transpose of $\mathbf{A}$). Relevant properties of Hermitian matrices are

- Their eigenvalues are real, and non-defective
- The eigenvectors are orthogonal (or can be made orthogonal) and form a basis for the whole space
- A Hermitian matrix can be diagonalized as $\mathbf{D}_\lambda = \mathbf{U}^*\mathbf{A}\mathbf{U}$ where $\mathbf{U}$ is a unitary matrix.

The last step tells us that, as long as we can find the right unitary matrix $\mathbf{U}$, we can find the (real) eigenvalues and (complex) eigenvectors of $\mathbf{A}$. As it turn out, it is possible to generalize the QR algorithm quite trivially to find $\mathbf{U}$ and therefore $\mathbf{D}_\lambda$, simply by using a complex version of the QR decomposition algorithm that writes a complex matrix

$$\mathbf{A} = \mathbf{U}\mathbf{R} \tag{4.91}$$

where $\mathbf{U}$ is unitary. Computing this complex QR factorization is also done using a relatively simple generalization of the standard QR algorithm that handles complex-valued vectors.

6.2. The Schur factorization for non-Hermitian matrices, and its relationship with the complex QR decomposition

We have just seen that for Hermitian matrices (or the special case of real symmetric matrices), there exist a factorization of $\mathbf{A}$ such that

$$\mathbf{D}_\lambda = \mathbf{V}^*\mathbf{A}\mathbf{V} \text{ (or } \mathbf{D}_\lambda = \mathbf{V}^T\mathbf{A}\mathbf{V} \text{ in the real case)} \tag{4.92}$$

where $\mathbf{V}$ is the unitary (orthogonal) eigenvector matrix, and $\mathbf{D}_\lambda$ is the eigenvalue matrix. This type of factorization is called an **eigenvalue-revealing factorization**, and the purpose of the QR algorithm we just learned was precisely to compute $\mathbf{D}_\lambda$ by successive iterations on $\mathbf{A}$.

As it turns out, while this factorization is not always possible for non real-symmetric matrices, there *is* another eigenvalue-revealing factorization that exists in the general case; this is the **Schur factorization**.

Theorem: Let $\mathbf{A}$ be an $m \times m$ complex matrix. Then there always exists a unitary matrix $\mathbf{U}$ such that

$$\mathbf{T} = \mathbf{U}^*\mathbf{A}\mathbf{U} \tag{4.93}$$

where $\mathbf{T}$ is upper triangular.

Since $\mathbf{U}^* = \mathbf{U}^{-1}$, this is a similarity transformation so $\mathbf{A}$ and $\mathbf{T}$ have the same eigenvalues. Furthermore, since the eigenvalues of an upper triangular matrix are its diagonal elements, we see that the Schur factorization is an eigenvalue-revealing factorization, i.e. it reveals the eigenvalues of $\mathbf{A}$ as the diagonal elements of $\mathbf{T}$. Note that if a matrix $\mathbf{A}$ is diagonalizable, then $\mathbf{T} = \mathbf{D}_\lambda$, so the Schur form can be viewed as a generalization of the concept of diagonalization for matrices that are not diagonalizable (i.e. when the eigenvectors are not orthogonal, or when the basis of eigenvectors is not complete). What is crucial is that this decomposition *always* exists, even if the matrix is defective.

We will not prove this theorem in general, although if you are interested in the proof, you can find it in Golub & Van Loan (the proof is based on a recursive algorithm). However, the case of a simple non-defective matrix is quite easy to prove. Indeed, if $\mathbf{A}$ is non-defective, then its eigenvectors form a basis for all space, and in that basis, $\mathbf{A}$ is diagonalizable. In other words,

$$\mathbf{D}_\lambda = \mathbf{V}^{-1}\mathbf{A}\mathbf{V} \tag{4.94}$$

where $\mathbf{V}$ is the eigenvector matrix (not necessarily unitary). Let us now construct the complex QR decomposition of $\mathbf{V}$ as $\mathbf{V} = \mathbf{U}\mathbf{R}$, where $\mathbf{U}$ is unitary. Then

$$\mathbf{D}_\lambda = (\mathbf{U}\mathbf{R})^{-1}\mathbf{A}\mathbf{U}\mathbf{R} = \mathbf{R}^{-1}\mathbf{U}^*\mathbf{A}\mathbf{U}\mathbf{R} \Rightarrow \mathbf{R}\mathbf{D}_\lambda\mathbf{R}^{-1} = \mathbf{U}^*\mathbf{A}\mathbf{U} \tag{4.95}$$

The multiplication of $\mathbf{D}_\lambda$ by an upper triangular matrix returns an upper triangular one. In addition, $\mathbf{R}^{-1}$ is also upper triangular, and the product of two upper triangular matrices is also upper triangular. As a result $\mathbf{R}\mathbf{D}_\lambda\mathbf{R}^{-1}$ is upper triangular, so, we have the desired decomposition

$$\mathbf{T} = \mathbf{U}^*\mathbf{A}\mathbf{U} \tag{4.96}$$

As it turns out, it can be shown that the complex QR algorithm (i.e. the QR algorithm that uses the complex QR decomposition), when applied to a non-Hermitian matrix, gradually transforms the matrix **A** into its Schur form[5]! The situation is therefore a priori very simple – just apply the algorithm repeatedly to the matrix **A**, and the (complex) eigenvalues shall be revealed on the diagonal!

6.3. The LAPACK routines for eigenvalue searches

While the theory is, in theory, very simple, actually getting these algorithms to work robustly is quite complicated, and at this point, it is best to leave this to specialists. MATLAB, for instance, has a large number of built-in routines for eigenvalue searches. In Fortran/C/C++, the standard routines are called BLAS and LAPACK. BLAS stands for **Basic Linear Algebra Subprograms** and contains a whole library of subroutines for Linear Algebra. LAPACK stands for **Linear Algebra PACKage**, and is a higher-level series of routines that use the BLAS to solve more advanced problems such as solving linear equations, finding eigenvalues, etc. See `http://www.netlib.org/lapack/#_presentation/` for more general information, and `http://www.netlib.org/lapack/lug/` for a detailed User Guide.

Using LAPACK libraries can be challenging; in fact, just finding out what each routine does is no mean feat, as their documentation assumes that the user has serious knowledge of linear algebra. Also, LAPACK has a number of different levels of routines, which can be confusing at first. They are split into

- Driver routines, each of which solves a complete problem (e.g. a system of linear equations, an eigenvalue problem, etc.)
- Basic computational routines, which perform standard tasks such as Cholesky decomposition, QR decomposition, etc..
- Auxiliary routines that perform low-level computations that are often needed by the driver or basic computational routines. These are usually done by the BLAS, and are highly optimized.

Generally speaking, you will rarely have to directly use the auxiliary routines, you may occasionally have to use the basic computational routines, but most likely will make use of the driver routines.

LAPACK routines are nearly always available for both single and double-precision, real or complex problems, and follow a simple naming scheme accordingly: the first letter of the routine tells you what kind of variables you are working with, and the rest of the name is the actual name of the routine.

[5] This statement, once again, will not be proved.

- `S` is for real, single precision
- `D` is for real, double precision
- `C` is for complex, single precision
- `Z` is for complex, double precision

The most useful section of the User Guide is the one that describes what each routine does: `http://www.netlib.org/lapack/lug/node25.html` For instance, suppose we want to find out how to compute the eigenvalues of a real symmetric matrix, we would go to the relevant section (Symmetric Eigenproblems), which then contains a description of the various driver routines that can be used. As you can see, there are a number of possibilities, which are all summarized in their table 2.5 (follow the link). Once you have the name of a routine, say for instance DSYEV (for Double-precision SYmmetric EigenValue problem), you can search for it online, and end up on the NETLIB page which has an in-depth description of what this particular routine does, what arguments it takes, and what arguments it returns. It also contains a diagram that explains what dependencies this routine has. You should definitely spend a little time marveling at the sophistication added to the basic algorithm to ensure that they are indeed returning robust answers for all possible input matrices!

Chapter 5

Singular Value Decomposition

We now reach an important Chapter in this course concerned with the Singular Value Decomposition of a matrix $\mathbf{A}$. SVD, as it is commonly referred to, is one of the most powerful non-trivial tools of Numerical Linear Algebra that is used for a vast range of applications, from image compression, to Least Square fitting, and generally speaking many forms of big-data analysis, as we shall see below.

We will first learn more about the SVD, discuss briefly how one may compute it, then see some examples. By contrast with other Chapters, here we will not focus so much on algorithmic development (which is a very complicated topic) but more on understanding the theoretical and practical aspects of the SVD.

1. What is the SVD?

See Chapter 4 of the textbook

1.1. General ideas

As we saw in Chapter 1, it is possible, from any matrix $\mathbf{A}$ of size $m \times n$, to define singular values σ_i and left and right singular vectors $\mathbf{u}_i$ and $\mathbf{v}_i$ so that

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i \tag{5.1}$$

where $\mathbf{u}_i$ are the normalized $m \times 1$ vectors parallel to the principal axes of the hyperellipse formed by the image of the unit ball via application of $\mathbf{A}$, $\sigma_i > 0$ are the lengths of these principal axes, and $\mathbf{v}_i$ are the pre-images of $\mathbf{u}_i$, and of size $n \times 1$. If $r = \text{rank}(\mathbf{A})$, then there will be r such principal axes, and therefore r non-zero singular values.

Let us now re-write this as the matrix operation

$$\mathbf{AV} = \mathbf{U\Sigma} = \mathbf{A}\begin{pmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_r \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_r \\ | & | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{pmatrix} \tag{5.2}$$

where $\mathbf{A}$ is $m \times n$, $\mathbf{V}$ is $n \times r$, $\mathbf{U}$ is $m \times r$ and $\mathbf{\Sigma}$ is $r \times r$.

A very non-trivial result (that we prove below) is that both $\mathbf{U}$ and $\mathbf{V}$ **are unitary matrices** (orthogonal, if $\mathbf{A}$ is real). While it may seem obvious that $\mathbf{U}$ should be unitary since it is formed by the principal axes of the hyperellipse, note that we have not proved yet that the image of the unit ball *is always* a hyperellipse. Furthermore, there is no obvious reason why the pre-images $\mathbf{v}_i$ of the $\mathbf{u}_i$ vectors should themselves be orthogonal, even if the $\mathbf{u}_i$ are. Nevertheless, assuming this is indeed the case, then we have

$$\mathbf{U}^*\mathbf{A}\mathbf{V} = \mathbf{\Sigma} \text{ or equivalently } \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{5.3}$$

The first of these two expressions reveals something that looks a lot like a diagonalization of $\mathbf{A}$ in its own subspace, and is clearly an extension of the more commonly known expression

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{D}_\lambda \tag{5.4}$$

when $\mathbf{A}$ is diagonalizable (i.e. square, and non-defective) and $\mathbf{V}$ is the basis of its eigenvectors. Crucially, however, we will show that writing $\mathbf{U}^*\mathbf{A}\mathbf{V} = \mathbf{\Sigma}$ can be done for any matrix $\mathbf{A}$, even defective ones, and *even* singular ones.

The second of these expression writes $\mathbf{A}$ as a factorization between interesting matrices – here, two orthogonal ones and one diagonal, positive definite one, and is directly related to the so-called reduced and full singular value decompositions of $\mathbf{A}$. The distinction between reduced SVD and full SVD here is very similar to the distinction we made between reduced QR decomposition and full QR decomposition in Chapter 3.

1.2. The reduced SVD

To construct the **reduced singular value decomposition** of $\mathbf{A}$, we add the normalized vectors $\mathbf{v}_{r+1}, \dots \mathbf{v}_n$ to $\mathbf{V}$, choosing them carefully in such a way as to be orthogonal to one another and to all the already-existing ones. This transforms $\mathbf{V}$ into a square orthogonal matrix, whose column vectors form a complete basis for the space of vectors of length n. Having done that, we then need to

- expand $\mathbf{\Sigma}$ to size $n \times n$, where the added rows and columns are identically zero
- add the vectors $\mathbf{u}_{r+1}, \dots \mathbf{u}_n$ to $\mathbf{U}$. These vectors must be normalized, orthogonal to one another, and to the existing ones. Note that these new vectors lie *outside* of the range of $\mathbf{A}$, by definition.

The new factorization of $\mathbf{A}$ then becomes $\mathbf{A} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^*$, and more precisely

$$\mathbf{A} = \begin{pmatrix} | & & | & | & & | \\ \mathbf{u}_1 & \dots & \mathbf{u}_r & \mathbf{u}_{r+1} & \dots & \mathbf{u}_n \\ | & & | & | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix} \begin{pmatrix} - & \mathbf{v}_1^* & - \\ & \vdots & \\ - & \mathbf{v}_r^* & - \\ - & \mathbf{v}_{r+1}^* & - \\ & \vdots & \\ - & \mathbf{v}_n^* & - \end{pmatrix} \tag{5.5}$$

Recalling that this can also be written $\mathbf{A}\hat{\mathbf{V}} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}$, we find that the added $\mathbf{v}$ vectors (in red) satisfy $\mathbf{A}\mathbf{v}_i = 0\mathbf{u}_i$, revealing them to span the nullspace of $\mathbf{A}$. With this construction, $\hat{\mathbf{V}}$ and $\hat{\mathbf{\Sigma}}$ have size $n \times n$, and $\hat{\mathbf{U}}$ has size $m \times n$. .

1.3. The full SVD

Just as in the QR case, we can then also define a full SVD, by completing the basis formed by the orthogonal vectors $\mathbf{u}_i$ into full basis for the space of $m \times 1$ vectors. We then continue to add vectors to $\mathbf{U}$, normalized, orthogonal to one another and orthogonal to the previous ones, as usual. We also continue to pad $\mathbf{\Sigma}$ with more rows containing only zeros. The matrix $\mathbf{V}$ on the other hand remains unchanged. This yields

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{5.6}$$

where, this time, $\mathbf{U}$ is an $m \times m$ matrix, $\mathbf{V}$ is an $n \times n$ matrix, and $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix, with zeros everywhere else. In this sense, $\mathbf{U}$ and $\mathbf{V}$ are square unitary matrices, but $\mathbf{\Sigma}$ has the same shape as the original matrix $\mathbf{A}$. As in the case of the reduced vs. full QR decompositions, the full SVD naturally contains the reduced SVD in its columns and rows.

1.4. The existence of the SVD

We now come to the tricky part, namely demonstrating that a SVD exists for all matrices. Note that the theorem applies to the full SVD, but since the reduced SVD is contained in the latter, it de-facto also proves the existence of the reduced SVD.

Theorem: (Singular Value Decomposition) Let $\mathbf{A}$ be an $m \times n$ matrix. Then there exist two unitary matrices $\mathbf{U}$ (of size $m \times m$) and $\mathbf{V}$ (of size $n \times n$) such that

$$\mathbf{U}^*\mathbf{A}\mathbf{V} = \mathbf{\Sigma}, \tag{5.7}$$

where $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix whose $p = \min(m, n)$ entries are $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_p \geq 0$. If $\text{rank}(\mathbf{A}) = r$, then there will be exactly r non-zero σ_i, and all the other ones are zero.

Idea behind the proof: The proof is inductive but unfortunately not very constructive. The idea is to work singular value by singular value. Let's begin with the first singular value of $\mathbf{A}$, which we proved a long time ago to satisfy

$$\sigma_1 = ||\mathbf{A}||_2 \tag{5.8}$$

Because of the way $||\mathbf{A}||_2$ is constructed, we know that there exists[1] a unit vector $\mathbf{x}$ that satisfies $||\mathbf{A}\mathbf{x}||_2 = \sigma_1$; let that vector be $\mathbf{v}_1$, and let $\mathbf{u}_1$ be the unit vector defined such that $\mathbf{A}\mathbf{v}_1 = \sigma_1\mathbf{u}_1$. Let's then construct a unitary matrix $\mathbf{U}_1$ whose first column is $\mathbf{u}_1$ (randomly pick $\mathbf{u}_2$, ... $\mathbf{u}_m$ normalized and orthogonal to one

[1] Note that this is where the proof lets us down a little since although we know $\mathbf{x}$ must exist, it doesn't tell us how to find it.

another and to $\mathbf{u}_1$), and another unitary matrix $\mathbf{V}_1$ whose first column is $\mathbf{v}_1$ (randomly pick $\mathbf{v}_2$, ... $\mathbf{v}_n$ normalized orthogonal to one another and to $\mathbf{v}_1$).

Then let

$$\mathbf{A}_1 = \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 \tag{5.9}$$

Since $\mathbf{A}\mathbf{v}_1 = \sigma_1 \mathbf{u}_1$ and since $\mathbf{u}_i^* \mathbf{u}_1 = \delta_{i1}$ (because $\mathbf{U}$ is unitary), it is easy to show that $\mathbf{A}_1$ takes the form

$$\mathbf{A}_1 = \begin{pmatrix} \sigma_1 & \mathbf{w}^* \\ \mathbf{0} & \mathbf{B} \end{pmatrix} \tag{5.10}$$

where $\mathbf{B}$ is an $m-1 \times n-1$ matrix, $\mathbf{w}$ is an $n-1$ long vector, and $\mathbf{0}$ is an $m-1$ long vector full of zeros. The key to the proof is to show that $\mathbf{w}$ is also zero.

To do so, first note that because $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices, $||\mathbf{A}_1||_2 = ||\mathbf{A}||_2$ (see Lecture 2). Next, let's consider the effect of $\mathbf{A}_1$ on the column vector $\mathbf{s} = (\sigma_1, \mathbf{w}^T)^T$.

$$\mathbf{A}_1 \mathbf{s} = \mathbf{A}_1 \begin{pmatrix} \sigma_1 \\ \mathbf{w} \end{pmatrix} = \begin{pmatrix} \sigma_1^2 + \mathbf{w}^* \mathbf{w} \\ \mathbf{B}\mathbf{w} \end{pmatrix} \tag{5.11}$$

On the one hand, because σ_1 is the norm of $\mathbf{A}_1$, we know that

$$||\mathbf{A}_1 \mathbf{s}||_2 \leq \sigma_1 ||\mathbf{s}||_2 = \sigma_1 \sqrt{\sigma_1^2 + \mathbf{w}^* \mathbf{w}} \tag{5.12}$$

by definition of the norm. On the other hand, we also know that

$$||\mathbf{A}_1 \mathbf{s}||_2 = \sqrt{(\sigma_1^2 + \mathbf{w}^* \mathbf{w})^2 + ||\mathbf{B}\mathbf{w}||_2^2} \geq \sigma_1^2 + \mathbf{w}^* \mathbf{w} \tag{5.13}$$

Putting these two bounds together, we have

$$\sigma_1 \sqrt{\sigma_1^2 + \mathbf{w}^* \mathbf{w}} \geq \sigma_1^2 + \mathbf{w}^* \mathbf{w} \tag{5.14}$$

which can only be true if $\mathbf{w} = 0$. In other words, any unitary matrices $\mathbf{U}_1$ and $\mathbf{V}_1$ constructed with $\mathbf{u}_1$ and $\mathbf{v}_1$ as their fist column vectors result in

$$\mathbf{A}_1 = \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 = \begin{pmatrix} \sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{pmatrix} \tag{5.15}$$

If we continue the process for $\mathbf{B}$, pick its first singular value (which we call σ_2), identify the vectors $\mathbf{u}_2$ and $\mathbf{v}_2$, and construct unitary matrices $\mathbf{U}_2$ and $\mathbf{V}_2$ based on them, we can then write

$$\mathbf{B}_2 = \hat{\mathbf{U}}_2^* \mathbf{B} \hat{\mathbf{V}}_2 = \begin{pmatrix} \sigma_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{C} \end{pmatrix} \tag{5.16}$$

which implies that

$$\mathbf{U}_2^* \mathbf{U}_1^* \mathbf{A} \mathbf{V}_1 \mathbf{V}_2 = \begin{pmatrix} \sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} = \begin{pmatrix} \sigma_1 & 0 & \mathbf{0} \\ 0 & \sigma_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{C} \end{pmatrix} \tag{5.17}$$

where

$$\mathbf{U}_2 = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{U}}_2 \end{pmatrix} \text{ and } \mathbf{V}_2 = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{V}}_2 \end{pmatrix} \tag{5.18}$$

The procedure can be continued over and over again, to reveal, one by one, all the singular values of **A**.

Two things are left to discuss. One, is the fact that $\mathbf{U}_2$ and $\mathbf{V}_2$ constructed as above are indeed unitary. This is the case, because by construction everything related to **B** is orthogonal to $\mathbf{v}_1$ and $\mathbf{u}_1$ (respectively). And since the product of unitary matrices is also unitary, the construction gradually constructs the singular value decomposition of **A** with

$$\mathbf{U} = \mathbf{U}_1\mathbf{U}_2 \dots \mathbf{U}_r \text{ and } \mathbf{V} = \mathbf{V}_1\mathbf{V}_2 \dots \mathbf{V}_r \tag{5.19}$$

It is worth noting that, because of the way we constructed **U** and **V**, the first column of **U** contains $\mathbf{u}_1$, the second column contains $\mathbf{u}_2$, etc.. , and similarly for **V**, the first column contains $\mathbf{v}_1$, the second $\mathbf{v}_2$, etc, as we expect from the original description of the SVD.

Second, is what to do when the next singular value happens to be zero. Well, the simple answer is nothing, because only the zero matrix can have a singular value that is identically 0. And since the matrix $\mathbf{U}_r$ and $\mathbf{V}_r$ were constructed with completed bases already, all the work has already been done! □

It is worth noting that a singular value decomposition is not unique. For instance, if two singular values are the same, then one can easily switch the order in which the vectors corresponding to that value appear in **U** and **V** to get two different SVDs. Also, note how one can easily switch the signs of both $\mathbf{u}_i$ and $\mathbf{v}_i$ at the same time, and therefore get another pair of matrices $\mathbf{U}'$ and $\mathbf{V}'$ that also satisfies $\mathbf{A} = \mathbf{U}'\mathbf{\Sigma}\mathbf{V}'^*$.

1.5. Simple / trivial examples of SVD.

Here are a few matrices where the SVD can easily be found by inspection.

Example 1: In this case, **A** is nearly in the right form, but recall that the diagonal matrix in the SVD needs to have only positive entries. This is easily fixed as

$$\begin{pmatrix} 3 & 0 \\ 0 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{5.20}$$

Other decomposition can be created by changing the signs of $\mathbf{u}_1$ and $\mathbf{v}_1$, and/or $\mathbf{u}_2$ and $\mathbf{v}_2$.

Example 2: In this case, **A** is again nearly in the right form, but the order of the singular values is wrong. This is easily fixed by doing a permutation of the rows and columns, as in

$$\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{5.21}$$

Again, other decomposition can be created by changing the signs of $\mathbf{u}_1$ and $\mathbf{v}_1$, and/or $\mathbf{u}_2$ and $\mathbf{v}_2$.

Example 3: In this case, $\mathbf{A}$ is again nearly in the right form, and would be if the columns were switched. This can be done by multiplying $\mathbf{A}$ from the right by a 2×2 permutation matrix, which ends up being $\mathbf{V}^*$. There is nothing left to do from the left, so we simply take $\mathbf{U}$ to be the identity matrix.

$$\begin{pmatrix} 0 & 2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{5.22}$$

And so forth.. More examples can be found in the textbook, and will be set as project questions.

2. Properties of the SVD

See Chapter 5 of the textbook

The SVD has a number of important properties that will turn out to be very useful. The first few we will just state; many of them we have already seen:

- The SVD of a real matrix has real (and therefore orthogonal) $\mathbf{U}$, $\mathbf{V}$
- The number of non-zero singular values is equal to rank($\mathbf{A}$).
- The vectors $\mathbf{u}_1 \dots \mathbf{u}_r$ span the range of $\mathbf{A}$, while the vectors $\mathbf{v}_{r+1} \dots \mathbf{v}_n$ span the nullspace of $\mathbf{A}$.
- $||\mathbf{A}||_2 = \sigma_1$ and $||\mathbf{A}||_F = \sqrt{\sigma_1^2 + \dots + \sigma_r^2}$. (To prove the second, simply recall that the Frobenius norm of the product of $\mathbf{A}$ with a unitary matrix is equal to the Frobenius norm of $\mathbf{A}$ (see Lecture 2)).
- If $\mathbf{A}$ is a square matrix $m \times m$, then $|\det \mathbf{A}| = \sigma_1 \sigma_2 \dots \sigma_m$ (To prove this, first remember or prove that the determinant of a unitary matrix is ±1).

Another very interesting property of the SVD comes from realizing that, if $\mathbf{A}$ is square and is non-singular, then the inverse of $\mathbf{A}$ satisfies

$$\mathbf{A}^{-1} = (\mathbf{U\Sigma V}^*)^{-1} = \mathbf{V\Sigma}^{-1}\mathbf{U}^* \tag{5.23}$$

which is *nearly* in SVD form. Indeed,

$$\mathbf{\Sigma}^{-1} = \begin{pmatrix} \frac{1}{\sigma_1} & & & \\ & \frac{1}{\sigma_2} & & \\ & & \ddots & \\ & & & \frac{1}{\sigma_m} \end{pmatrix} \tag{5.24}$$

so the elements are not in the right order. This is easy to fix, however, simply by doing a permutation of the rows and columns on either side:

$$\mathbf{A}^{-1} = (\mathbf{VP})(\mathbf{P\Sigma}^{-1}\mathbf{P})(\mathbf{PU}^*) \tag{5.25}$$

where $\mathbf{P}$ is the same reverse identity matrix we introduced in Lecture 11, which satisfies $\mathbf{PP} = \mathbf{I}$. Since $\mathbf{P}$ is merely a permutation, both $\mathbf{VP}$ and $\mathbf{PU}^*$ are unitary matrices since $\mathbf{V}$ and $\mathbf{U}$ are. Hence, the ordered singular values of $\mathbf{A}^{-1}$ are $(\sigma_m^{-1}, \dots, \sigma_1^{-1})$. This then also proves one of the results we have learned a while back, that $||\mathbf{A}^{-1}||_2 = \sigma_m^{-1}$, a result we then used to show that the condition number of $\mathbf{A}$ is

$$\mathrm{cond}(\mathbf{A}) = ||\mathbf{A}^{-1}||_2 ||\mathbf{A}||_2 = \frac{\sigma_1}{\sigma_m} \tag{5.26}$$

Next, let us construct the matrix $\mathbf{A}^*\mathbf{A}$, in terms of the SVD of $\mathbf{A}$. We have

$$\mathbf{A}^*\mathbf{A} = (\mathbf{U\Sigma V}^*)^*(\mathbf{U\Sigma V}^*) = \mathbf{V\Sigma}^*\mathbf{U}^*\mathbf{U\Sigma V}^* \tag{5.27}$$

$$= \mathbf{V\Sigma}^*\mathbf{\Sigma V}^* = \mathbf{V} \begin{pmatrix} \sigma_1^2 & & & \\ & \sigma_2^2 & & \\ & & \ddots & \\ & & & \sigma_n^2 \end{pmatrix} \mathbf{V}^* \tag{5.28}$$

This is effectively a diagonalization of the matrix $\mathbf{A}^*\mathbf{A}$, which shows that **the non-zero eigenvalues of $\mathbf{A}^*\mathbf{A}$ are the square of the nonzero singular values of $\mathbf{A}$**!

Finally if $\mathbf{A}$ is Hermitian ($\mathbf{A} = \mathbf{A}^*$), then we know that its eigenvalue/eigenvector decomposition is

$$\mathbf{A} = \mathbf{Q}^*\mathbf{D}_\lambda\mathbf{Q} \tag{5.29}$$

where $\mathbf{Q}$ is the matrix whose column vectors are the eigenvectors of $\mathbf{A}$. Vector by vector, this implies $\mathbf{Aq}_i = \lambda_i\mathbf{q}_i$.

This actually looks very much like a singular value decomposition, except for the fact that some of the λ_i can be negative while σ_i must be positive. However by noting that you can write $\lambda_i = \mathrm{sign}(\lambda_\mathrm{i})\sigma_i$, then

$$\mathbf{Aq}_i = \lambda_i\mathbf{q}_i = \mathrm{sign}(\lambda_\mathrm{i})\sigma_i\mathbf{q}_i = \sigma_i\mathbf{u}_i \tag{5.30}$$

where $\mathbf{u}_i = \mathrm{sign}(\lambda_\mathrm{i})\mathbf{q}_i$. We can therefore use the $\mathbf{q}_i$ to find the left and right singular vectors $\mathbf{u}_i$, and $\mathbf{v}_i = \mathbf{q}_i$, and then create the matrices $\mathbf{U}$ and $\mathbf{V}$ associated with the SVD of $\mathbf{A}$. All of this shows that, **for a Hermitian matrix, we have both $\sigma_i = |\lambda_i|$ and the corresponding eigenvectors $\mathbf{v}_i$ are also the right singular vectors**. Note that if the singular values are not ordered, then we can simply use the trick of multiplying $\mathbf{\Sigma}$ by a permutation matrix.

3. How to (not) compute the SVD

See textbook Chapter 31

Computing the SVD is a little like solving a Least-Square problem: there are very easy but very unstable ways of doing it, but it is much harder to do it in a stable and accurate way. In fact, it seems to be so difficult that most textbooks ignore the question entirely, or give very brief and incomplete statements. We shall therefore follow suite, and leave any computation of the SVD to professionals (e.g. LAPACK routines). Nevertheless, it is quite informative to see *why* the easy/obvious method may fail.

In the previous lecture, we saw that, given a matrix $\mathbf{A}$, the SVD of the matrix $\mathbf{A}^*\mathbf{A}$ is

$$\mathbf{A}^*\mathbf{A} = \mathbf{V}\mathbf{D}_\sigma^2\mathbf{V}^* \tag{5.31}$$

where $\mathbf{D}_\sigma$ is a diagonal matrix containing all of the singular values, and $\mathbf{V}$ is the matrix of the right singular vectors of $\mathbf{A}$. But this expression is *also* an expression for the diagonalization of $\mathbf{A}^*\mathbf{A}$. Therefore, $\mathbf{V}$ is revealed to be the matrix of its eigenvectors, and its eigenvalues are therefore σ_i^2.

This suggests that a very simple technique for finding the singular value decomposition of $\mathbf{A}$ is:

- Form $\mathbf{A}^*\mathbf{A}$
- Find its eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ using the QR algorithm for symmetric matrices
- Let $\sigma_i = \sqrt{\lambda_i}$, and $\mathbf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i$. This can always be done because $\mathbf{A}^*\mathbf{A}$ is positive definite.

If $\mathbf{A}$ is not an ill-conditioned matrix, just as in the case of Least Square problems, this does indeed work. However, one of the main advantages of using Singular Value Decomposition is to be able to deal with matrices that are close to being singular, or that are actually singular. Unfortunately, recall that

$$\text{cond}(\mathbf{A}^*\mathbf{A}) = \text{cond}(\mathbf{A})^2 \tag{5.32}$$

so if $\mathbf{A}$ is already poorly conditioned, $\mathbf{A}^*\mathbf{A}$ will be even more poorly conditioned and any operation of this matrix will be prone to truncation errors of order one.

The trick to performing a Singular Value Decomposition numerically therefore crucially hangs on *not* forming the product $\mathbf{A}^*\mathbf{A}$. The textbook gives some hint as to how this can be done. See more on the topic in *Matrix Computations* by Golub & Van Loan. The SVD algorithms from LAPACK are described in `http://www.netlib.org/lapack/lug/node32.html`, and come in two forms: basic algorithms (`xGESVD`), and divide and conquer algorithms (`xGESDD`), where `x` here stands for the letters `S` (single precision), `D` (double precision), `C` (single precision complex) and `Z` (double precision complex). Read the documentation to see which one is more appropriate / more efficient for your particular problem, though both work for all matrices.

4. Practical uses of the SVD

See Chapter 5 of the textbook

Now comes the fun part: why and how should we use the SVD? Here are a number of practical uses for the SVD, ranging from simple, to very useful, to really cool.

4.1. Calculating the condition number of a matrix

As mentioned in one of the very first lectures, and proved in the last one, the condition number of a matrix is equal to the ratio of the first to the last singular value: for a general $m \times n$ matrix,

$$\mathrm{cond}(\mathbf{A}) = \frac{\sigma_1}{\sigma_p} \text{ where } p = \min(m, n) \tag{5.33}$$

This ratio is trivially computed once we have all the singular values. SVD is therefore the main way of estimating the condition number of a matrix. We generally say that a matrix is well conditioned if $\mathrm{cond}(\mathbf{A})$ is close to one, poorly conditioned if $1 \ll \mathrm{cond}(\mathbf{A}) \ll \epsilon_{\mathrm{mach}}^{-1}$, ill-conditioned if $\mathrm{cond}(\mathbf{A}) = O(\epsilon_{\mathrm{mach}}^{-1})$ and singular if $\mathrm{cond}(\mathbf{A}) \to \infty$.

4.2. Least Square Problems

Recall that a Least-Square problem is an overconstrained linear problem of the kind $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is an $m \times n$ matrix with $m > n$. It can be solved very easily and *robustly* with SVD. Indeed, first, let us re-write the problem using the normal equations, as

$$\mathbf{A}^*\mathbf{Ax} = \mathbf{A}^*\mathbf{b} \tag{5.34}$$

and then use the *reduced* SVD of $\mathbf{A}$ to rewrite this as

$$\mathbf{V}\hat{\mathbf{\Sigma}}\hat{\mathbf{U}}^*\hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\mathbf{V}^*\mathbf{x} = \mathbf{V}\hat{\mathbf{\Sigma}}\hat{\mathbf{U}}^*\mathbf{b} \tag{5.35}$$

which can be reduced further to

$$\hat{\mathbf{\Sigma}}\hat{\mathbf{\Sigma}}\mathbf{V}^*\mathbf{x} = \hat{\mathbf{\Sigma}}\hat{\mathbf{U}}^*\mathbf{b} \tag{5.36}$$

because $\mathbf{V}$ is by construction non-singular, and using the fact that $\hat{\mathbf{U}}^*\hat{\mathbf{U}} = \mathbf{I}$.

This time, as long as $\hat{\mathbf{\Sigma}}$ is non-singular (i.e. none of the singular values are zero), then we can invert this somewhat trivially to

$$\mathbf{x} = \mathbf{V}\hat{\mathbf{\Sigma}}^{-1}\hat{\mathbf{U}}^*\mathbf{b} \tag{5.37}$$

where $\hat{\mathbf{\Sigma}}^{-1}$ is the $n \times n$ diagonal matrix containing $1/\sigma_i$ as entries. In essence, the solution can be obtained simply by matrix multiplication.

The advantage of using SVD is that the problem can still be solved, at least approximately, *if the matrix* $\mathbf{A}$ *is singular*, i.e. some of the singular values are exactly zero, and therefore $\hat{\mathbf{\Sigma}}$ is singular. To proceed, we simply solve the first non-zero $r-$ rows of this set of equations, and leave the other variables as undetermined. This may seem weird, but there is a very simple geometric interpretation of the problem, and why this solution works.

Consider for instance a Least Square problem, in the weird situation where all the data you are trying to fit lies exactly on a straight line (and you have a lot of points on that line!), but you are somehow trying to fit the equation of a plane. One of the coefficients *must* remain undetermined because there is an infinite number of planes going through a single line. Hence, by solving the smaller problem and leaving the last coefficient undetermined, we can actually get the correct equation of that line!

This is quite important *even if your data does not exactly lie on a straight line.* Suppose *most* of your data lies close to a straight line (i.e, lies on a line with some small scatter around it), and you are trying to fit the equation of a plane – then, the matrix $\mathbf{A}$ will not be singular, but instead will be quite poorly conditioned. The SVD method will return very small singular values in its last entries, and you should use this information as a hint that perhaps it was a bad idea to try to fit a plane. The solution is to discard the equations associated with singular values that are too small, and only focus on those that have reasonable singular values.

4.3. Poorly conditioned exact problems

The remarks made in the previous section provide important insight on what to do with poorly-conditioned or ill-conditioned exact problems (by exact, we imply problems of the $\mathbf{Ax} = \mathbf{b}$ kind with $\mathbf{A}$ square). As discussed in Chapter 2, solving a poorly conditioned/ill-conditioned exact problem using the standard exact methods such as Gaussian Elimination or LU decomposition can be very problematic – either returning an answer that has very large errors, or not being able to return an answer at all (e.g. when the matrix appears to be singular in floating-point arithmetic even though it is not actually exactly singular).

In this case, SVD can be of useful to provide the answer. Using the SVD of $\mathbf{A}$, we write

$$\hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\mathbf{V}^*\mathbf{x} = \mathbf{b} \rightarrow \hat{\mathbf{\Sigma}}\mathbf{V}^*\mathbf{x} = \hat{\mathbf{U}}^*\mathbf{b} \tag{5.38}$$

As in the Least-Square case, if $\hat{\mathbf{\Sigma}}$ is exactly singular or contains singular values that are too small when compared with σ_1, we can simply solve a reduced problem containing the first s equations that have sufficiently large singular values. This will leave some of the entries of $\mathbf{x}$ undetermined, but it is better to know that these entries are undetermined, than to return answers that are wrong by a very large factor.

4.4. Low-rank approximation of matrices

We now come to the coolest application to SVD, namely the possibility of approximating large matrices with much smaller ones (called low-rank approximations).

The idea is the following. From the definition of the SVD, we have that

$$a_{ij} = \sum_{k,n} u_{ik}(\mathbf{\Sigma})_{kn} v^*_{jn} = \sum_{k=1}^{r} u_{ik}\sigma_k v^*_{jk} \tag{5.39}$$

since $\mathbf{\Sigma}$ is diagonal, containing non-zero entries from 1 to r. This can be rewritten in vector form as

$$\mathbf{A} = \sum_{k=1}^{r} \sigma_k \mathbf{B}_k \tag{5.40}$$

where the matrix $\mathbf{B}_k$ has components $(\mathbf{B}_k)_{ij} = u_{ik}v^*_{jk}$. As such the matrix $\mathbf{B}_1$ is formed by the outer product of the first column vectors of $\mathbf{U}$ and $\mathbf{V}$,

$$\mathbf{B}_1 = \mathbf{u}_1\mathbf{v}_1^* \tag{5.41}$$

the matrix $\mathbf{B}_2$ is formed by the outer product of the second column vectors,

$$\mathbf{B}_2 = \mathbf{u}_2\mathbf{v}_2^* \tag{5.42}$$

and so forth.

Since the singular values appearing in the sum from 1 to r gradually decrease with k, one may actually wonder if it is possible to ignore the terms in the sum for which σ_k is very small, i.e., to truncate the sum at an order ν that is smaller than r, and still have a reasonably good approximation for $\mathbf{A}$:

$$\mathbf{A} \simeq \sum_{k=1}^{\nu} \sigma_k \mathbf{B}_k, \quad \text{where } \nu < r. \tag{5.43}$$

The answer is not only yes, but it can be shown in fact that this approximation is the *best* possible approximation we can make of rank-r $\mathbf{A}$ using a matrix of rank $\nu < r$. This is stated in this theorem (that shall not be proved):

Theorem: If we define, for any $1 \leq \nu < r$ the matrix

$$\mathbf{A}_\nu = \sum_{k=1}^{\nu} \sigma_k \mathbf{u}_k \mathbf{v}_k^* \tag{5.44}$$

then

$$||\mathbf{A} - \mathbf{A}_\nu||_2 = \inf_{\mathbf{B},\ \text{rank}(\mathbf{B}) \leq \nu} ||\mathbf{A} - \mathbf{B}||_2 = \sigma_{\nu+1} \tag{5.45}$$

and

$$||\mathbf{A} - \mathbf{A}_\nu||_F = \inf_{\mathbf{B},\ \text{rank}(\mathbf{B}) \leq \nu} ||\mathbf{A} - \mathbf{B}||_F = \sqrt{\sigma^2_{\nu+1} + \cdots + \sigma^2_r} \tag{5.46}$$

This theorem implies that the norm of the residual $||\mathbf{A} - \mathbf{A}_\nu||$ has the smallest possible norm among all possible $||\mathbf{A} - \mathbf{B}||$ created using matrices $\mathbf{B}$ that have rank ν or less. In other words, $\mathbf{A}_\nu$ is the best possible matrix of rank ν that can be created to approximate $\mathbf{A}$ in both Euclidean and Frobenius norm sense. This is incredibly powerful, and can be used in a vast number of applications.

Example: Image compression. Consider for instance image compression: a black and white image is simply an $m \times n$ matrix with entries a_{ij} measuring the grey-level associated with pixel located at position $(x_i,\ y_j)$. Usually, an image will have several million pixels, with m and n well in excess of a few thousand pixels in each direction, so r can be extremely large. As it turns out, by doing an SVD of the matrix $\mathbf{A}$, we can reconstruct the image quite accurately using $\nu = O(100)$ (sometimes more, sometimes less, depending on the actual image and the resolution required). This means that the image contained in the matrix $\mathbf{A}$ can be compressed by saving only the entries of a few hundred vectors $\mathbf{u}_i$ and $\mathbf{v}_i$, instead of all thousands of them. This is a huge compression rate!

Example: Low order models. SVD is often used to extract so-called low order models for complex systems. Used in this fashion, it is usually called with other names, such as Principle Component Analysis, Empirical Orthogonal Functions, Proper Orthogonal Decomposition (each field having reinvented the wheel in different ways, it seems). etc.. The idea is to take a system that has some complex spatiotemporal dependence, and to see whether that dependence can be approximated with the evolution of a few basic modes only – a little bit like the idea of SVD for image reconstruction, but now adding the full 3D dependence of a field, as well as its time dependence. To see how it works, imagine for instance working with a time-dependent 2D field. That field is resolved in a $m \times n$ map, and contains p snapshots of the map. One can then construct an $m \times np$ matrix (or, equivalently, an $mp \times n$ one), by stacking all the snapshots together. The SVD of that matrix will then pick up the spatiotemporal evolution of the modes that have the most energy, and that information can then be used to focus our modeling efforts on understanding the behavior of these few modes only (instead of the entirety of the complex system). Examples of use of this method in real-life applications include:

- Climate and weather monitoring/forecast (to identify recurrent patterns)
- Monitoring and modeling neurological activity
- Study of turbulent flows in engineering, solar cycle, etc.

Chapter 6

Iterative methods for the solution of linear systems

In this Chapter we now go back to the study of exact linear systems of the kind $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{A}$ an $m \times m$ matrix, and look at the possibility of designing *iterative* methods to solve this equation. We have seen the power of well-designed iterative methods in solving eigenvalue problems, and it is well worth asking whether such methods can be applied here too, with two considerations in mind:

- The basic schemes for the solution of linear systems are all schemes that take $O(m^3)$ floating point operations to find $\mathbf{x}$. This rapidly becomes *very* expensive for large matrices, and it is worth asking whether it is possible to design algorithms that iteratively converge to the right solution using N iterations of an $O(m^2)$ operation (e.g. a matrix multiplication), where $N \ll m$ (see Figure 1).
- The basic LU and Gaussian schemes are also inherently difficult to parallelize. Can iterative schemes be designed that are much easier to operate in parallel?

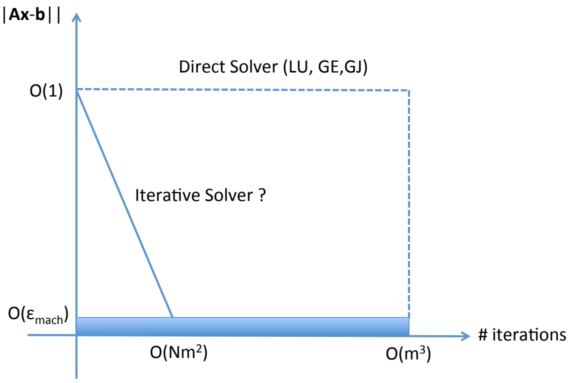


Figure 1. Illustration of the convergence of direct vs. (ideal) iterative solvers.

The answer to both questions is yes, if the matrix $\mathbf{A}$ is well-conditioned. Let us now learn a few examples of such algorithms.

1. Gauss-Jacobi, Gauss-Seidel and Successive Over-Relaxation algorithms

These three algorithms are effectively variants of the same basic idea, which is a nice introduction to the theme of iterative methods for exact equations. Let's begin with the Gauss-Jacobi method, which is the simplest one.

1.1. Gauss-Jacobi (or simply, Jacobi)

The idea behind the algorithm is to write $\mathbf{A} = \mathbf{D} + \mathbf{R}$ where $\mathbf{D}$ is a diagonal matrix containing the diagonal elements of $\mathbf{A}$, and $\mathbf{R}$ is the rest, i.e. a matrix whose diagonal elements are 0. Then,

$$(\mathbf{D} + \mathbf{R})\mathbf{x} = \mathbf{b} \rightarrow \mathbf{D}\mathbf{x} = \mathbf{b} - \mathbf{R}\mathbf{x} \rightarrow \mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}) \tag{6.1}$$

Note that since $\mathbf{D}$ is diagonal with elements a_{ii}, $\mathbf{D}^{-1}$ is known too (it is the diagonal matrix with elements $1/a_{ii}$).

Where does this leave us? Well, the key is that this can be viewed as an iterative algorithm, where we start with a guess $\mathbf{x}^{(0)}$, and apply

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}) \tag{6.2}$$

If (and this is a BIG if!) the algorithm converges, then the limit satisfies $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x})$, which is equivalent to $\mathbf{A}\mathbf{x} = \mathbf{b}$, and so $\mathbf{x}$ is indeed the solution we are looking for. Note that the operation count for each step is very low, as we can effectively write

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{m} r_{ij} x_j^{(k)}) \tag{6.3}$$

which contains exactly m multiplications/divisions per value of i (recalling that the diagonal r_{ij} are zero), hence a total of m^2 multiplications/divisions for each iteration.

We are therefore exactly in the situation envisaged: (1) if the algorithm converges in a number of steps $N \ll m$, then we can solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ in $O(Nm^2)$ steps instead of $O(m^3)$ steps, and (2) updating each component can be done independently of the others, so this is inherently parallelizable.

The big IF, however, needs to be resolved: why should this iterative algorithm converge at all? To answer this question, we need to look at the **spectral radius** of a matrix $\mathbf{M}$. Recall that the spectral radius of any square matrix $\mathbf{M}$ is $\rho(\mathbf{M}) = \max_i |\lambda_i|$. There are two important properties/theorems associated with the spectral radius of a matrix.

Property 1: If $\mathbf{M}$ is diagonalizable, and $\rho(\mathbf{M}) < 1$, then $\lim_{k\to\infty} \mathbf{M}^k\mathbf{x} = 0$ for any vector $\mathbf{x}$.

Proof: Since $||\mathbf{v}|| = 0$ iff $\mathbf{v} = 0$ for all $\mathbf{v}$, it suffices to show $||\mathbf{M}^k\mathbf{x}|| \to 0$ as $k \to \infty$. If $\mathbf{M}$ is diagonalizable, then we can write $\mathbf{x} = \sum_i \alpha_i \mathbf{v}_i$ where the $\mathbf{v}_i$ are the eigenvectors of $\mathbf{M}$ with eigenvalues λ_i. Then,

$$||\mathbf{M}^k\mathbf{x}|| = ||\sum_i \alpha_i \lambda_i^k \mathbf{v}_i|| \leq \rho(\mathbf{M})^k || \sum_i \alpha_i \mathbf{v}_i|| = \rho(\mathbf{M})^k ||\mathbf{x}|| \tag{6.4}$$

Since $\rho(\mathbf{M}) < 1$, then $\rho(\mathbf{M})^k \to 0$ as k tends to infinity, which completes the proof. □

Note that the condition $\mathbf{M}$ is diagonalizable is not particularly constraining as long as we allow for complex eigenvectors/eigenvalues, because it has been shown that the set of non-diagonalizable matrices in $\mathbb{C}^{m\times m}$ has measure 0 (i.e. almost all matrices are diagonalizable in $\mathbb{C}^{m\times m}$).

Property 2: If $\rho(\mathbf{M}) < 1$ then $(\mathbf{I} - \mathbf{M})^{-1}$ exists and

$$(\mathbf{I} - \mathbf{M})^{-1} = \sum_{k=0}^{\infty} \mathbf{M}^k \tag{6.5}$$

Proof: Let's consider the partial sum $\mathbf{B}_K = 1 + \mathbf{M} + \cdots + \mathbf{M}^K$. Then,

$$(\mathbf{I} - \mathbf{M})\mathbf{B}_K = (\mathbf{I} - \mathbf{M})(1 + \mathbf{M} + \cdots + \mathbf{M}^K) = 1 - \mathbf{M}^{K+1} \tag{6.6}$$

so, for any vector $\mathbf{x}$,

$$(\mathbf{I} - \mathbf{M})\mathbf{B}_K\mathbf{x} = \mathbf{x} - \mathbf{M}^{K+1}\mathbf{x} \tag{6.7}$$

Now, we just proved that if $\rho(\mathbf{M}) < 1$, $\lim_{K\to\infty} \mathbf{M}^K\mathbf{x} = 0$, which implies that

$$\lim_{K\to\infty} (\mathbf{I} - \mathbf{M})\mathbf{B}_K\mathbf{x} = \mathbf{x} \Rightarrow (\mathbf{I} - \mathbf{M}) \lim_{K\to\infty} \mathbf{B}_K = \mathbf{I} \tag{6.8}$$

This finally proves that the inverse of $\mathbf{I} - \mathbf{M}$ is $\lim_{K\to\infty} \mathbf{B}_K$, as required. □

We can now use both properties to find a necessary condition for convergence of the algorithm, and that condition is stated in the following theorem.

Theorem: The iterative algorithm

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{c} \tag{6.9}$$

where $\mathbf{T}$ is a square matrix and $\mathbf{c}$ is a constant vector, converges provided $\rho(\mathbf{T}) < 1$.

Before we prove it, note that it is indeed related to the algorithm we want to use, with $\mathbf{T} = -\mathbf{D}^{-1}\mathbf{R}$ and $\mathbf{c} = \mathbf{D}^{-1}\mathbf{b}$. The theorem would then imply that the algorithm converges provided $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$.

Proof: We can expand the recursion back to $k = 0$ as

$$\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)}+\mathbf{c} = \mathbf{T}(\mathbf{T}\mathbf{x}^{(k-1)}+\mathbf{c})+\mathbf{c} = \cdots = \mathbf{T}^{k+1}\mathbf{x}^{(0)}+(\mathbf{T}^{k}+\mathbf{T}^{k-1}+\cdots+\mathbf{T}+\mathbf{I})\mathbf{c} \tag{6.10}$$

Using the fact that $\rho(\mathbf{T}) < 1$, we have on the one hand that $\lim_{k\to\infty} \mathbf{T}^{k+1}\mathbf{x}^{(0)} = 0$, and on the other hand, that

$$\lim_{k\to\infty} \mathbf{T}^{k} + \mathbf{T}^{k-1} + \cdots + \mathbf{T} + \mathbf{I} = (\mathbf{I} - \mathbf{T})^{-1} \tag{6.11}$$

This then shows that

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = (\mathbf{I} - \mathbf{T})^{-1}\mathbf{c} \tag{6.12}$$

or in other words that $\mathbf{x}^{(k)}$ indeed converges. Furthermore, the limit is

$$\mathbf{x} = (\mathbf{I} - \mathbf{T})^{-1}\mathbf{c} \tag{6.13}$$

which is indeed the solution of $\mathbf{x} = \mathbf{T}\mathbf{x} + \mathbf{c}$. □

To summarize, we therefore have that the algorithm

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}) \tag{6.14}$$

converges provided $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$, and that the limit is the solution of the equation $(\mathbf{D} + \mathbf{R})\mathbf{x} = \mathbf{b}$. This is the **Gauss-Jacobi** algorithm.

Note that proving that $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$ requires calculating the largest eigenvalue of $\mathbf{D}^{-1}\mathbf{R}$, which can be done for instance using a basic power iteration on a single vector. In practice, however, this can be as expensive as actually applying the algorithm, so usually no-one bothers checking first – we just apply the Gauss-Jacobi algorithm, and if it converges, all is good (if it doesn't, too bad!).

As a rule of thumb, however, matrices whose diagonal coefficients are large compared with the non-diagonal ones are well-behaved from the point of view of Gauss-Jacobi convergence, because if $\mathbf{D}$ has large eigenvalues, then $\mathbf{D}^{-1}$ will have small ones, and so one may hope that $\mathbf{D}^{-1}\mathbf{R}$ will also have small eigenvalues. Matrices that satisfy $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$ are therefore called **diagonally dominant**. They are very common in problems that arise from the solution of PDEs for instance (e.g. the heat equation). In addition, the matrix $\mathbf{A}$ obtained in this case is usually banded, so calculating $\mathbf{R}\mathbf{x}$ is $O(m)$ instead of $O(m^2)$!

Algorithm: Gauss-Jacobi algorithm:

```
Create $\mathbf{D}$ and $\mathbf{R}$ from $\mathbf{A}$ Set $\mathbf{x}$ equal to your guess
do while $||\mathbf{r}|| >$ desired accuracy
    $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x})$ !  [Calculate new vector]
    $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ !  [Calculate a new residual]
enddo
```

Note that it should be obvious that in practice we do not actually create the matrices $\mathbf{D}^{-1}$ and $\mathbf{R}$ – the algorithm written as is is just a short-hand notation for the component wise formula given in equation (6.3), with $r_{ij} = a_{ij}$ unless $i = j$ in which case $r_{ij} = 0$.

One of the disadvantages of the Gauss-Jacobi algorithm is that its convergence is typically quite slow. On the other hand, as discussed earlier, it is also very easily parallelizable. So using it really depends on the application and computer architecture at hand!

1.2. Gauss-Seidel

The Gauss-Seidel algorithm is very similar to the Gauss-Jacobi algorithm but directly over-writes the vector $\mathbf{x}$ as the algorithm proceeds. The advantage is that it effectively uses the updated values of the coefficients of $\mathbf{x}$ as they come to calculate the next coefficients, and also does not require storing 2 vectors ($\mathbf{x}$ *and* $\mathbf{y}$). Effectively, instead of using (6.3) to compute $x_i^{(k+1)}$, the algorithm uses

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{m} a_{ij}x_j^{(k)}\right) \tag{6.15}$$

noting that $r_{ii} = 0$ again. In a compact form, the GS algorithm is written as

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)}) = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)}), \tag{6.16}$$

where $\mathbf{L}$ and $\mathbf{U}$ are respectively the lower and upper triangular matrices excluding the diagonal entries, resulting $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. In practice you can write it very simply as follows:

Algorithm: Gauss-Seidel algorithm:

```
Create $\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$ from $\mathbf{A}$ Set $\mathbf{x}$ equal to your guess
do while $||\mathbf{r}|| >$ desired accuracy
    $\mathbf{x} = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x})$ !  [Calculate new vector]
    $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ !  [Calculate a new residual]
enddo
```

The only remaining part here is to put together a meaningful exit strategy (i.e., how does the algorithm know it has converged).

The Gauss-Seidel algorithm generally has much better convergence properties than the Gauss-Jacobi algorithm. For instance, it has been shown that *if the Gauss-Jacobi algorithm converges*, then the Gauss-Seidel algorithm converges faster. Furthermore, the Gauss-Seidel algorithm sometimes converges for matrices for which Gauss-Jacobi does not converge. For instance, Gauss-Seidel has been shown to converge for any symmetric, positive definite matrix, which is not the case for Gauss-Jacobi. For serial algorithms, one should therefore always prefer the Gauss-Seidel algorithm. On the other hand, the algorithm is now no longer perfectly parallelizable, and so the gain in convergence rate may not outweigh the loss in scalability on parallel architectures. In short, the Gauss-Jacobi algorithm may remain preferable in that case.

1.3. Successive over-relaxation algorithm (SOR)

The SOR algorithm is a generalization of the Gauss-Seidel algorithm which can be fine-tuned to have a significantly faster convergence rate. To see how it works, first note that we can re-write the Gauss-Seidel algorithm as

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)}\right) \tag{6.17}$$

where $\mathbf{L}$ and $\mathbf{U}$ are the lower-triangle and upper-triangles of the matrix $\mathbf{A}$ respectively (both have zeros on the diagonal).

Let's subtract $\mathbf{x}^{(k)}$ on both sides, to get

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)} - \mathbf{D}\mathbf{x}^{(k)}\right) \tag{6.18}$$

The term on the right-hand-side can therefore be viewed as the correction to be applied to $\mathbf{x}^{(k)}$ to get to $\mathbf{x}^{(k+1)}$.

The idea of over-relaxation is, as the name suggests, to *over*apply the correction, as in

$$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \omega\mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)} - \mathbf{D}\mathbf{x}^{(k)}\right) && (6.19) \\ &= (1-\omega)\mathbf{x}^{(k)} + \omega\mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)}\right) && (6.20) \end{aligned}$$

where ω is a factor between 1 and 2. Note that $\omega = 1$ simply recovers the Gauss-Seidel algorithm, which is why SOR is often viewed as a simple generalization of the latter.

A complete theory of convergence for this algorithm remains to be constructed, but we have the following theorems:

- Kahan (1958) proved that if $\omega \geq 2$ or $\omega \leq 0$, the algorithm necessarily diverges. This justifies the upper limit selected above.
- A further theorem states that *if* $\mathbf{A}$ *is positive definite* then the algorithm converges for any $0 < \omega < 2$, for any choice of initial vector.

- If $\mathbf{A}$ is positive definite *and* tridiagonal, then the optimal choice of ω for convergence is
$$\omega = \frac{2}{1+\sqrt{1-\rho(\mathbf{D}^{-1}\mathbf{R})^2}} \tag{6.21}$$
Note that actually computing $\rho(\mathbf{D}^{-1}\mathbf{R})^2$ may be too expensive in practice, unless $\mathbf{D}^{-1}\mathbf{R}$ is of a particular well-known form (which may well be the case for tridiagonal positive definite matrices arising from PDE problems).

The subject of what the *optimal* value of ω for fastest-convergence is for any general linear system is still the subject of ongoing research, however. Some trial and error in selecting ω may be needed in that case.

Being based on Gauss-Seidel, it should be obvious that the SOR algorithm is not as naturally parallelizable as the Gauss-Jacobi algorithm. As a result, while it should always be preferred for serial algorithms, you should look at the trade-off between convergence rate and scalability to decide whether to use it in parallel.

2. The conjugate gradient method

As we will demonstrate in this Section, there exists an incredibly powerful iterative method for solving $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is a real, positive definite, symmetric matrix, for which convergence is not only guaranteed, but for which it is guaranteed to take *at most m* steps. In other words, this iterative method is *at its worse*, an $O(m^3)$ method, which is the same as direct methods. At its best it can find solutions as expected in $N \ll m$ steps, therefore providing a much faster way of finding the solution $\mathbf{x}$. This method is called the **conjugate gradient method**. In the following sections, we will gradually construct the method and see why it works.

2.1. Real symmetric positive definite linear systems and minimization

An interesting non-trivial interpretation of the linear system $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is a real, positive definite, symmetric matrix is that its solution $\mathbf{x}$ is *also* the solution of a minimization problem for a quadratic form.

Proof: To see this, consider the scalar function from $\mathbb{R}^m$ to $\mathbb{R}$,

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b} = \frac{1}{2}\sum_{j,k=1}^{m} x_j x_k a_{jk} - \sum_{j=1}^{m} x_j b_j \tag{6.22}$$

This function is effectively a homogeneous multidimensional quadratic function (a quadratic form), that can be shown to be convex when $\mathbf{A}$ is positive definite, and therefore has a single global minimum. To find this minimum, we calculate ∇f and set it to zero :

$$(\nabla f)_i = \sum_{k=1}^{m}\frac{1}{2}x_k a_{ik} + \sum_{j=1}^{m}\frac{1}{2}x_j a_{ji} - b_i = \sum_{j=1}^{m}(a_{ij}x_j - b_i) = (\mathbf{Ax}-\mathbf{b})_i = 0 \tag{6.23}$$

The solution to $\nabla f = 0$ is therefore also the solution to $\mathbf{Ax} = \mathbf{b}$. $\square$

2.2. Iterative methods for minimizing quadratic forms

An iterative method for minimizing a quadratic form starts at a given point $\mathbf{x}^{(0)}$, and applies an algorithm of the kind $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)})$ that gradually decreases the distance between $\mathbf{x}^{(k)}$ and the minimum. For a quadratic form that is known to be convex (as it is the case here) a simple way to proceed is to start going in a particular direction, and go to the minimum of $f(\mathbf{x})$ *along* that direction, e.g. see Figure 2. Then choose another direction, and minimize f along the new direction, and so forth.

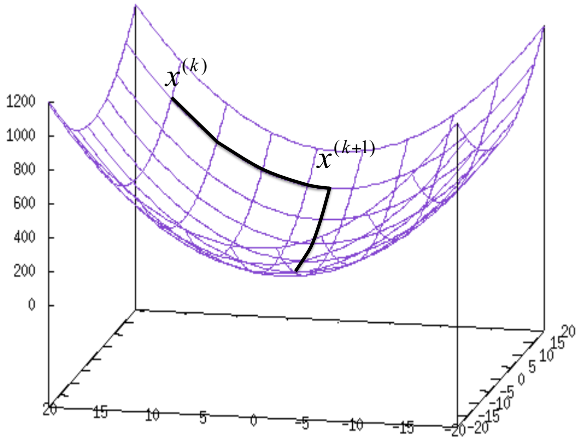


Figure 2. Minimization of f along a particular direction.

Minimizing f along a particular direction can be done analytically for a quadratic form. Indeed, suppose the selected direction vector at step k is $\mathbf{p}^{(k)}$, starting from vector $\mathbf{x}^{(k)}$. Then we know that the next iterate $\mathbf{x}^{(k+1)}$ satisfies

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \tag{6.24}$$

The key is to chose the scalar α_k so that $\mathbf{x}^{(k+1)}$ minimizes f along that line. Since

$$\begin{aligned} f(\mathbf{x}^{(k+1)}) &= \frac{1}{2}\mathbf{x}^{(k+1)T}\mathbf{A}\mathbf{x}^{(k+1)} - \mathbf{x}^{(k+1)T}\mathbf{b} \qquad (6.25) \\ &= \frac{1}{2}\left(\mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}\right)^T \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}) - (\mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)})^T\mathbf{b} \\ &= \frac{1}{2}\mathbf{x}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} - \mathbf{x}^{(k)T}\mathbf{b} \\ &+ \frac{\alpha_k}{2}\left(\mathbf{p}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} + \mathbf{x}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}\right) + \frac{\alpha_k^2}{2}\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)} - \alpha_k\mathbf{p}^{(k)T}\mathbf{b} \end{aligned}$$

we set $df(\mathbf{x}^{(k+1)})/d\alpha_k = 0$ to get

$$\mathbf{p}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)} - \mathbf{p}^{(k)T}\mathbf{b} = 0 \tag{6.26}$$

where we used the fact that $\mathbf{A}$ is symmetric to show that $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{x}^{(k)} = \mathbf{x}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}$. This equation has the solution

$$\alpha_k = \frac{\mathbf{p}^{(k)T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}} \tag{6.27}$$

where $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ is the residual at step k. In other words, as long as we have picked a direction $\mathbf{p}^{(k)}$ at step k, we can find the minimum of f along this direction analytically. The only remaining problem is *how to choose the directions* $\mathbf{p}^{(k)}$ *?*

2.3. Gradient descent

The most natural algorithm one can immediately come up with when picking the directions $\mathbf{p}^{(k)}$ is to find the direction of **steepest descent**, i.e. to go toward minus the gradient of f at $\mathbf{x}^{(k)}$ (recall that the gradient always points upward, and we want to go downward):

$$\mathbf{p}^{(k)} = -\nabla f(\mathbf{x}^{(k)}) = -(\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}) = \mathbf{r}^{(k)} \tag{6.28}$$

This suggests the following algorithm for the solution of real symmetric and positive definite linear systems by gradient descent:

Algorithm: Gradient descent for the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$:

```
Initialize guess $\mathbf{x} = 0$
$\mathbf{r} = \mathbf{b}$
do while $||\mathbf{r}|| >$ desired accuracy
    $\mathbf{p} = \mathbf{r}$  !  [Set direction]
    $\alpha = \frac{\mathbf{p}^T\mathbf{r}}{\mathbf{p}^T\mathbf{A}\mathbf{p}}$  !  [Find optimal $\alpha$]
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$  !  [Update $\mathbf{x}$]
    $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$  !  [Calculate new residual]
enddo
```

Example: Consider the system

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{6.29}$$

We then create the function

$$f(\mathbf{x}) = x^2 + xy + y^2 - x - 2y \tag{6.30}$$

The contours of this function in the (x, y) plane are shown in Figure 4, as well as the trajectory traced by the points $\mathbf{x}^{(k)}$ gradually obtained by the algorithm above. Starting from $\mathbf{x}^{(0)} = (0, 0)$, we follow the direction of the gradient toward the minimum. Since the gradient is perpendicular to the contours of f, we basically leave perpendicularly to the contour. We stop when the function f begins to increase again, which happens exactly when the line becomes tangent to a contour. This is $\mathbf{x}^{(1)}$. We repeat the algorithm, and gradually approach the true solution, which is $(0, 1)$.

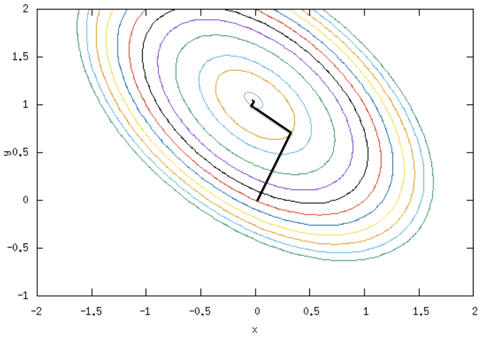


Figure 3. Gradient descent trajectory for example 1.

We see that it takes more than 2 steps to get to a good answer for this 2D problem – which is not great, because 2 is exactly the dimension of $\mathbf{A}$. This slow convergence only gets worse for more poorly conditioned matrices, where convergence rate is very slow. The problem with gradient descent, as we see here, is that the algorithm revisits a direction many times. In what follows, we therefore look at another strategy which guarantees convergence in at most m steps, by making sure that directions are never re-visited. This is the principle behind the **conjugate gradient method**.

2.4. The conjugate gradient method

In order to understand the conjugate gradient method, we first need to introduce the notion of **conjugate directions**.

Definition: Two non-zero vectors **u** and **v** are said to be **A-conjugate** if

$$\mathbf{u}^T\mathbf{A}\mathbf{v} = 0 \tag{6.31}$$

This basically identifies pairs of vectors where one is orthogonal to the *image* of the other after application of **A**. Since $\mathbf{u}^T\mathbf{A}\mathbf{v}$ looks somewhat like an inner product, note that if **A** is positive definite we can also define a new norm:

$$||\mathbf{u}||_A = \sqrt{\mathbf{u}^T\mathbf{A}\mathbf{u}} \tag{6.32}$$

which will be useful later.

Definition: A set of vectors $\{\mathbf{p}_i\}$ forms a **conjugate set** (with respect to the matrix **A**) provided

$$\mathbf{p}_i^T\mathbf{A}\mathbf{p}_j = 0 \text{ for all } i \neq j \tag{6.33}$$

For example, if $\mathbf{A}$ is a real symmetric matrix of size $m \times m$, it is easy to see that the eigenvectors form a conjugate set, which in turns immediately shows that *any real symmetric matrix has a conjugate set* $\{\mathbf{p}_i\}_{i=0...m-1}$, and this set forms a basis for $\mathbb{R}^m$.

Suppose we had somehow already found a conjugate set for $\mathbf{A}$. We could then write the solution $\mathbf{x}$ of $\mathbf{Ax} = \mathbf{b}$ as

$$\mathbf{x} = \sum_{i=0}^{m-1} \alpha_i \mathbf{p}_i \tag{6.34}$$

Substituting this expression into $\mathbf{Ax} = \mathbf{b}$, then taking the inner product with $\mathbf{p}_j$, we get

$$\mathbf{p}_j^T \mathbf{A} \sum_{i=0}^{m-1} \alpha_i \mathbf{p}_i = \mathbf{p}_j^T \mathbf{b} \rightarrow \alpha_j = \frac{\mathbf{p}_j^T \mathbf{b}}{\mathbf{p}_j^T \mathbf{A} \mathbf{p}_j} \tag{6.35}$$

which essentially provides the solution $\mathbf{x}$ of the equation when we plug these back into (6.34). In practice, however, finding the set $\{\mathbf{p}_i\}_{i=0...m-1}$ is already an $O(m^3)$ task (if we have to find for instance the eigenvectors of $\mathbf{A}$), at which point it is as expensive as solving $\mathbf{Ax} = \mathbf{b}$ directly (and therefore pointless).

However, if we were somehow able to construct the vectors $\mathbf{p}_0$, $\mathbf{p}_1$, etc iteratively, so that $\mathbf{p}^{(0)} = \mathbf{p}_0$ is our starting vector, then $\mathbf{p}^{(1)} = \mathbf{p}_1$ is the next iterate, and so forth, and at each iteration let

$$\mathbf{x}^{(k+1)} = \sum_{i=0}^{k} \alpha_k \mathbf{p}^{(k)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \tag{6.36}$$

then we are effectively constructing successive approximations to $\mathbf{x}$, which we *know* must converge to $\mathbf{x}$ exactly in a finite number of iterations, since $\mathbf{x}^{(m)} = \mathbf{x}$. If, furthermore, the coefficients α_k decay very rapidly with k, then $\mathbf{x}^{(k)}$ may in fact tend to the true solution $\mathbf{x}$ much faster, using only $N \ll m$ terms in this sum, instead of all m of them. If that is the case, we would only need to compute N of the conjugate directions, and the operation count will be much smaller than $O(m^3)$. The crux of the problem, however, is to create an algorithm to find $\mathbf{p}^{(k+1)}$ given knowledge of $\mathbf{p}^{(k)}$.

To do this, let's try to inspire ourselves from the gradient descent algorithm, but modify it slightly to make sure the directions $\mathbf{p}^{(k)}$ that are iteratively created are conjugate to one another. At the very first step, we start as usual:

$$\mathbf{x}^{(0)} = 0 \text{ and } \mathbf{p}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} \tag{6.37}$$

therefore picking the direction of steepest descent. Then we construct

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \alpha_0 \mathbf{p}^{(0)} \tag{6.38}$$

as before, where α_0 is chosen to minimize f along the $\mathbf{p}^{(0)}$ direction. At the next step, steepest descent would normally pick

$$\mathbf{p}^{(1)} = \mathbf{r}^{(1)} = \mathbf{b} - \mathbf{Ax}^{(1)} \tag{6.39}$$

However, It is easy to see that this is not conjugate to $\mathbf{p}^{(0)}$. Indeed,

$$\mathbf{p}^{(1)T}\mathbf{A}\mathbf{p}^{(0)} = \mathbf{r}^{(1)T}\mathbf{A}\mathbf{b} = (\mathbf{b} - \mathbf{A}\mathbf{x}^{(1)})^T\mathbf{A}\mathbf{b} = \mathbf{b}^T\mathbf{A}\mathbf{b} - \mathbf{x}^{(1)T}\mathbf{A}^T\mathbf{A}\mathbf{b} \tag{6.40}$$

which has no reason to be zero. However, suppose instead we wrote

$$\mathbf{p}^{(1)} = \mathbf{r}^{(1)} + \beta_0\mathbf{p}^{(0)} \tag{6.41}$$

that is, we shift $\mathbf{p}^{(1)}$ slightly away from steepest descent, and *choose* β_0 *so that* $\mathbf{p}^{(1)T}\mathbf{A}\mathbf{p}^{(0)} = 0$. This simply requires picking

$$\mathbf{p}^{(1)T}\mathbf{A}\mathbf{p}^{(0)} = (\mathbf{r}^{(1)} + \beta_0\mathbf{p}^{(0)})^T\mathbf{A}\mathbf{p}^{(0)} = 0 \rightarrow \beta_0 = -\frac{\mathbf{r}^{(1)T}\mathbf{A}\mathbf{p}^{(0)}}{\mathbf{p}^{(0)T}\mathbf{A}\mathbf{p}^{(0)}} \tag{6.42}$$

Note that, with this choice, we have that

$$\mathbf{r}^{(1)T}\mathbf{r}^{(0)} = 0 \tag{6.43}$$

because

$$\begin{aligned}
\mathbf{r}^{(1)T}\mathbf{r}^{(0)} \quad &= (\mathbf{b} - \mathbf{A}\mathbf{x}^{(1)})^T\mathbf{r}^{(0)} = \mathbf{b}^T\mathbf{r}^{(0)} - (\mathbf{x}^{(0)} + \alpha_0\mathbf{p}^{(0)})^T\mathbf{A}^T\mathbf{r}^{(0)} \\
&= (\mathbf{b} - \mathbf{A}\mathbf{x}^{(0)})^T\mathbf{r}^{(0)} - \alpha_0\mathbf{p}^{(0)T}\mathbf{A}\mathbf{r}^{(0)} \\
&= \mathbf{r}^{(0)T}\mathbf{r}^{(0)} - \alpha_0\mathbf{p}^{(0)T}\mathbf{A}\mathbf{r}^{(0)} = 0
\end{aligned} \tag{6.44}$$

using the symmetry of $\mathbf{A}$, and by definition of α_0.

By extension, we see that it may indeed be possible to construct a sequence of conjugate directions from the following rough algorithm:

- $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}$ where $\alpha_k = \frac{\mathbf{p}^{(k)T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}}$
- $\mathbf{r}^{(k+1)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)}$
- $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k\mathbf{p}^{(k)}$ where $\beta_k = -\frac{\mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(k)}}{\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}}$

To be sure that this algorithm is indeed doing the right thing, we must prove that at every iteration, the new direction created is conjugate to all the previous ones. To do this, we will simultaneously prove that at each iteration

- $\mathbf{r}^{(k)T}\mathbf{p}^{(i)} = 0$ for any $i < k$. In other words, the residuals are orthogonal to the previous directions
- $\mathbf{r}^{(k)T}\mathbf{r}^{(i)} = 0$ for any $i < k$. In other words, the residuals are orthogonal to each other
- $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} = 0$ for any $i < k$. In other words, the directions form a conjugate set

Proof by induction: At step 1, we have that $\mathbf{r}^{(1)T}\mathbf{p}^{(0)} = \mathbf{r}^{(1)T}\mathbf{r}^{(0)} = 0$ (see proof earlier), and we have constructed β_0 so that $\mathbf{p}^{(1)T}\mathbf{A}\mathbf{p}^{(0)} = 0$. Let's now assume that these statements are true at step k, and prove them at step $k+1$.

To do so, first, let's note that

$$\mathbf{r}^{(k+1)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) = \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)} \tag{6.45}$$

so

$$\mathbf{r}^{(k+1)T}\mathbf{p}^{(i)} = (\mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)})^T \mathbf{p}^{(i)} = \mathbf{r}^{(k)T}\mathbf{p}^{(i)} - \alpha_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} \tag{6.46}$$

At this point, there are two possibilities: either $i < k$, in which case we know by the induction assumption that $\mathbf{r}^{(k)T}\mathbf{p}^{(i)} = 0$ and $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} = 0$. Or $i = k$, in which case $\mathbf{r}^{(k+1)T}\mathbf{p}^{(k)} = \mathbf{r}^{(k)T}\mathbf{p}^{(k)} - \alpha_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)}$. But this must be equal to zero by the definition of α_k.

Next, we evaluate

$$\mathbf{r}^{(k+1)T}\mathbf{r}^{(i)} = (\mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)})^T \mathbf{r}^{(i)} = \mathbf{r}^{(k)T}\mathbf{r}^{(i)} - \alpha_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{r}^{(i)} \tag{6.47}$$

Noting that $\mathbf{r}^{(i)} = \mathbf{p}^{(i)} - \beta_{i-1}\mathbf{p}^{(i-1)}$ (from the algorithm) then this is also

$$\mathbf{r}^{(k+1)T}\mathbf{r}^{(i)} = \mathbf{r}^{(k)T}\mathbf{r}^{(i)} - \alpha_k \mathbf{p}^{(k)T}\mathbf{A}(\mathbf{p}^{(i)} - \beta_{i-1}\mathbf{p}^{(i-1)}) \tag{6.48}$$

There are then two possibilities: either $i < k$, in which case by the induction assumptions each term is zero, or, $i = k$, in which case the third term is zero, and we are left with

$$\mathbf{r}^{(k+1)T}\mathbf{r}^{(k)} = \mathbf{r}^{(k)T}\mathbf{r}^{(k)} - \alpha_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)} = 0 \tag{6.49}$$

by the definition of α_k (once again).

Finally, let's evaluate $\mathbf{p}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)}$. We have

$$\mathbf{p}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)} = (\mathbf{r}^{(k+1)} + \beta_k \mathbf{p}^{(k)})^T \mathbf{A}\mathbf{p}^{(i)} = \mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)} + \beta_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} \tag{6.50}$$

Again, there are two possibilities: either $i < k$, in which case the second term is zero, and we are left with

$$\mathbf{p}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)} = \mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)} = \mathbf{r}^{(k+1)T}\left(\frac{\mathbf{r}^{(i)} - \mathbf{r}^{(i+1)}}{\alpha_i}\right) = 0 \tag{6.51}$$

since we have already proved that $\mathbf{r}^{(k+1)}$ is orthogonal to any residual $\mathbf{r}^{(i)}$ with $i \leq k$. Or, $i = k$ in which have we have

$$\mathbf{p}^{(k+1)T}\mathbf{A}\mathbf{p}^{(i)} = \mathbf{r}^{(k+1)T}\mathbf{A}\mathbf{p}^{(k)} + \beta_k \mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(k)} \tag{6.52}$$

but that is zero by the definition of β_k. □

This rather cumbersome proof shows that that directions created form the required conjugate set. This finally leads to the following theorem.

Theorem: The conjugate gradient algorithm, starting from $\mathbf{x}^{(0)} = 0$, $\mathbf{r}^{(0)} = \mathbf{p}^{(0)} = \mathbf{b}$, and applying the iterations described above, converges to the true solution in at most m iterations. The residuals are orthogonal to one another with $\mathbf{r}^{(k)T}\mathbf{r}^{(i)} = 0$ for any $i < k$, and the directions are conjugate to one another with $\mathbf{p}^{(k)T}\mathbf{A}\mathbf{p}^{(i)} = 0$ for any $i < k$. We also have the identity of the following **Krylov** subspaces:

$$\begin{aligned}\mathcal{K}^{(k)} &= \langle \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \dots, \mathbf{x}^{(k)} \rangle = \langle \mathbf{p}^{(0)}, \mathbf{p}^{(1)} \dots, \mathbf{x}^{(k-1)} \rangle \\ &= \langle \mathbf{r}^{(0)}, \mathbf{r}^{(1)} \dots, \mathbf{r}^{(k-1)} \rangle = \langle \mathbf{b}, \mathbf{A}\mathbf{b} \dots, \mathbf{A}^k\mathbf{b} \rangle \end{aligned} \tag{6.53}$$

Finally, if we consider the error $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$ (i.e. the difference between the true solution and the approximate solution at step k), then

$$||\mathbf{e}^{(k)}||_A = \inf_{\mathbf{u}\in\mathcal{K}^{(k)}} ||\mathbf{x} - \mathbf{u}||_A \tag{6.54}$$

(or in other words, $\mathbf{x}^{(k)}$ is the best possible approximation of the true solution $\mathbf{x}$ that lives in the subspace $\mathcal{K}^{(k)}$, at least when that distance is measured with the norm $||\cdot||_A$).

Much of the content of this theorem has already been proved, or is relatively trivial to prove (as the equivalence of the subspaces for instance). The only remaining item, which is also the most important aspect of the conjugate gradient algorithm, is the proof that the solution at the $k-$th iteration actually minimizes the distance to the true solution among all possible vectors in the Krylov subspace $\mathcal{K}^{(k)}$. To show this, consider an arbitrary vector $\mathbf{u}$ in $\mathcal{K}^{(k)}$. Then

$$\begin{aligned}||\mathbf{x} - \mathbf{u}||_A^2 &= ||\mathbf{e}^{(k)} + \mathbf{x}^{(k)} - \mathbf{u}||_A^2 = (\mathbf{e}^{(k)} + \mathbf{x}^{(k)} - \mathbf{u})^T\mathbf{A}(\mathbf{e}^{(k)} + \mathbf{x}^{(k)} - \mathbf{u}) \\ &= ||\mathbf{e}^{(k)}||_A^2 + 2(\mathbf{x}^{(k)} - \mathbf{u})^T\mathbf{A}\mathbf{e}^{(k)} + (\mathbf{x}^{(k)} - \mathbf{u})^T\mathbf{A}(\mathbf{x}^{(k)} - \mathbf{u}) \end{aligned} \tag{6.55}$$

Noting that (1) $\mathbf{A}\mathbf{e}^{(k)} = \mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{r}^{(k)}$, (2) $\mathbf{r}^{(k)}$ is perpendicular to all the vectors in $\mathcal{K}^{(k)}$, and (3) that $\mathbf{x}^{(k)} - \mathbf{u} \in \mathcal{K}^{(k)}$, we have that $2(\mathbf{x}^{(k)} - \mathbf{u})^T\mathbf{A}\mathbf{e}^{(k)} = 0$, leaving

$$||\mathbf{x} - \mathbf{u}||_A^2 = ||\mathbf{e}^{(k)}||_A^2 + ||\mathbf{x}^{(k)} - \mathbf{u}||_A^2 \tag{6.56}$$

Since this is the sum of two squares, this quantity reaches a minimum for all possible $\mathbf{u}$ in $\mathcal{K}^{(k)}$ when $\mathbf{u} = \mathbf{x}^{(k)}$. □

The implications of the theorem are quite profound. For instance, the last statement immediately implies that

$$||\mathbf{e}^{(k+1)}||_A \leq ||\mathbf{e}^{(k)}||_A \tag{6.57}$$

since we are expanding the space over which $||\mathbf{x} - \mathbf{u}||_A$ is minimized at each iteration. One important question, which we have not addressed yet, is the rate of convergence of the algorithm. In that regard, it can be shown (see textbook) that

$$||\mathbf{e}^{(k)}||_A \leq \frac{2}{\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-k}} ||\mathbf{e}^{(0)}||_A \tag{6.58}$$

where

$$\kappa = \mathrm{cond}(\mathbf{A}) = \frac{\sigma_1}{\sigma_m} \tag{6.59}$$

Note that since κ is usually large-ish to very large, this is often written more simply as

$$||\mathbf{e}^{(k)}||_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k ||\mathbf{e}^{(0)}||_A \tag{6.60}$$

We see that the rate of convergence depends on how small $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ is. If κ is too large, then this number is close to 1 and the convergence rate will be slow. On the other hand if κ is close to 1, then the convergence rate can be very fast.

Finally, here is a basic version of the algorithm. The most expensive step (which is $O(m^2)$) is the multiplications of $\mathbf{A}$ with the vector $\mathbf{p}$, which is done 3 times. For this reason, we first construct $\mathbf{y} = \mathbf{Ap}$, and save it.

Algorithm: Basic Conjugate gradient algorithm:

```
$\mathbf{x} = \mathbf{x}_0$, $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$  $\mathbf{p} = \mathbf{r}$,
Calculate $||\mathbf{r}||$
do while $||\mathbf{r}|| >$ desired accuracy
    $\mathbf{y} = \mathbf{Ap}$  !  [Calculate temporary vector to save time]
    $\alpha = \frac{\mathbf{p}^T\mathbf{r}}{\mathbf{p}^T\mathbf{y}}$
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$  !  [Uses this formula for $\mathbf{r}$ which uses $\mathbf{y}$, avoids
calculating $\mathbf{Ax}$]
    Calculate $||\mathbf{r}||$
    $\beta = -\frac{\mathbf{r}^T\mathbf{y}}{\mathbf{p}^T\mathbf{y}}$
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$
enddo
```

Note how here we have allowed the algorithm to start at a value of $\mathbf{x}$ that is different from 0, which can be useful when we already know a good approximation to the solution.

A smarter version of this algorithm also exists, which saves a little time on the computation of α and β:

Algorithm: Smart Conjugate gradient algorithm:

```
$\mathbf{x} = \mathbf{x}_0$,  $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$,  $\mathbf{p} = \mathbf{r}$,
Calculate $E = ||\mathbf{r}||^2$
```

```
do while $\sqrt{E} >$ desired accuracy
    $\mathbf{y} = \mathbf{A}\mathbf{p}$ !  [Calculate temporary vector, store in $\mathbf{y}$]
    $\alpha = \frac{E}{\mathbf{p}^T\mathbf{y}}$
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$ !  [Uses this formula for $\mathbf{r}$ which uses $\mathbf{y}$, avoids calculating $\mathbf{Ax}$]
    Calculate $E_{\text{new}} = ||\mathbf{r}||^2$
    $\beta = \frac{E_{\text{new}}}{E}$
    $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$
    $E = E_{\text{new}}$
enddo
```

To show that this is indeed equivalent to the previous one it suffices to show that, at iteration k, $\mathbf{r}^{(k)T}\mathbf{p}^{(k)} = \mathbf{r}^{(k)T}\mathbf{r}^{(k)}$, and that $\beta_k = \frac{\mathbf{r}^{(k+1)T}\mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}$. This can be done by induction (see project).

Finally, in cases where $\mathbf{A}$ is a sparse matrix, then it is advantageous to write your own custom matrix multiplication algorithm for the matrix $\mathbf{A}$. If the matrix is banded, for instance (e.g. tridiagonal), this can reduce the operation count of the matrix multiplication step from $O(m^2)$ to $O(m)$.

Let's now back to the example given earlier, of the system

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{6.61}$$

and compare the performance of the two algorithms. We see that, while the gradient descent algorithm took 7 iterations to get to the true answer (within machine precision), the conjugate gradient algorithm takes 2 iterations (the maximum allowed). For well-behaved matrices, the performance of the conjugate gradient algorithm can be very fast!

3. Preconditioning for the conjugate gradient algorithm

See Chapter 40 of the textbook

As we have seen in the previous lectures, the conjugate gradient algorithm is guaranteed to find the solution to $\mathbf{Ax} = \mathbf{b}$ (for a real symmetric positive definite matrix $\mathbf{A}$) in at most m steps. However, we have also seen quite a few examples where it does indeed take exactly m steps, at which point the algorithm is as expensive as a direct solve. The question that we may ask is whether there may be a way to accelerate convergence. This question is actually quite general, and can also be raised in the context of other iterative algorithms (Gauss-Jacobi, Gauss-Seidel, SOR, but also the eigenvalue solvers discussed in Chapter 4).

To see how one may increase the convergence rate for the Conjugate Gradient algorithm, recall that it is directly related to the condition number $\kappa = \sigma_1/\sigma_m$

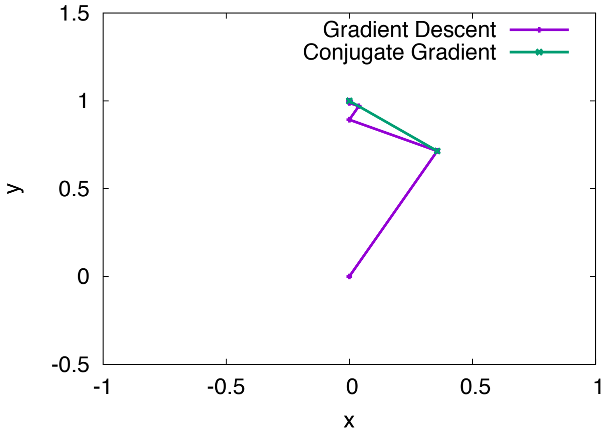


Figure 4. Comparison of Gradient Descent trajectory with Conjugate Gradient trajectory for example 1.

of the matrix $\mathbf{A}$. Furthermore, since $\mathbf{A}$ is real, symmetric and positive definite, they eigenvalues are equal to the singular values, so $\kappa = \lambda_1/\lambda_m$. If the eigenvalues differ widely in size, this number is large, and the matrix $\mathbf{A}$ is poorly conditioned so the algorithm converges very slowly. However, suppose for a moment that, instead of solving $\mathbf{Ax} = \mathbf{b}$, we solved

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b} \tag{6.62}$$

where $\mathbf{M}$ is an invertible matrix, chosen so that $\kappa(\mathbf{M}^{-1}\mathbf{A}) \ll \kappa(\mathbf{A})$. The two problems are equivalent, but the second would converge much more rapidly. The matrix $\mathbf{M}$ is thus called a **preconditioner**[1] for $\mathbf{A}$.

Choosing a good preconditioner for solving $\mathbf{Ax} = \mathbf{b}$ usually involves some trade-off. If $\mathbf{M} = \mathbf{A}$, then $\mathbf{M}^{-1}\mathbf{A} = \mathbf{I}$ and has condition number of 1(so the iteration would converge immediately). But computing $\mathbf{M}^{-1} = \mathbf{A}^{-1}$ is as hard a solving the original problem, so this is not very useful. On the other hand if $\mathbf{M} = \mathbf{I}$ then we can easily compute $\mathbf{M}^{-1}$ but we are not preconditioning at all. We therefore see that, in practice, we want to find a matrix $\mathbf{M}$ that is as close as possible to the original matrix $\mathbf{A}$ but that is also very easily inverted.

Finding a good preconditioner is an art, as there is no general theory or algorithm to help us generate it. In practice, different types of preconditioners are

[1] Note that it may seem a little weird to write $\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$, and then call $\mathbf{M}$ the preconditioner, rather than, say, writing $\mathbf{MAx} = \mathbf{Mb}$, but this is indeed the traditional way of doing it.

recommended for different types of matrices, there is no one-size-fits-all. Preconditioning is the subject of ongoing research. A nice review article on the subject is given in the paper by Wathen (see website). For the sake of looking at some examples, however, in this course we will use the **diagonal preconditioner**, which consists in letting $\mathbf{M}$ be the diagonal matrix that contains the diagonal elements of $\mathbf{A}$, and whose inverse is trivially computed:

$$\mathbf{M} = \begin{pmatrix} a_{11} & & \\ & a_{22} & \\ & & & a_{mm} \end{pmatrix} \Rightarrow \mathbf{M}^{-1} = \begin{pmatrix} a_{11}^{-1} & & \\ & a_{22}^{-1} & \\ & & & a_{mm}^{-1} \end{pmatrix} \tag{6.63}$$

Provided $\mathbf{A}$ *is diagonally dominant*, this preconditioner fits our requirements nicely : it is a good approximation to $\mathbf{A}$, and its inverse is very easy to find. Note, however, that it is sometimes also quite useless (see for instance the Project).

Even when we have a good preconditioner, there is a fairly major problem with preconditioning for the Conjugate Gradient algorithm, namely that there is no guarantee that $\mathbf{M}^{-1}\mathbf{A}$ should be positive definite or symmetric even when $\mathbf{A}$ is. For instance, even something as simple as the diagonal preconditioner can be problematic. In other words, we can't just multiply $\mathbf{Ax} = \mathbf{b}$ by $\mathbf{M}^{-1}$ and use the Conjugate Gradient algorithm to solve $\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$ for $\mathbf{x}$. With a little bit of extra work, however, it turns out that we can construct an equivalent Conjugate Gradient algorithm that uses preconditioning and doesn't run into this problem.

The idea begins with finding[2] a matrix $\mathbf{C}$ such that

$$\mathbf{M} = \mathbf{CC}^T \rightarrow \mathbf{M}^{-1} = \mathbf{C}^{-T}\mathbf{C}^{-1} \tag{6.64}$$

Then, the original problem $\mathbf{Ax} = \mathbf{b}$ becomes

$$\mathbf{C}^{-T}\mathbf{C}^{-1}\mathbf{Ax} = \mathbf{C}^{-T}\mathbf{C}^{-1}\mathbf{b} \rightarrow \mathbf{C}^{-1}\mathbf{Ax} = \mathbf{C}^{-1}\mathbf{b} \tag{6.65}$$

Finally, if we let $\tilde{\mathbf{A}} = \mathbf{C}^{-1}\mathbf{AC}^{-T}$, $\tilde{\mathbf{x}} = \mathbf{C}^T\mathbf{x}$ and $\tilde{\mathbf{b}} = \mathbf{C}^{-1}\mathbf{b}$, then this becomes $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. While the introduction of the tilde variable seems bizarre at first, it has the advantage that $\tilde{\mathbf{A}}$ is now symmetric and positive definite, as long as $\mathbf{A}$ was (this is very easy to prove). This suggests that we may be able to apply preconditioning to the Conjugate Gradient algorithm as follows:

- From $\mathbf{M}$, construct $\mathbf{C}$ and $\mathbf{C}^{-1}$
- Compute $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{b}}$
- Solve $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ using the Conjugate Gradient algorithm
- Let $\mathbf{x} = \mathbf{C}^{-T}\tilde{\mathbf{x}}$.

[2] In practice, we will not actually have to find this matrix, but let's assume it exists.

However, it should be fairly obvious that this would take far too long, and that it is not the right way to proceed. But the introduction of the tilde variables is still useful as a thought process, because it can be used to determine how to modify the original Conjugate Gradient algorithm to use preconditioning.

Indeed, let's explicitly write one iteration of the basic Conjugate Gradient algorithm in the preconditioned tilde variables:

1. $\tilde{\mathbf{y}} = \tilde{\mathbf{A}}\tilde{\mathbf{p}}$
2. $\tilde{\alpha} = \frac{\tilde{E}}{\tilde{\mathbf{p}}^T\tilde{\mathbf{y}}}$
3. $\tilde{\mathbf{x}} = \tilde{\mathbf{x}} + \tilde{\alpha}\tilde{\mathbf{p}}$
4. $\tilde{\mathbf{r}} = \tilde{\mathbf{r}} - \tilde{\alpha}\tilde{\mathbf{y}}$
5. $\tilde{E}_{\text{new}} = ||\tilde{\mathbf{r}}||^2$
6. $\tilde{\beta} = \frac{\tilde{E}_{\text{new}}}{\tilde{E}}$
7. $\tilde{E} = \tilde{E}_{\text{new}}$
8. $\tilde{\mathbf{p}} = \tilde{\mathbf{r}} + \tilde{\beta}\tilde{\mathbf{p}}$

Step 3 involves $\tilde{\mathbf{x}}$, so let's write it in terms of the original $\mathbf{x}$, to see whether we can design an algorithm that returns it directly instead of having to compute $\tilde{\mathbf{x}}$ first.

$$\tilde{\mathbf{x}} = \tilde{\mathbf{x}} + \tilde{\alpha}\tilde{\mathbf{p}} \rightarrow \mathbf{C}^T\mathbf{x} = \mathbf{C}^T\mathbf{x} + \tilde{\alpha}\tilde{\mathbf{p}} \rightarrow \mathbf{x} = \mathbf{x} + \tilde{\alpha}\mathbf{C}^{-T}\tilde{\mathbf{p}} \tag{6.66}$$

This suggest letting $\mathbf{p} = \mathbf{C}^{-T}\tilde{\mathbf{p}}$, in which case we have $\mathbf{x} = \mathbf{x} + \tilde{\alpha}\mathbf{p}$ as in the original algorithm (except for $\tilde{\alpha}$).

Next, let's look at Step 1: with the new $\mathbf{p}$, we have

$$\tilde{\mathbf{y}} = \tilde{\mathbf{A}}\tilde{\mathbf{p}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-T}\tilde{\mathbf{p}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{p} = \mathbf{C}^{-1}\mathbf{y} \tag{6.67}$$

as long as we redefine $\mathbf{y}$ such that $\tilde{\mathbf{y}} = \mathbf{C}^{-1}\mathbf{y}$. In that case, we then have $\mathbf{A}\mathbf{p} = \mathbf{y}$, as in the original algorithm.

Next, let's look at Step 4.

$$\tilde{\mathbf{r}} = \tilde{\mathbf{r}} - \tilde{\alpha}\tilde{\mathbf{y}} = \tilde{\mathbf{r}} - \tilde{\alpha}\mathbf{C}^{-1}\mathbf{y} \tag{6.68}$$

so if we let $\tilde{\mathbf{r}} = \mathbf{C}^{-1}\mathbf{r}$, then this becomes $\mathbf{r} = \mathbf{r} - \tilde{\alpha}\mathbf{y}$ as in the original algorithm (except for $\tilde{\alpha}$).

Next, let's look at Step 5:

$$\tilde{E}_{\text{new}} = \tilde{\mathbf{r}}^T\tilde{\mathbf{r}} = \mathbf{r}^T\mathbf{C}^{-T}\mathbf{C}^{-1}\mathbf{r} = \mathbf{r}^T\mathbf{M}^{-1}\mathbf{r} \tag{6.69}$$

This is not quite as in the original algorithm, but is still nice because it no longer contains $\mathbf{C}$ and $\mathbf{C}^{-1}$, but instead contains the original preconditioner inverse $\mathbf{M}^{-1}$. Similarly, Step 2 contains

$$\tilde{\mathbf{p}}^T\tilde{\mathbf{y}} = (\mathbf{C}^T\mathbf{p})^T\mathbf{C}^{-1}\mathbf{y} = \mathbf{p}^T\mathbf{C}\mathbf{C}^{-1}\mathbf{y} = \mathbf{p}^T\mathbf{y} \tag{6.70}$$

which is exactly the same in tilde and original variables.

Finally, Step 8 is rewritten as

$$\tilde{\mathbf{p}} = \tilde{\mathbf{r}} + \tilde{\beta}\tilde{\mathbf{p}} \to \mathbf{C}^T\mathbf{p} = \mathbf{C}^{-1}\mathbf{r} + \tilde{\beta}\mathbf{C}^T\mathbf{p} \to \mathbf{p} = \mathbf{M}^{-1}\mathbf{r} + \tilde{\beta}\mathbf{p} \tag{6.71}$$

which is nearly the same as the original one, except for the fact that $\mathbf{M}^{-1}$ now multiplies $\mathbf{r}$.

Putting this all together, we see that the algorithm re-written in terms of the non-tilde variables look very much like the original one, with the exception of a few places where the expression $\mathbf{M}^{-1}\mathbf{r}$ appears. This suggests the following corrections to the Smart Conjugate Gradient algorithm, to account for preconditioning:

Algorithm: Smart Preconditioned Conjugate gradient algorithm:

```
$\mathbf{x} = \mathbf{x}_0$,  $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$,
$\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$
$\mathbf{p} = \mathbf{z}$
$E = \mathbf{r}^T\mathbf{z}$
do while $\sqrt{E} >$ desired accuracy
    $\mathbf{y} = \mathbf{A}\mathbf{p}$
    $\alpha = \frac{E}{\mathbf{p}^T\mathbf{y}}$
    $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
    $\mathbf{r} = \mathbf{r} - \alpha\mathbf{y}$
    $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$
    $E_{\rm new} = \mathbf{r}^T\mathbf{z}$
    $\beta = \frac{E_{\rm new}}{E}$
    $\mathbf{p} = \mathbf{z} + \beta\mathbf{p}$
    $E = E_{\rm new}$
enddo
```

Note that in practice, it is convenient to do the operation $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$ outside of the actual algorithm (by calling a subroutine to do it, for instance), so different preconditioners $\mathbf{M}$ can easily be swapped in an out. The user then merely passes the name of that routine as argument to the solver. When done in this fashion, the beauty of this algorithm is that it is nearly identical to the original one (the modified steps are written in red for ease of identification), but can provide fairly dramatic acceleration with a good pre-conditioner $\mathbf{M}$. Note also how the matrices related to $\mathbf{C}$ never appear, and that the algorithm returns $\mathbf{x}$ directly.

4. The Generalized Minimal Residual Method

See Chapters 33 and 35 of the textbook

We now introduce an iterative method for solving the matrix equation $\mathbf{Ax} = \mathbf{b}$ called the Generalized Minimal RESidual method or GMRES for short. This method is related to the Conjugate Gradient method in the sense that it works by minimizing the error on the true solution within a Krylov subspace. However, there are significant differences, allowing the GMRES algorithm to be applicable for any matrix $\mathbf{A}$, provided it is a square $m \times m$ and an invertible matrix. The algorithm works by creating a sequence of subspaces $\mathcal{K}_n \subset \mathbb{R}^m$ of fixed dimension n and finding at each iteration the vector $\mathbf{x}_n$ such that $\|\mathbf{Ax}_n - \mathbf{b}\|_2 = \inf_{\mathbf{u}\in\mathcal{K}_n} \|\mathbf{Au} - \mathbf{b}\|_2$. Note that the choice of the norm is specifically set as the 2-norm since we will utilize the norm-preserving property of orthogonal matrices $\mathbf{Q}$, i.e., $\|\mathbf{QA}\|_2 = \|\mathbf{A}\|_2$. If $n \ll m$ and if the algorithm converges rapidly, then the total operation count is much smaller than that of a direct solver.

4.1. Krylov Subspace

The subspace that we will be creating for the GMRES algorithm is a Krylov subspace defined, in all generality, as follows:

Definition: The **Krylov subspace** of a matrix $\mathbf{A}$ with respect to a vector $\mathbf{v}$ is defined as

$$\mathcal{K}_n(\mathbf{A}, \mathbf{v}) := \text{span}\{\mathbf{v}, \mathbf{Av}, \mathbf{A}^2\mathbf{v}, \ldots, \mathbf{A}^{n-1}\mathbf{v}\} \subset \mathbb{R}^m \tag{6.72}$$

Definition: In addition, we also define the $m \times n$ **Krylov matrix**

$$\mathbf{K}_n = [\mathbf{v}|\mathbf{Av}|\ldots|\mathbf{A}^{n-1}\mathbf{v}]. \tag{6.73}$$

The Krylov space $\mathcal{K}_n(\mathbf{A}, \mathbf{v})$ is, therefore, the same as the column space of $\mathbf{K}_n$.

Let's now consider a preliminary version of the GMRES algorithm. Note that we wish to minimize $||\mathbf{b} - \mathbf{Ax}_n||_2$ with respect to some vector $\mathbf{x}_n \in \mathcal{K}_n(\mathbf{A}, \mathbf{v})$. Then there exists some vector $\mathbf{c} \in \mathbb{R}^n$ such that $\mathbf{K}_n\mathbf{c} = \mathbf{x}_n$. With this in mind, we can now rewrite the problem as a minimization problem of $\|\mathbf{b} - \mathbf{AK}_n\mathbf{c}||_2$. Note that $\mathbf{AK}_n$ is an $m \times n$ matrix where $m > n$, which means we are dealing with a Least Squares problem (i.e., an over-determined problem) $\mathbf{AK}_n\mathbf{c} \approx \mathbf{b}$. We can use any of the techniques from Chapter 3 to solve it for $\mathbf{c}$. As an example, if we use the reduced QR decomposition, we first construct the $m \times n$ orthogonal matrix $\hat{\mathbf{Q}}$ and the $n \times n$ upper-triangular matrix $\hat{\mathbf{R}}$ such that

$$\hat{\mathbf{Q}}\hat{\mathbf{R}} = \mathbf{AK}_n.$$

The problem becomes an exact problem using the orthogonal projector $\mathbf{p} = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T$,

$$\mathbf{A}\mathbf{K}_n\mathbf{c} = \mathbf{p}\mathbf{b} = \hat{\mathbf{Q}}\hat{\mathbf{Q}}^T\mathbf{b}.$$

Since $\mathbf{A}\mathbf{K}_n = \hat{\mathbf{Q}}\hat{\mathbf{R}}$, we solve

$$\hat{\mathbf{R}}\mathbf{c} = \hat{\mathbf{Q}}^T\mathbf{b}$$

for $\mathbf{c}$. Once $\mathbf{c}$ is known, we obtain the final solution from $\mathbf{x}_n = \mathbf{K}_n\mathbf{c}$, which is the closest vector to the true solution that lies in the Krylov subspace $\mathcal{K}_n(\mathbf{A}, \mathbf{v})$

You might rightfully be wondering what vector $\mathbf{v}$ to use to construct $\mathbf{K}_n$. You may even be wondering why we're using the Krylov subspace in the first place, as opposed to any other arbitrary subspace, as the basis for the minimization. After all, there is a non-negligible start-up cost associated with creating the Krylov subspace because of the repeated multiplication by $\mathbf{A}$. If we were to come up with arbitrary choices for subspaces that are less computationally expensive, we could quickly think of the span of the standard basis vectors, span$\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$ or the span of the columns of $\mathbf{A}$, span$\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n\}$. To see why the Krylov subspace is the ideal choice, we need to define the minimal polynomial.

Definition: If $\mathbf{A}$ is an $m \times m$ matrix, then the **minimal polynomial** $p(\mathbf{A})$ is defined as the unique monic (i.e., $\alpha_n = 1$) polynomial of least degree such that $p(\mathbf{A})\mathbf{v} = \mathbf{0}$ for all $\mathbf{v} \in \mathbb{R}^m$, i.e.,

$$\begin{aligned} p(\mathbf{A})\mathbf{v} &= \sum_{i=0}^{n} \alpha_i \mathbf{A}^i \mathbf{v} \text{ (where } \alpha_n = 1) && (6.74) \\ &= (\alpha_0\mathbf{I} + \alpha_1\mathbf{A} + \cdots + \mathbf{A}^n)\,\mathbf{v} && (6.75) \\ &= \mathbf{0} \end{aligned}$$

It isn't necessary to know any information about the minimal polynomial of $\mathbf{A}$ nor is it required that you know how to form them, but to make the concept more concrete, consider the following examples.

Example: Minimal Polynomial of Diagonal Matrices

- It is an easy exercise to verify that the minimal polynomial of $\mathbf{I}$ is $p(\mathbf{I}) = \mathbf{I} - \mathbf{I}$ (that is, with $\alpha_1 = 1$ and $\alpha_0 = -1$). Given any $\mathbf{v} \in \mathbb{R}^m$, we indeed have
$$(-\mathbf{I} + \mathbf{I})\mathbf{v} = \mathbf{0}.$$
- Let $\mathbf{A}$ be the 2×2 diagonal matrix with $a_{11} = 2$ and $a_{22} = 1$. Then the minimal polynomial of $\mathbf{A}$ is
$$(\mathbf{A} - \mathbf{I})(\mathbf{A} - 2\mathbf{I}) = \mathbf{A}^2 - 3\mathbf{A} + 2\mathbf{I}.$$

In these examples, you may have noticed that the degree of the minimal polynomial was, at most, equal to the number of rows/columns of the matrix. Even though the examples were simple, this turns out to always be the case.

Theorem: Given a matrix a real valued square matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$, the degree of the minimum polynomial n_A is most equal to m.

Using minimal polynomials, we can show why a Krylov subspace $\mathcal{K}_n(\mathbf{A}, \mathbf{v})$ is a good choice for a subspace to form our approximations from. If $p(\mathbf{A}) = \alpha_0 \mathbf{I} + \alpha_1 \mathbf{A} + \cdots + \alpha_{n-1} \mathbf{A}^{n_A - 1} + \mathbf{A}^{n_A}$ then we can write

$$\mathbf{I} = -\frac{1}{\alpha_0}\left(\alpha_1 \mathbf{A} + \cdots + \alpha_{n_A-1}\mathbf{A}^{n_A-1} + \mathbf{A}^{n_A}\right) = -\frac{1}{\alpha_0}\sum_{i=1}^{n_A} \alpha_i \mathbf{A}^i,$$

$$\Rightarrow \mathbf{A}^{-1} = -\frac{1}{\alpha_0}\left(\alpha_1 I + \cdots + \alpha_{n_A-1}\mathbf{A}^{n_A-2} + \mathbf{A}^{n_A-1}\right) = -\frac{1}{\alpha_0}\sum_{i=0}^{n_A-1} \alpha_{i+1} \mathbf{A}^i,$$

which implies that the true solution to $\mathbf{Ax} = \mathbf{b}$ can be written as the linear combination:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = -\frac{1}{\alpha_0}\sum_{i=0}^{n_A-1} \alpha_{i+1}\mathbf{A}^i \mathbf{b}. \tag{6.76}$$

This shows that the Krylov subspace $\mathcal{K}_{n_A}(\mathbf{A}, \mathbf{b})$ (i.e., the one based on the matrix $\mathbf{A}$ but also on the vector $\mathbf{b}$) is a natural basis over which one can span the true solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$.

It can be demonstrated – though we will not prove it – that the GMRES algorithm will converge to $\mathbf{x}$ to within high accuracy for $n \leq m$ when exact arithmetic is being used. However, in some cases, the algorithm might give the exact solution for $n \ll m$. This can happen when the degree of the **minimal polynomial with respect to the specific vector b** (denoted as $p_{\mathbf{b}}(\mathbf{A})$) is smaller than the degree of the minimal polynomial, $p(\mathbf{A})$.

Definition: The **minimal polynomial of A with respect to the vector b**, written as $p_{\mathbf{b}}(\mathbf{A})$, is defined as the unique monic polynomial of least degree n_b such that $p_{\mathbf{b}}(\mathbf{A})\mathbf{b} = \mathbf{0}$, i.e.,

$$p_{\mathbf{b}}(\mathbf{A}) = \left(\beta_0 \mathbf{I} + \beta_1 \mathbf{A} + \cdots + \mathbf{A}^{n_b}\right)\mathbf{b} = \mathbf{0}.$$

By construction, it should be clear that the degree n_b of $p_{\mathbf{b}}(\mathbf{A})$ is always less than or equal to the degree of $p(\mathbf{A})$. We can then write

$$\mathbf{x} = -\frac{1}{\beta_0}\sum_{i=0}^{n_b-1} \beta_{i+1}\mathbf{A}^i \mathbf{b}, \tag{6.77}$$

and we see that $\mathbf{x}$ can be found exactly by minimizing $||\mathbf{b} - \mathbf{Ax}||_2$ over $\mathcal{K}_{n_b}$ only, instead of over the whole space. If $n_b \ll m$, the subspace $\mathcal{K}_{n_b}$ has a very low dimension, and the GMRES algorithm becomes extremely efficient.

4.2. Faux-GMRES

The considerations of the previous section suggest the construction of the following algorithm:

Algorithm: Faux-GMRES Algorithm:

```
Select a value of $n \leq m$
Compute $\mathbf{K}_n = [\mathbf{b}|\mathbf{A}\mathbf{b}|, \ldots, |\mathbf{A}^{n-1}\mathbf{b}]$
Find $\mathbf{c}$ that minimizes $\|\mathbf{b} - \mathbf{A}\mathbf{K}_n\mathbf{c}\|_2$
$\mathbf{x}_n = \mathbf{K}_n\mathbf{c}$
```

In theory, if $n = n_A$ or $n = n_b$, this should be guaranteed to yield the true solution, and the hope is that one may actually obtain a very good approximation to the true solution for some $n \ll m$. In practice, though, because of floating point arithmetic, this turns out to be a very unstable numerical algorithm because vectors $\mathbf{A}^i\mathbf{b}$ become more and more parallel to the eigenvector associated with the largest eigenvalue (this is the power iteration method!). To make the algorithm more stable, we seek to create an orthonormal basis of vectors for the Krylov subspace. This process is referred to as Arnoldi's Method.

4.3. Arnoldi's Method

Our goal until now has been to minimize $\|\mathbf{b}-\mathbf{A}\mathbf{x}_n\|_2$ where $\mathbf{x}_n \in \mathcal{K}_n(\mathbf{A}, \mathbf{b})$. After forming the matrix $\mathbf{K}_n$ we can restate our goal to be to minimize $\|\mathbf{b} - \mathbf{A}\mathbf{K}_n\mathbf{c}\|_2$. The problem with this plan is that $\mathbf{K}_n$ becomes increasingly poorly conditioned as n grows larger, so the error on $\mathbf{c}$ becomes too large to be a useful algorithm.

On the other hand, if we can orthogonalize the columns of $\mathbf{K}_n$ into a new matrix $\mathbf{Q}_n$, then we can write $\mathbf{x}_n = \mathbf{Q}_n\mathbf{y}$ and then solve $\|\mathbf{b}-\mathbf{A}\mathbf{Q}_n\mathbf{y}\|_2$ without having to deal with a poorly conditioned matrix. Unfortunately, it isn't as straightforward as just orthogonalizing the vectors of $\mathbf{K}_n$ because, as n gets larger, the vectors become too close to parallel to each other, and thus standard orthogonalization techniques would fail to be reliable. To fix our problem we need to develop a technique to orthogonalize the vectors of $\mathbf{K}_n$ as each column is being formed as opposed to after all the columns have been formed. This is where a little ingenuity is required.

Let's take a little detour and first recall the Hessenberg form of a matrix $\mathbf{A}$,

$$\mathbf{A} = \mathbf{Q}\mathbf{H}\mathbf{Q}^T \qquad \text{or equivalently} \qquad \mathbf{A}\mathbf{Q} = \mathbf{Q}\mathbf{H} \tag{6.78}$$

where $\mathbf{Q}$ is an $m \times m$ orthogonal matrix, and $\mathbf{H}$ is the Hessenberg form of $\mathbf{A}$ (we already studied how to obtain $\mathbf{Q}$ in Chapter 4). If we write this equation in terms of its column vectors, then we have

$$[\mathbf{A}\mathbf{q}_1|, \ldots, |\mathbf{A}\mathbf{q}_{n-1}|\mathbf{A}\mathbf{q}_n|, \ldots, |\mathbf{A}\mathbf{q}_m] = [\mathbf{Q}\mathbf{h}_1|, \ldots, |\mathbf{Q}\mathbf{h}_n|\mathbf{Q}\mathbf{h}_{n+1}|, \ldots, |\mathbf{Q}\mathbf{h}_m] \tag{6.79}$$

Let $\mathbf{Q}_n$ be the matrix that contains the first n columns of $\mathbf{Q}$, and let $\mathbf{H}_n$ denote the matrix that contains the first $n+1$ rows and n columns of $\mathbf{H}$. The ingenuity we need to solve our problem is found by noting that

$$\mathbf{A}\mathbf{Q}_n = \mathbf{Q}_{n+1}\mathbf{H}_n \tag{6.80}$$

Proof: The first n columns of $\mathbf{AQ}$ are

$$[\mathbf{Aq}_1|\mathbf{Aq}_2|,\ldots,|\mathbf{Aq}_n] = [\mathbf{Qh}_1|\mathbf{Qh}_2|,\ldots,|\mathbf{Qh}_n]\,,$$

which can be rewritten as the following system of equations:

$$\begin{aligned}
\mathbf{Aq}_1 &= \mathbf{q}_1 h_{11} + \mathbf{q}_2 h_{21}\\
\mathbf{Aq}_2 &= \mathbf{q}_1 h_{12} + \mathbf{q}_2 h_{22} + \mathbf{q}_3 h_{32}\\
&\vdots\\
\mathbf{Aq}_n &= \sum_{i=1}^{n+1} \mathbf{q}_i h_{in}
\end{aligned}$$

since the entries $\mathbf{h}_i$ are zero for $j > i+1$. This is then in the form of (6.80). □

We can now use this to construct the next vector $\mathbf{q}_{n+1}$ given knowledge of the previous ones. Indeed if we solve for the last $\mathbf{q}_i$ term for each equation we have

$$\begin{aligned}
\mathbf{q}_2 &= \frac{\mathbf{Aq}_1 - \mathbf{q}_1 h_{11}}{h_{21}}\\
\mathbf{q}_3 &= \frac{\mathbf{Aq}_2 - \mathbf{q}_1 h_{12} - \mathbf{q}_2 h_{22}}{h_{32}}\\
&\vdots\\
\mathbf{q}_{n+1} &= \frac{\mathbf{Aq}_n - \sum_{i=1}^{n} \mathbf{q}_i h_{in}}{h_{n+1,n}}
\end{aligned}$$

where the denominators are found simply by normalization of the vectors $\mathbf{q}_j$, namely $h_{j+1,j} = \|\mathbf{Aq}_j - \sum_{i=1}^{j} \mathbf{q}_i h_{ij}\|_2$ for $j = 1,\ldots,n$. If we focus on the first of the equations above, we can see that if we want $\mathbf{q}_2$ to be orthonormal to $\mathbf{q}_1$ then we need to multiply both sides by $\mathbf{q}_1^T$ and set this to zero:

$$\begin{aligned}
\mathbf{q}_1^T \mathbf{q}_2 &= \frac{\mathbf{q}_1^T \mathbf{Aq}_1 - \mathbf{q}_1^T \mathbf{q}_1 h_{11}}{h_{21}}\\
\Rightarrow 0 &= \mathbf{q}_1^T \mathbf{Aq}_1 - \mathbf{q}_1^T \mathbf{q}_1 h_{11}\\
\Rightarrow h_{11} &= \mathbf{q}_1^T \mathbf{Aq}_1
\end{aligned}$$

since $\mathbf{q}_1$ is normalized. We can generalize this to obtain the conditions required for $\mathbf{q}_{n+1}$ to be orthogonal to all $\mathbf{q}_k$ with $k \le n$:

$$\begin{aligned}
\mathbf{q}_k^T \mathbf{q}_{n+1} &= \frac{\mathbf{q}_k^T \mathbf{Aq}_n - \sum_{i=1}^{n} \mathbf{q}_k^T \mathbf{q}_i h_{in}}{h_{n+1,n}}\\
\Rightarrow 0 &= \mathbf{q}_k^T \mathbf{Aq}_n - \sum_{i=1}^{n} \mathbf{q}_k^T \mathbf{q}_i h_{in} = \mathbf{q}_k^T \mathbf{Aq}_n - \mathbf{q}_k^T \mathbf{q}_k h_{kn}\\
\Rightarrow h_{kn} &= \frac{\mathbf{q}_k^T \mathbf{Aq}_n}{\mathbf{q}_k^T \mathbf{q}_k} = \mathbf{q}_k^T \mathbf{Aq}_n
\end{aligned}$$

where we have used the fact that the vectors $\mathbf{q}_k$ for $k = 1..n$ are already mutually orthogonal and normalized.

We are now equipped to write the Arnoldi Algorithm:

Algorithm: Arnoldi Algorithm (Gram-Schmidt version):

```
Choose a vector $\mathbf{q}_1$ such that $\|\mathbf{q}_1\|_2 = 1$
do $j = 1$ to $n$
    do $i = 1$ to $j$
        $h_{ij} = \mathbf{q}_i^T \mathbf{A} \mathbf{q}_j$
    enddo
    $\mathbf{q}_{j+1} = \mathbf{A}\mathbf{q}_j - \sum_{i=1}^{j} h_{ij}\mathbf{q}_i$
    $h_{j+1,j} = \|\mathbf{q}_{j+1}\|_2$
    if $h_{j+1,j} = 0$ then stop
    if $j < n$, then $\mathbf{q}_{j+1} = \frac{\mathbf{q}_{j+1}}{h_{j+1,j}}$
end do
```

Recall that the problem with the Faux-GMRES algorithm was that the matrix $\mathbf{K}_n$ was too poorly conditioned to use when we tried to solve our Least Squares problem. With the Arnoldi Algorithm we fix this by replacing $\mathbf{K}_n$ with the orthogonal matrix $\mathbf{Q}_n$. Please notice that if you compare the Arnoldi Algorithm to the Gram-Schmidt orthogonalization process, then you'll see that the steps are nearly identical. The big difference between them is that in Arnloldi's Algorithm we start with one vector, $\mathbf{q}_1$, and then create $\mathbf{q}_{i+1}$ from $\mathbf{q}_i$ instead of iterating over a known basis like with the Gram-Schmidt orthogonalization scheme.

Before continuing we need to address the fact that it is not clear that the column space of $\mathbf{Q}_n$ is the same as the column space of $\mathbf{K}_n$. Fortunately our next theorem will address this for us.

Theorem: Assuming that the Arnoldi Algorithm does not stop before the n-th step. Then the vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ form an orthonormal basis of the Krylov subspace $\mathcal{K}_n(\mathbf{A}, \mathbf{q}_1)$.

Proof: The vectors $\mathbf{q}_j$ for $j = 1, 2, \ldots, n$ are orthonormal by construction. We will show they span $\mathcal{K}_n(\mathbf{A}, \mathbf{q}_1)$ by showing that each $\mathbf{q}_j$ can be expressed as a polynomial of $\mathbf{A}$ of degree $j-1$, i.e. $\mathbf{q}_j = p_{j-1}(\mathbf{A})\mathbf{q}_1$ where p_{j-1} is a polynomial of degree $j-1$. We will accomplish this using strong induction.

For the base case note that $\mathbf{q}_1 = p_0(\mathbf{A})\mathbf{q}_1$ where $p_0(\mathbf{A}) = I$. Now assume that $\mathbf{q}_j = p_{j-1}(\mathbf{A})\mathbf{q}_1$ for all integers less than j. Consider $\mathbf{q}_{j+1}$. Note that

$$h_{j+1,j}\mathbf{q}_{j+1} = \mathbf{A}\mathbf{q}_j - \sum_{i=1}^{n} h_{ij}\mathbf{q}_i = \mathbf{A}p_{j-1}(\mathbf{A})\mathbf{q}_1 - \sum_{i=1}^{j} h_{ij}p_{i-1}(\mathbf{A})\mathbf{q}_1$$

Since $\mathbf{A}p_{j-1}(\mathbf{A})\mathbf{q}_1$ is a polynomial of degree j then we have demonstrated that $\mathbf{q}_{j+1}$ can be expressed as a polynomial of degree j. □

Finally, note that for numerical stability, we shouldn't use the standard Gram-Schmidt algorithm. Instead we should consider more numerically stable orthogonalization processes such as the modified Gram-Schmidt.

Algorithm: Arnoldi Algorithm (Modified Gram-Schmidt version):

```
Choose a vector $\mathbf{q}_1$ such that $\|\mathbf{q}_1\|_2 = 1$
do $j = 1$ to $n$:
    $\mathbf{q}_{j+1} = \mathbf{A}\mathbf{q}_j$
    do $i = 1$ to $j$
        $h_{ij} = \mathbf{q}_{j+1}^T\mathbf{q}_i$
        $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
    enddo
    $h_{j+1,j} = \|\mathbf{q}_{j+1}\|_2$
    If $h_{j+1,j} = 0$ then stop
    if $j < n$, then $\mathbf{q}_{j+1} = \frac{\mathbf{q}_{j+1}}{h_{j+1,j}}$
enddo
```

One final note: up until now, we have always used the vector $\mathbf{b}$ to create the Krylov subspace. However, moving forward, it will be desirable to choose our own vector. To do so, we will pick a vector $\mathbf{x}_0$ which we believe is a good guess for the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$. With our initial guess we then create the Krylov subspace $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$ where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$. So now, instead of minimizing $\|\mathbf{b} - \mathbf{A}\mathbf{Q}_n\mathbf{y}\|_2$ we will instead be minimizing $\|\mathbf{b} - \mathbf{A}(\mathbf{x}_0 + \mathbf{Q}_n\mathbf{y})\|_2$ and our solution $\mathbf{x}_n$ will be given by $\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Q}_n\mathbf{y}$.

4.4. GMRES Algorithm

Now that we know how to construct an orthogonal basis for the Krylov subspace we have everything we need to solve the Least Squares problem in a stable way. However, there is one more improvement we can make to our algorithm. Using the identities we've demonstrated earlier we are going to make minimizing $\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2$ less expensive. Recall that

$$\mathbf{A}\mathbf{Q}_n = \mathbf{Q}_{n+1}\mathbf{H}_n$$

using this allows us to write

$$\begin{aligned}
\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2 &= \|\mathbf{b} - \mathbf{A}(\mathbf{x}_0 + \mathbf{Q}_n\mathbf{y})\|_2 \\
&= \|\mathbf{r}_0 - \mathbf{A}\mathbf{Q}_n\mathbf{y}\|_2 \\
&= \|\beta\mathbf{q}_1 - \mathbf{Q}_{n+1}\mathbf{H}_n\mathbf{y}\|_2 \text{ where } \mathbf{q}_1 := \frac{\mathbf{r}_0}{\beta} \text{ and } \beta := \|\mathbf{r}_0\| \\
&= \|\beta\mathbf{Q}_{n+1}^T\mathbf{q}_1 - \mathbf{Q}_{n+1}^T\mathbf{Q}_{n+1}\mathbf{H}_n\mathbf{y}\|_2 \\
&= \|\beta\mathbf{e}_1 - \mathbf{H}_n\mathbf{y}\|_2
\end{aligned}$$

where, in the second to last step, we have used the fact that $\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2 = \|\mathbf{Q}_{n+1}^T(\mathbf{b} - \mathbf{A}\mathbf{x}_n)\|_2$ since $\mathbf{Q}_{n+1}$ is an orthonormal matrix. The minimizer for $\mathbf{y}_n$ is now much less expensive to compute since it requires solving the least squares problem involving the $(n+1) \times n$ matrix $\mathbf{H}_n$ instead of the $m \times n$ matrix $\mathbf{AQ}_n$. With this we can finally write the GMRES algorithm.

Algorithm: GMRES Algorithm:

```
Select $n$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\beta = \|\mathbf{r}_0\|_2$, $\mathbf{q}_1 = \mathbf{r}_0/\beta$
Call the Arnoldi algorithm to generate:
    (i) $\mathbf{Q}_n$
    (ii) $\mathbf{H}_n$ from $\mathbf{q}_i$
Solve the Least Square problem $\beta\mathbf{e}_1 - \mathbf{H}_n\mathbf{y}_n = 0$ for $\mathbf{y}_n$
$\mathbf{x}_n = \mathbf{x}_0 + \mathbf{Q}_n\mathbf{y}_n$
```

You may have noticed that this algorithm ends without knowing if we reached a desirable level of accuracy. To fix this, we can simply add a tweak to the algorithm above by implementing a “restart” by computing the error $\|\mathbf{b} - \mathbf{A}\mathbf{x}_n\|_2$ at the end of the algorithm and checking that it is less than a certain desired accuracy. If it does, then we’re done. If it doesn’t, then we define $\mathbf{x}_0 = \mathbf{x}_n$ from the previous iteration and start over.

Algorithm: Restart GMRES Algorithm:

```
Select $n$
Select guess $\mathbf{x}_0$
$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$, $\beta = \|\mathbf{r}_0\|_2$
do while $\beta >$ desired accuracy
    $\mathbf{q}_1 = \mathbf{r}_0/\beta$
    Call the Arnoldi algorithm to generate:
        (i) $\mathbf{Q}_n$
        (ii) $\mathbf{H}_n$ from $\mathbf{q}_i$
    Solve the Least Square problem $\beta\mathbf{e}_1 - \mathbf{H}_n\mathbf{y}_n = 0$ for $\mathbf{y}_n$
    $\mathbf{x}_0 = \mathbf{x}_0 + \mathbf{Q}_n\mathbf{y}_n$
    $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
    $\beta = \|\mathbf{r}_0\|_2$
end do
```

4.5. Convergence of GMRES

When it comes to convergence, there are certain matrices that will more quickly converge to the correct solution than the general matrix. In particular, if $\mathbf{A}$ is symmetric positive definite or if the symmetric part of $\mathbf{A}$ (the symmetric part of $\mathbf{A}$ is defined as $(\mathbf{A}^T + \mathbf{A})/2$) is positive definite, then GMRES will converge more quickly. It is also true that matrices with tightly clustered eigenvalues centered away from the origin will also converge more quickly. When dealing with matrices with slow convergence, preconditioning can be applied to speed up the convergence.