Systems of Linear Equations

1 A Quick Reminder on Linear Algebra

1.1 Matrices, column, and row vectors

A matrix is a rectangular array of numbers conventionally represented within parentheses or square brackets. It is not only the value of the numbers that matters, but also their position, indicated by a row number and a column number. The size of the matrix is defined by its number of rows and number of columns. A matrix with \( m \) rows and \( n \) columns is referred to as a \( m \times n \) matrix. In applied mathematics (and generally whenever a small number of symbols have to be re-used with different type faces) it is fairly common to use capital letters in boldface to refer to matrices.

\[
A = \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
  a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\
  \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn}
\end{pmatrix}.
\]

One frequently encounters the notation \( A = (a_{ij})_{i=1,\ldots,m} \) or \( A = [a_{ij}]_{j=1,\ldots,n} \). Sometimes, the index bounds are not even indicated when there is no ambiguity: \( A = (a_{ij}) \) or \( A = [a_{ij}] \).

A matrix with a single column (an \( m \times 1 \) matrix) is called a column vector. Boldfaced letters (but rarely capital letters) are also used to refer to column vectors.

\[
v = \begin{pmatrix}
  v_1 \\
v_2 \\
  \vdots \\
v_i \\
  \vdots \\
v_m
\end{pmatrix}.
\]

I would like to insist on the fact that a “column vector” is not a vector. A vector is a geometric entity that exists independently of any coordinate system, and therefore of any matrix representation.

Just as with matrices, we can use a compact representation for column vectors: \( v = (v_i)_{i=1,\ldots,m} \), or simply \( v = (v_i) \).

A matrix with a single row (a \( 1 \times n \) matrix) is called a row vector.

\[
c = (c_1, c_2, \ldots, c_n).
\]

Using a compact representation for row vectors would lead to ambiguity. Instead, we will use the transposition operator (see Subsection 1.6) and write \( c = (c_i)^T \) or \( c^T = (c_i) \).
It is often convenient to represent a matrix as a row vector of column vectors or as a column vector of row vectors. We can write

\[
A = \begin{pmatrix}
  a_{1*} \\
  \vdots \\
  a_{m*}
\end{pmatrix}
\quad \text{or} \quad
A = (a_{*1} \ a_{*2} \ \cdots \ a_{*j} \ \cdots \ a_{*n}),
\]

where \(a_{i*}\) and \(a_{*j}\) denote respectively the \(i\)th row and the \(j\)th column of \(A\). This notation convention for rows and columns is not as common as the ones we have seen for matrices, column vectors, and row vectors, but this is the one I shall use in these short notes.

### 1.2 Matrix product

I won’t go over matrix addition and multiplication by a scalar. I probably don’t really need to go over matrix multiplication either, but I will do it just to insist on a couple of points that will play a crucial role in some of the algorithms we will see today.

If \(A\) is an \(m \times n\) matrix and \(B\) is an \(n \times p\) matrix, then \(C = AB\) is an \(m \times p\) matrix and the element at row \(i\) and column \(j\) in \(C\),

\[
c_{i,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}.
\]

In other words\(^1\), \(c_{i,j}\) is obtained by multiplying row \(i\) of \(A\) and column \(j\) of \(B\). Figure 1 shows how a row of \(A\) is “applied” to a column of \(B\) to produce an element of \(C\).

---

1For those who are familiar with the “suffix notation” mathematicians are so fond of, the above equation can be written as \(c_{ij} = a_{ik}b_{kj}\).
Note that the only thing we need for this product to be defined is that the number of columns of \( A \) be equal to the number of rows of \( B \). The number of rows of \( A \) gives us the number of rows of the product. The number of columns of \( B \) defines the number of columns of the product.

Now here are the important points I alluded to earlier:

- If we swap two rows of Matrix \( A \), then we must swap the corresponding two rows of Matrix \( C \), while keeping \( B \) unchanged, if we want to preserve the equality;
- If we swap two columns of Matrix \( A \), then we must swap the corresponding two rows of Matrix \( B \) to preserve the result (\( C \) is left unchanged by this operation);
- If we swap two rows of Matrix \( B \) then we must swap the corresponding two columns of Matrix \( A \) to preserve the result (this is just the previous item rewritten in a different order);
- The above points remain valid in the particular case where \( B \) and \( C \) are column vectors.

**A word on algorithmic complexity of matrix product**

If we are computing the product of an \( m \times n \) and an \( n \times p \) matrix, we know that the product will be an \( m \times p \) matrix, and therefore we will need to compute \( mp \) elements according to the expression shown in (2). This expression requires the computation of \( n \) products and \( n-1 \) additions. If the computation time for a multiplication is \( \alpha \) and that for an addition is \( \beta \), then the computation time for a single element of the product matrix is \( \alpha n + \beta(n-1) \approx (\alpha + \beta)n \). So we see that computing the product of our two matrices will take a time

\[
T(m, n, p) \approx mp(\alpha + \beta)n = (\alpha + \beta)mn = O(mnp).
\]

From now on I will not differentiate anymore between the computation time of an addition, product or division. I will simply count these as “one flop” (for floating point operation).

### 1.3 A re-interpretation of matrix-vector product

We can gain some insight on what is going on when we premultiply a column vector by a matrix (or postmultiply a row vector by a matrix) by using the row and column notations we introduced in (1). Let us consider an \( m \times n \) matrix \( A \) and an \( n \times 1 \) column vector \( v \).

\[
A = (a_{e1} \ a_{e2} \ \cdots \ a_{ej} \ \cdots \ a_{en}) \quad \text{and} \quad v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_j \\ \vdots \\ v_n \end{pmatrix}.
\]

since the number of columns of \( A \) matches the number of rows of \( v \), we can compute the product \( A \ v \):

\[
A \ v = (a_{e1}v_1 + a_{e2}v_2 + a_{ej}v_j + \cdots + a_{en}v_n).
\]

We see that the column vector that we obtain is a linear combination (a weighted average) of the columns of \( A \). Keep that in mind; we will get back to this when we talk about systems of linear equations.
1.4 Block matrices

We can expand on the row and column notations we saw in (1) to introduce the notion of block matrices. A block matrix is simply a matrix whose elements are themselves matrices. For example

\[
A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\
A_{21} & A_{22} \end{pmatrix},
\]

where

\[
A_{11} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad A_{12} = \begin{pmatrix} a_{14} & a_{15} \\
a_{24} & a_{25} \\
a_{34} & a_{35} \end{pmatrix}, \quad A_{21} = \begin{pmatrix} a_{41} & a_{42} & a_{43} \end{pmatrix}, \\
A_{22} = \begin{pmatrix} a_{44} & a_{45} \end{pmatrix}.
\]

What do we gain by introducing block matrices? What is the point of all this? First, anything that makes the notation more compact represents a gain in my book. The most useful aspect of this notation, however, is that when we multiply together block matrices whose blocks have sizes that match², then the product of the block matrices can be computed in terms of their submatrices. To illustrate this, let us consider the following block matrix:

\[
B = \begin{pmatrix} B_{11} & B_{12} & B_{13} \\
B_{21} & B_{22} & B_{23} \end{pmatrix},
\]

where \(B_{11}\) is \(3 \times 2\), \(B_{12}\) is \(4 \times 3\), \(B_{13}\) is \(3 \times 1\), \(B_{21}\) is \(2 \times 2\), \(B_{22}\) is \(2 \times 4\), and \(B_{23}\) is \(2 \times 1\).

First, we can notice that the number of columns of \(A\) and the number of rows of \(B\) match, both in the block format (2 rows for \(A\), 2 columns for \(B\)) and in the “expanded” format (3 + 2 columns for \(A\) vs. 3 + 2 rows for \(B\)). Furthermore we can notice that the dimensions of the submatrices of \(A\) and \(B\) match as well: the number of columns of \(A_{11}\) is equal to the number of rows of \(B_{11}\), \(B_{12}\), and \(B_{13}\).

We can now compute the product of the two matrices.

\[
AB = \begin{pmatrix} A_{11} & A_{12} \\
A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\
B_{21} & B_{22} & B_{23} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} & A_{11}B_{13} + A_{12}B_{23} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} & A_{21}B_{13} + A_{22}B_{23} \end{pmatrix}.
\]

²Of course, the way the blocks are “carved” in the matrix is not arbitrary: the blocks should reflect the structure of the underlying computational problem.
Now let’s imagine that we know that the underlying computational problem dictates that $A_{11} = I_3$ (where $I_3$ is the $3 \times 3$ identity matrix), $A_{21} = 0$, $B_{13} = 0$, and $B_{22} = 0$. Then the product simplifies to

$$A \cdot B = \begin{pmatrix} I_3 & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & 0 \\ B_{21} & 0 & B_{23} \end{pmatrix} = \begin{pmatrix} B_{11} + A_{12} B_{21} & B_{12} & A_{12} B_{23} \\ A_{22} B_{21} & 0 & A_{22} B_{23} \end{pmatrix}$$

which requires fewer computations that the general form of the product.

### 1.5 Inverse of a matrix

Let $A$ be an $m \times m$ matrix. If there exists an $m \times m$ matrix $B$ such that

$$A \cdot B = B \cdot A = I_m,$$

where $I_m$ is the $m \times m$ identity matrix, then we say that $A$ is nonsingular. $B$ is unique, is called the inverse of $A$, and is denoted by $A^{-1}$. If there is no matrix $B$ verifying the above property, then $A$ is said to be singular.

**Theorem 1.** For any $m \times m$ matrices $A$ and $B$, and any $\lambda \in \mathbb{R}^*$,

- $A \cdot B$ is nonsingular and $(A \cdot B)^{-1} = B^{-1} A^{-1},$

- $\lambda \cdot A$ is nonsingular and $(\lambda \cdot A)^{-1} = \frac{1}{\lambda} A^{-1},$

- $A + B$ is not necessarily nonsingular.

### 1.6 Transposition

The transpose of an $m \times n$ matrix $A$ is an $n \times m$ matrix denoted by $A^T$ obtained by writing the rows of $A$ as columns (and the columns of $A$ as rows).

Examples:

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 8 & 6 \\ 5 & 2 & 1 \\ 3 & 2 & 7 \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} 1 & 2 & 5 & 3 \\ 4 & 8 & 2 & 2 \\ 7 & 6 & 1 & 7 \end{pmatrix},$$

$$x = \begin{pmatrix} 6 \\ 8 \\ 5 \\ 1 \end{pmatrix} \Rightarrow x^T = (6, 8, 5, 1).$$

A matrix that is equal to its transpose is said to be symmetric. It goes without saying that only a square matrix can be symmetric.
Theorem 2. For any \( m \times n \) matrices \( A \) and \( B \) and any \( \lambda \in \mathbb{R} \),

- \( (A + B)^T = A^T + B^T \),
- \( (AB)^T = B^T A^T \),
- \( (\lambda A)^T = \lambda A^T \).

The dot product between two vectors can be expressed as the product of a row vector and a column vector. Let us consider an \( n \)-dimensional vector space \( \mathbf{E} \) on which a basis \( \mathbf{B} \) has been defined, and let \( \mathbf{u} \) and \( \mathbf{v} \) be two vectors of \( \mathbf{E} \) whose coordinate vectors on \( \mathbf{B} \) are \( \mathbf{u} \) and \( \mathbf{v} \) respectively:

\[
(\tilde{u})_B = \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, \quad \text{and} \quad (\tilde{v})_B = \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}.
\]

Then the dot product of \( \mathbf{u} \) and \( \mathbf{v} \) (in \( \mathbf{B} \)) is a scalar (a number):

\[
\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i = \begin{pmatrix} u_1, u_2, \ldots, u_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \mathbf{u}^T \mathbf{v}.
\]

Note that the product of an \( m \times 1 \) column vector and a \( 1 \times n \) row vector is not a scalar but an \( m \times n \) matrix (the two vectors need not have the same number of elements):

\[
\mathbf{u} \mathbf{v}^T = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} \begin{pmatrix} v_1 & v_2 & \cdots & v_n \end{pmatrix} = \begin{pmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{pmatrix}.
\]

### 1.7 Systems of linear equations

A linear equation in the variables \( x_1, x_2, \ldots, x_n \) is an equation of the form

\[
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b,
\]

where \( a_1, a_2, \ldots, a_n \), and \( b \) are all real numbers. If the right-side term \( b = 0 \) the linear equation is said to be homogeneous.

---

3Actually, they could be complex numbers as well, and everything we will discuss henceforth applies to complex linear equations and systems of linear equations. In practice, however, separate implementations are required for complex numbers.
If we have \( m \) such linear equations in the same variables \( x_1, x_2, \ldots, x_n \), they define a system of linear equations, or SLE. We use a second index to identify the coefficients of the equation:

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\
    a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \\
\end{align*}
\]  

(4)

We can rewrite the SLE in matrix form as \( Ax = b \) if we pose

\[
A = \begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn}
\end{pmatrix}, \quad x = \begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}, \quad \text{and} \quad b = \begin{pmatrix}
    b_1 \\
    b_2 \\
    \vdots \\
    b_m
\end{pmatrix}.
\]

(5)

A is called the matrix of the SLE, and \( b \) is called the right-side term.

When \( m < n \) (fewer equations than unknowns), the SLE is said to be under-determined. When \( m > n \) (more equations than unknowns), the problem is said to be over-determined. We will treat these two cases next week when we talk about Linear Least Squares (LLS). For the remainder of this document, we will only consider the case where \( m = n \), that is, \( A \) is a square matrix.

Some authors use the extended matrix notation convention to represent the SLE\(^4\) as follows:

\[
\begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\
    a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn} & b_m
\end{pmatrix}.
\]

(6)

if \( B = 0 \) (all the linear equations are homogeneous), we say that the SLE is homogeneous. If \( x \) is a solution to an homogeneous SLE, then for any \( \lambda \in \mathbb{R} \), \( \lambda x \) is a solution of the SLE as well. Therefore, if an homogenous SLE has a nonzero solutions, it has an infinity of solutions.

### 1.8 Mathematical solution of an SLE

I don’t want to go into proofs, even simple ones, so I will simply state the basic theorem.

**Theorem 3.**

A non-homogenous SLE \( Ax = b \) admits a solution if and only if \( A \) is nonsingular. That solution is \( x = A^{-1}b \) (it is therefore unique).

In theory, this is very simple: all we have to do is invert the matrix of our SLE. In practice, however, inverting a matrix is

- algorithmically complex,
- a possible source of important numerical error when not done right.

In the remainder of this document we are going to view different algorithms for computing the solution without first computing \( A^{-1} \).

---

\(^4\)I am sure that it is very convenient. I am just not used to it yet, and therefore don’t use it much.
1.9 Determinants

1.9.1 When is a matrix nonsingular?

If we know that the SLE’s matrix is singular, then it is not worth the effort to try to attempt finding a solution. The following lemma is easily verified.

**Lemma 1.**

An $m \times m$ matrix $A$ is singular if and only if there exists an $m \times 1$ column vector $x \neq 0$ such that $Ax = 0$.

Now let us interpret this in light of the observation we made in Subsection 1.3, that multiplying a matrix and column vector amounts to computing a weighted average of the columns of the matrix. What this means is that a matrix is singular if and only if one of its rows can be expressed as a linear combination of the others.

For a $1 \times 1$ matrix, it is trivially sufficient to verify that the matrix’ single element is not equal to zero. For a $2 \times 2$ matrix

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix},$$

let us assume that $a_{21} \neq 0$. If $a_{22} = 0$, then the two columns are colinear if and only if $a_{12} = 0$ as well. Otherwise, we must have $a_{11}/a_{21} = a_{12}/a_{22}$, which gives us the familiar expression

$$a_{11}a_{22} - a_{21}a_{12} = 0.$$ 

We verify easily that this equation also covers the case when $a_{21} = 0$.

For a $3 \times 3$ matrix, it is a bit trickier. We need to express the fact that one column can be expressed as a linear combination of the other two, say $a_{\bullet 1} = \alpha a_{\bullet 2} + \beta a_{\bullet 3}$. After playing around with the equations, we obtain the following condition:

$$a_{11}a_{22}a_{33} + a_{13}a_{21}a_{32} + a_{12}a_{23}a_{31} - a_{11}a_{23}a_{32} - a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33}.$$

We notice that this expression is the sum of terms, each of which is the signed product of three elements of the matrix. We also notice that each of the terms contains one element from each column and one from each row of the matrix exactly. This leads us to a general expression in the case of an $n \times n$ matrix.

1.9.2 Iterative definition

**Definition 1 (permutation).**

Given a sequence of $n$ elements, we call permutation of that sequence any reordering of its elements. It can be shown that any permutation $\sigma$ is the combination of a series of swappings of two elements in the sequence. In addition, the combinations of swappings that produce a given permutation are either all odd or all even. We define $\varepsilon(\sigma)$, the signature of a permutation $\sigma$ as

$$\varepsilon(\sigma) = \begin{cases} -1, & \text{if } \sigma \text{ is an odd permutation,} \\ 1, & \text{if } \sigma \text{ is an even permutation.} \end{cases}$$
Definition 2 (determinant).
Let us consider the integers between 1 and $n$ and let us call $P_n$ the set of permutations of these numbers. Then we can define the determinant of an $n \times n$ matrix $A = [a_{ij}]_{i,j=1}^{n}$ as

$$\det(A) = \sum_{\sigma \in P_n} \varepsilon(\sigma) \prod_{i=1}^{n} a_{i \sigma(i)}$$

(7)

$$= \sum_{\sigma \in P_n} \varepsilon(\sigma) \prod_{j=1}^{n} a_{\sigma(j)j}.$$  

(8)

1.9.3 Recursive computation

Definition 3 (cofactor).
Given an $n \times n$ matrix $A = [a_{ij}]_{i,j=1}^{n}$ (the first index corresponds to rows, and the second to columns), we note $A_{kl}$ the submatrix of $A$ that is obtained by eliminating from $A$ its $k$th row and its $l$th column. We then call cofactor of $a_{kl}$ the determinant

$$D_{kl} = (-1)^{k+l} \det(A_{kl}).$$

Then you know that the determinant of $A$ can be computed as follows (note that this is really a property, not a definition):

$$\det(A) = \sum_{i=1}^{n} a_{1,i} D_{1i},$$

(9)

$$= \sum_{j=1}^{n} a_{j,1} D_{j1}.$$  

(10)

The first equation corresponds to a development along the first column, while the second equation corresponds to a development along the first row. The main quality of this technique is that it is very straightforward, but in practice you should almost never use it: It is simply too inefficient, as we shall see next.

1.9.4 Algorithmic complexity of the recursive algorithm

Let $d(n)$ be the number of flops required to compute the determinant of an $n \times n$ matrix by applying the recursive algorithm. The number of elementary operations is the following:

- $n$ computations of the determinant of an $(n - 1) \times (n - 1)$ matrix, each requiring $d(n - 1)$ flops,
- $n$ multiplications,
- $n - 1$ additions.
The total is \( d(n) = n \, d(n-1) + 2n - 1 \), with \( d(2) = 3 \) (2 multiplications and one addition).

We can see that \( d(n) = O(n!) \), which is bad news since \( n! \) grows even faster than \( 2^n \), which is already considered to be pretty bad.

**Theorem 4 (Stirling’s formula).** *In the neighborhood of \( +\infty \),*

\[
n! \approx \sqrt{2\pi n} \, n^n \, e^{-n}.
\]

### 1.9.5 Another interpretation of the determinant

You may remember from computer graphics that the matrix of a transformation from a basis \( B \) to a basis \( B' \) is obtained by writing for its columns the coordinates of the vectors of \( B' \) relative to \( B \). You may also remember that the determinant of this transformation matrix is the “volume” of the unit hypercube defined by the vectors of \( B' \). For example, in 2D, this “volume” is simply the area of the parallelogram defined by the two vectors.

If the determinant of the matrix is zero, this means that the volume of the hypercube is zero, which you can interpret as meaning that one direction (or more) of our vector space has been “collapsed”. We will return to this observation when we talk of eigensystems.

### 1.9.6 Properties

**Theorem 5 (properties of the determinant).**

*For any \( n \times n \) matrices \( A \) and \( B \), and any \( \lambda \in \mathbb{R} \),*

- \( \det(AB) = \det A \, \det B \),
- \( \det(\lambda A) = \lambda^n \, \det A \),
- \( \det(A^T) = \det A \),
- *If \( A \) is non singular then \( \det(A^{-1}) = \frac{1}{\det A} \),
- *Generally, \( \det(A + B) \neq \det A + \det B \).*

**Theorem 6 (determinant of a block triangular matrix).**

*Let \( A \) be a block upper-triangular matrix of the form*

\[
A = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1m} \\
0 & A_{22} & \cdots & A_{2m} \\
& \cdots & \cdots & \cdots \\
0 & 0 & \cdots & A_{mm}
\end{pmatrix},
\]

*where the \( A_{ij} \) are square matrices that don’t necessarily have all the same dimensions, and such that \( A_{ij} = 0 \) if \( i > j \). Then*

\[
\det A = \prod_{i=1}^{i=m} \det A_{ii}.
\]
1.10 Solution of an SLE by substitution

One of the classical techniques frequently taught in mathematics courses is the substitution algorithm. Let us consider the following SLE:

\[
\begin{align*}
3x_1 + 2x_2 + x_3 &= -10, \\
3x_1 - 3x_2 + x_3 &= 15, \\
6x_1 - 2x_2 - 2x_3 &= -2.
\end{align*}
\]

To solve this system, we proceed by eliminating \(x_3\) from the SLE. To do this, we use the third equation to get

\[x_3 = 3x_1 - x_2 + 1.\]

We use this equation as a substitution rule in the first and second linear equations, which gives us the simplified SLE

\[
\begin{align*}
6x_1 + 1x_2 &= 1, \\
6x_1 - 4x_2 &= 2,
\end{align*}
\]

From the first equation we get

\[x_2 = -6x_1 - 11,
\]

which, once substituted in the second equation gives us \(x_1\) as the solution to the equation

\[30x_1 = -30,
\]

that is, \(x_1 = -1, x_2 = -5, x_3 = 3.\) Although this method is simple enough for solving small SLEs by hand, it suffers from one major problem: errors accumulate pretty fast as the result of one computation step is used as input to the next one.

1.11 Crammer’s rule

Another technique that works well in mathematics, but less so in the real world. Given the SLE defined in Eq. (4) and Eq. (5) with \(m = n\), we can define \(D_i\) to be the \(m \times m\) matrix equal to \(A\) except for the \(i\)th column that has been replaced by the right-side term \(b\):

\[
D_i = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1i-1} & b_1 & a_{1i+1} & \cdots & a_{1m} \\
a_{21} & a_{22} & \cdots & a_{2i-1} & b_2 & a_{2i+1} & \cdots & a_{2m} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
a_{m1} & a_{m2} & \cdots & a_{mi-1} & b_m & a_{mi+1} & \cdots & a_{mm}
\end{pmatrix},
\]

\[
= \begin{pmatrix}
A_{i1} & A_{i2} & \cdots & A_{i,i-1} & b & A_{i,i+1} & \cdots & A_{im}
\end{pmatrix}.
\]

**Theorem 7 (Crammer’s rule).**
The variables of the SLE can be computed by application of the ratio

\[x_i = \frac{\det D_i}{\det A}.
\]

Needless to say, this is not a method that has much practical interest, would it be only because of its prohibitive computational cost. In addition, the calculation of a determinant has the potential to require the addition of terms with different orders of magnitude, which we know to be a source of serious problems.
1.12 Norm of a vector, norm of a matrix

1.12.1 Introduction

The norm is the mathematical entity defined to capture the notion of length of a vector. You should be familiar with the Euclidean norm (the square root of the sum of squares of all coordinates), but it is not the only norm that exists, and it is not always the best.

Definition 4 (norm over a vector space $\mathbf{E}$). A norm is any function $\Phi : \mathbf{E} \rightarrow \mathbb{R}^+$ that has the following properties:

1. $\forall \vec{u} \in \mathbf{E}, \Phi(\vec{u}) = 0$ if $\vec{u} = \vec{0}$,
2. $\forall (\vec{u}, \lambda) \in \mathbf{E} \times \mathbb{R}, \Phi(\lambda \vec{u}) = |\lambda| \Phi(\vec{u})$,
3. $\forall (\vec{u}, \vec{v}) \in \mathbf{E} \times \mathbf{E}, \Phi(\vec{u} + \vec{v}) \leq \Phi(\vec{u}) + \Phi(\vec{v})$.

Note that the above definition of a norm does not refer to the coordinates of the vector. In practice, however, a given norm will be defined in terms of the coordinates relative to a specific basis. When performing numerical calculations, we assume that our calculations are performed with coordinates expressed in the same given basis, so that the coordinates are the vector. When I write $\Phi(\vec{u})$, where $\vec{u} = (u_1, \ldots, u_n)^T$ is a column vector, I refer to the norm $\Phi(\vec{u})$ of the vector $\vec{u}$ whose coordinate vector (in the implicitly defined basis $\mathcal{B}$) $\vec{u}$ is.

1.12.2 Classical vector norms

We can define the following norms for a column vector $\vec{u} = (u_1, \ldots, u_n)^T$:

$$N_1(\vec{u}) = \|\vec{u}\|_1 = \sum_{i=1}^{n} |u_i|,$$

$$N_2(\vec{u}) = \|\vec{u}\|_2 = \left(\sum_{i=1}^{n} (u_i)^2\right)^{1/2},$$

$$N_{\infty}(\vec{u}) = \|\vec{u}\|_{\infty} = \max_{i=1,\ldots, n} |u_i|.$$

For example, if we take $\vec{v} = (-3, 2.5)$, then we will get:

$$N_1(\vec{v}) = 3 + 2.5 = 5.5,$$
$$N_2(\vec{v}) = \sqrt{9 + 6.25} = \sqrt{15.25} = 3.9051 \cdots,$$
$$N_{\infty}(\vec{v}) = 3.$$

The $N_2$ norm is the traditional Euclidean norm and gives the “bird-fly” length of the vector. The $N_1$ norm is associated with the “Manhattan” distance. It is easy to see why: it measures the length of travel when one can only move along specified axes (no moving through a building). The $N_{\infty}$


norm is rarely used in geometry because it does not have any easy intuitive interpretation, but it is fairly common in vector spaces such as the space of $m \times n$ matrices.

There are immediate consequences to the choice of a norm. For example, let us define the *circle of center* $\mathcal{O}$ and *radius* $R$ as the following set:

$$ C(\mathcal{O}, R) = \{ \mathcal{M} \in \mathcal{E} : \| \mathcal{O} - \mathcal{M} \| = R \}. $$

Then, depending on the norm that we choose, circles will have very different appearances. Figure 2 shows, for the basis $\mathcal{B} = \{ \vec{i}, \vec{j} \}$ drawn on the left, what the circle $C(\mathcal{O}, R)$ would look like if the norm is $N_1$, $N_2$, or $N_\infty$.

$$ \Phi = N_1 \quad \Phi = N_2 \quad \Phi = N_\infty $$

Figure 2: Appearance of the circle of center $\mathcal{O}$ and radius $R$ depending on the choice of a norm.

### 1.12.3 Matrix norms

If we have a matrix $\mathbf{A} = (a_{ij})_{i=1 \ldots m, j=1 \ldots n}$, then we define the norm of $\mathbf{A}$, $\| \mathbf{A} \|$, as

$$ \| \mathbf{A} \| = \max_{\mathbf{v} \neq \mathbf{0}} \frac{\| \mathbf{A} \mathbf{v} \|}{\| \mathbf{v} \|}. $$

Clearly, the norm in the space of matrices depends on the type of norm we choose for our vectors. It turns out that if we choose the $N_1$ or $N_\infty$ for vectors, then we get a simple expression for the norm of $\mathbf{A}$:

$$ N_1(\mathbf{A}) = \| \mathbf{A} \|_1 = \max_{j=1 \ldots n} \sum_{i=1}^{m} |a_{ij}|, $$

$$ N_\infty(\mathbf{A}) = \| \mathbf{A} \|_\infty = \max_{i=1 \ldots m} \sum_{j=1}^{n} |a_{ij}|. $$

The $N_2$ norm of a matrix is more complicated to determine. On the other hand, it has some interesting properties, to which we will return when we talk about eigensystems. Other norms, easy to compute but not necessarily easy to manipulate, can be defined. One such norm is the Frobenius norm:

$$ \| \mathbf{A} \|_F = \left( \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}, $$
Note that for column vectors the Frobenius norm $\| \|_f$ coincides with the Euclidean norm $\| \|_2$. This is, however, not the case with other matrices.

**Example:** Let us consider $A = \begin{pmatrix} 1 & -3 & 2 \\ 4 & -1 & 6 \\ -5 & 4 & 0 \end{pmatrix}$

Then the $N_1$ and $N_\infty$ norms of matrix $A$ are

$$\|A\|_1 = \max (1 + 4 + 5, 3 + 1 + 5, 2 + 6 + 0) = 10,$$

$$\|A\|_\infty = \max (1 + 3 + 2, 4 + 1 + 6, 5 + 4 + 0) = 11.$$

Note that for the matrix and vector norms we have defined, the following properties (which are simply restatements of the definition of a norm) hold for any compatible matrices $A$ and $B$, and vector $x$:

- $\|A + B\| \leq \|A\| + \|B\|$,  
- $\|AB\| \leq \|A\| \|B\|$,  
- $\|Ax\| \leq \|A\| \|x\|$,  

provided that the same norm is used for all terms in the equation.

### 1.13 Condition number of a matrix

#### 1.13.1 Definition

The *condition number* of a square nonsingular matrix $A$ is defined by

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|. \quad (11)$$

By convention $\text{cond}(A) = \infty$ if $A$ is singular. The condition number measures the ratio of the maximum amount of stretching to the maximum amount of shrinking that the matrix can apply to nonzero vectors. The condition number cannot be smaller than 1. A matrix that has a very large condition number is nearly singular, and is said to be *ill-conditioned*.

#### 1.13.2 Error bounds

We have already seen that except that except in “made for exam” toy problems, we can always expect *some* amount of noise to be present in our data. It seems therefore natural to ask what the effect of that noise on the solution we compute can be expected to be. Let us study first the effect
of noise in the right-side term. Let $x$ be a solution to the SLE $Ax = b$ and let $\hat{x}$ be a solution to the SLE $A\hat{x} = b + \Delta b$. We pose $\Delta x = \hat{x} - x$. Then

$$\|b\| = \|Ax\| \leq \|A\| \|x\| \Rightarrow \frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}$$

$$\|\Delta x\| = \|A^{-1}\Delta b\| \leq \|A^{-1}\| \|\Delta b\|.$$  

From the two above equations and the definition of the condition number (11), we obtain the following result:

$$\frac{\|\Delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\Delta b\|}{\|b\|}.$$  

A similar result can be derived for an upper bound on the effect of error in the SLE’s matrix. Let $x$ be a solution to the SLE $Ax = b$ and let $\hat{x}$ be a solution to the SLE $(A + E)\hat{x} = b$, where $E$ is a “small” error term. Again, we pose $\Delta x = \hat{x} - x$. Then

$$\frac{\|\Delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|E\|}{\|A\|}.$$  

If the input data is accurate to machine precision, then

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \text{cond}(A) \epsilon_{\text{mach}}.$$  

## 2 LU Factorization

### 2.1 Gaussian elimination

#### 2.1.1 An example

This is probably the algorithms that you have used the most in your linear algebra course in college. Let us consider the following SLE:

$$
\begin{pmatrix}
3 & 4 & -1 & 2 \\
3 & 4 & 3 & 3 \\
-3 & -6 & 4 & -1 \\
-6 & -2 & -3 & -5
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
10 \\
0 \\
11
\end{pmatrix}.
$$

To solve this system, we are going to transform it to “echelon form.” The idea is to eliminate the terms below the diagonal, starting from the first column, and then moving to the right. We have already observe that we can swap rows in the SLE or add to a row a linear combination of other rows without changing the solution of the SLE. We exploit this observation by subtracting the first row from the second row. This give us the modified SLE

$$
\begin{pmatrix}
3 & 4 & -1 & 2 \\
0 & 0 & 4 & 1 \\
-3 & -6 & 4 & -1 \\
-6 & -2 & -3 & -5
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
11 \\
0 \\
11
\end{pmatrix}.
$$
Note that we applied the transformation to the right-side term as well. We do the same thing to the third row (add the first row) and the fourth row (add two times the first row) to get the new SLE in which all the subdiagonal terms in the first column are now equal to zero:

\[
\begin{pmatrix}
3 & 4 & -1 & 2 \\
0 & 0 & 4 & 1 \\
0 & -2 & 3 & 1 \\
0 & 6 & -5 & -1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
11 \\
-1 \\
9
\end{pmatrix}.
\]

This new SLE poses a problem: We can’t use the second row to eliminate subdiagonal elements in the second column (and we can’t use the first row anymore since that would reintroduce nonzero elements in the first column). To go around this problem, we swap the second and third rows of our revised SLE, as well as the corresponding right-side elements:

\[
\begin{pmatrix}
3 & 4 & -1 & 2 \\
0 & 0 & 4 & 1 \\
0 & -2 & 3 & 1 \\
0 & 6 & -5 & -1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
-1 \\
11 \\
9
\end{pmatrix}.
\]

We add 3 times the second row to the fourth row to complete our processing of the second column:

\[
\begin{pmatrix}
3 & 4 & -1 & 2 \\
0 & 0 & 4 & 1 \\
0 & -2 & 3 & 1 \\
0 & 6 & 4 & 2
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
-1 \\
11 \\
6
\end{pmatrix}.
\]

Finally, we subtract the third row from the fourth row to get the following upper-triangular SLE:

\[
\begin{pmatrix}
3 & 4 & -1 & 2 \\
0 & -2 & 3 & 1 \\
0 & 0 & 4 & 1 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{pmatrix}
= 
\begin{pmatrix}
-1 \\
-1 \\
11 \\
-5
\end{pmatrix}.
\]

The system is called upper-triangular because all the elements of its matrix that are below the diagonal are equal to zero. An upper-triangular SLE is solved by \textit{backward substitution}, or \textit{back-substitution}. The last row of the SLE gives us \(x_4 = -5\). We report this into the linear equation defined by the third row:

\[4x_3 + x_4 = 11 \Rightarrow x_3 = (11 + 5)/4 = 4.\]

Knowing \(x_3\) and \(x_4\) we can now compute \(x_2\), using the second row of the SLE:

\[-2x_2 + 3x_3 + x_4 = -1 \Rightarrow x_2 = (-1 - 12 + 5)/2 = 4.\]

Finally, reporting \(x_2\), \(x_3\), and \(x_4\) in the first equation, we get the value of the first unknown, \(x_1 = -1\).
2.1.2 The algorithm

The Gaussian elimination (and the closely related Gauss-Jordan elimination) algorithm that we use when solving an \( n \times n \) SLE \( A \mathbf{x} = \mathbf{b} \) “by hand” transforms the matrix of the SLE into an upper-triangular matrix, so that the new SLE can be easily solved by (backward) substitution. All the transformations that are applied to rows of the matrix (swapping, scaling, addition of another row) must also be applied to the corresponding elements of the right-side term if we want the modified SLE to remain mathematically equivalent to the original SLE.

We proceed with each column \( k \), from the first to the penultimate, and try to cancel all the terms in column \( k \) that lie below the diagonal. At step \( k \), our SLE looks as follows:

\[
\begin{pmatrix}
u_{1,1} & u_{1,2} & \cdots & u_{1,k-1} & u_{1,k} & \cdots & u_{1,n} \\
0 & u_{2,2} & \cdots & u_{2,k-1} & u_{2,k} & \cdots & u_{2,n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \ddots & u_{k-1,k-1} & u_{k-1,k} & \cdots & u_{k-1,n} \\
\vdots & \vdots & \ddots & 0 & u_{k,k} & \cdots & u_{k,n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & a_{n,k} & \cdots & a_{n,n}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{k-1} \\
x_k \\
\vdots \\
x_n
\end{pmatrix}
= \begin{pmatrix}
-d_1 \\
d_2 \\
\vdots \\
d_{k-1} \\
d_k \\
\vdots \\
b'_n
\end{pmatrix},
\]

that is,

\[ A^{(k-1)} \mathbf{x} = \mathbf{b}^{(k-1)}. \]

In this expression, the matrix terms labelled \( a_{ij} \) and the right hand terms labelled \( d_i \), with \( l < k \) are “final”: they corresponds to rows that have already been processed as the column index increased from 1 to \( k - 1 \). We use the notation \( A^{(k-1)} \) and \( \mathbf{b}^{(k-1)} \) to refer to the partially triangularized SLE before step \( k \). Naturally, we have \( A^{(0)} = A \) and \( \mathbf{b}^{(0)} = \mathbf{b} \). The diagonal element in row \( k \), \( u_{kk} \), is called the pivot for column \( k \). This element will be used for scaling all the rows below the diagonal.

If we set aside for the time being the issue of pivoting (i.e. of swapping rows or columns when the pivot \( u_{kk} \) is null), we saw in our example that if we want to “cancel all the terms in column \( k \) that lie below the diagonal,” we simply have to apply the following algorithm:

```csharp
for row index i from k + 1 to n do
    \( m_{ik} = a'_{ik} / u_{kk} \)
    // subtract \( m_{ik} \) * row_k from row_i;
    for j from k to n do
        \( a'_{ij} = a'_{ij} - m_{ik} * u_{kj} \)
        \( b'_i = b'_i - m_{ik} * d_k \)
```

Using this algorithm, we can solve the SLE by the following steps:

1. \( k = 1 \) and \( i = 2 \) [\( \# 0 \)]:
   - \( u_{11} \) is the pivot for column \( 1 \).
   - \( x_1 = b_1 / u_{11} \).

2. \( k = 2 \) and \( i = 3 \) [\( \# 1 \)]:
   - \( u_{22} \) is the pivot for column \( 2 \).
   - \( x_2 = (b_2 - u_{21} x_1) / u_{22} \).

3. \( k = 3 \) and \( i = 4 \) [\( \# 2 \)]:
   - \( u_{33} \) is the pivot for column \( 3 \).
   - \( x_3 = (b_3 - u_{31} x_1 - u_{32} x_2) / u_{33} \).

And so on.

\[ A \mathbf{x} = \mathbf{b} \] is the fully triangular form of the original SLE, with its solution \( \mathbf{x} \) obtained by backward substitution.
After this step, column $k$ is now “final”:

For $i = 1 \ldots k$, $u_{ik} = a'_{ik}$. For $i = k + 1 \ldots n$, $u_{ik} = a'_{ik} = 0$.

So is row $k + 1$:

For $j = 1 \ldots k$, $u_{k+1,j} = a'_{k+1,j} = 0$. For $j = k + 1 \ldots n$, $u_{k+1,j} = a'_{k+1,j}$.

And so is the $k + 1$-th element of the right-side term:

$$d_{k+1} = b'_{k+1}.$$ 

Of course, since the operation we just applied is linear, it can be expressed as a multiplication by a matrix on both sides of the equation:

$$A^{(k-1)} x = b^{(k-1)}$$

$$\Rightarrow M_k A^{(k-1)} x = M_k b^{(k-1)},$$

where

$$M_k = \begin{pmatrix}
1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & \cdots & -m_{k+1,k} & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \ddots & \vdots \\
0 & \cdots & -m_{n,k} & 0 & \cdots & 1
\end{pmatrix}. \quad (12)$$

At each step we pose

$$A^{(k)} = M_k A^{(k-1)} \text{ and } b^{(k)} = M_k b^{(k-1)},$$

so that the modified SLE is now

$$A^{(k)} x = b^{(k)}.$$ 

After we have run our elimination algorithm for columns 1 through $n - 1$ of matrix $A$, we end up with an upper-triangular SLE that can be solved easily by backsubstitution:

$$U x = d. \quad (13)$$

Expressed in terms of the $M_k$ matrices, we have:

$$A x = b,$$

$$M_1 A x = M_1 b,$$

$$M_2 M_1 A x = M_2 M_1 b,$$

$$\vdots \quad \vdots$$

$$M_{n-1} \ldots M_2 M_1 A x = M_{n-1} \ldots M_2 M_1 b.$$ 

If we pose $M = M_{n-1} \ldots M_2 M_1$ we can rewrite the above as

$$M A x = M b. \quad (14)$$
As you can see in Equation 12, the $M_k$ are all lower-triangular with a unit diagonal (1s along their diagonal), therefore $M$ will also be lower-triangular with a unit diagonal. If we compare Equation (14) with Equation (13), we see that we have $MA = U$ and $Mb = d$, which brings up the following comments:

1. The upper-triangular SLE in Equation (13) is very easy to solve, but it obliges us to process the right hand term $b$ while we “triangularize” $A$ through Gaussian elimination: It is not possible to “pre-process” $A$ separately from $b$.

2. Equation (14) opens up the way for some separate processing of $A$ (to compute $M$), but we would prefer to have a form of this equation without $M$ on the right side of the equation.

3. If we knew $M$ to be non-singular (invertible), then we could combine both equations and try to solve $M^{-1} U x = b$.

It turns out that $M$ is non-singular (you should be able to figure out why on your own) and that its inverse matrix is also lower-triangular with a unit diagonal. For example,

\[
\text{if } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 1 & 0 & 0 \\ 2 & -1 & 1 & 0 \\ 3 & 3 & 2 & 1 \end{pmatrix}, \quad \text{then } M^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ -11 & -4 & -2 & 1 \end{pmatrix}.
\]

What this boils down to is that if we pose $L = M^{-1}$, we have $A = LU$, where $L$ is a lower-triangular matrix with a unit diagonal and $U$ is an upper-triangular matrix. This “LU factorization” is independent from the right hand term $b$, and the original SLE can now be written as

\[
L U x = b.
\]

This SLE can be solved in two easy steps:

1. First we solve the SLE $Ly = b$, which is easily done by forward substitution (see the algorithm below).

2. Then we solve the SLE $Ux = y$, which is (again, easily) solved by backsubstitution (see the algorithm below).

2.1.3 Algorithm for backsubstitution

For a right-side term $b = (\hat{b}_i)_{i=1\ldots n}$:

\[
x_n = b_n / u_{nn}
\]

\[\text{for } i \text{ from } n-1 \text{ to } 1 \text{ do}
\]

\[
x_i = \frac{1}{u_{ii}} \left( \hat{b}_i - \sum_{j=i+1}^{n} u_{ij} x_j \right)
\]
2.1.4 Algorithm for forward substitution

For a right-side term \( d = (d_i)_{i=1}^{n} \):

\[
x_1 = d_1
\]

\textbf{for} \( i \) from 2 to \( n \) \textbf{do}

\[
x_i = d_i - \sum_{j=1}^{i-1} l_{ij}x_j
\]

2.2 Crout’s algorithm

Crout’s algorithm factorizes the SLE’s matrix into the product of a square lower-triangular matrix \( L \) and a square upper-triangular matrix \( U \). There exist an infinity of pairs of such matrices. For example, if \( L \) and \( U \) have the required shape and verify \( LU = A \), then for any \( \lambda \in \mathbb{R}^* \), \( \lambda L \) and \( (1/\lambda) U \) would provide another valid factorization of \( A \), as would \( (1/\lambda) L \) and \( \lambda U \). Crout imposes the following additional condition on \( L \):

\( L \) has a unit diagonal, i.e. it has only ones on its diagonal: \( l_{ii} = 1, i = 1 \ldots n \).

It is easy to verify that if \( A \) admits an \( LU \) factorization, then there exists such a factorization for which \( L \) verifies the above condition.

Computing \( M \) first and inverting it is not a very efficient way to factorize \( A \). Instead, let us write out all the product of the \( L \) and \( U \) matrices:

\[
\begin{pmatrix}
1 & 0 & \cdots & 0 \\
l_{2,1} & 1 & \cdots & 0 \\
\vdots & \ddots & \ddots & \vdots \\
l_{n,1} & \cdots & l_{n,n-1} & 1
\end{pmatrix}
\begin{pmatrix}
u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\
0 & u_{2,2} & \cdots & u_{2,n} \\
\vdots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & u_{n,n}
\end{pmatrix}
= \begin{pmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
\vdots & \ddots & \ddots & \vdots \\
a_{n,1} & a_{n,2} & \cdots & a_{n,n}
\end{pmatrix}
\]

If we develop this equation to get an expression for each of the terms of \( A \), \( a_{i,j} \) we must differentiate between three cases:

- if \( i < j \) : \( l_{i1}u_{1j} + l_{i2}u_{2j} + \cdots + u_{ij} = a_{i,j} \),
- if \( i = j \) : \( l_{i1}u_{1j} + l_{i2}u_{2j} + \cdots + u_{jj} = a_{i,j} \),
- if \( i > j \) : \( l_{i1}u_{1j} + l_{i2}u_{2j} + \cdots + l_{ij}u_{jj} = a_{i,j} \).

A careful look at these equations reveals something very interesting: It is possible to compute the \( l \) and \( u \) terms in one pass, working over the columns from 1 through \( n \).
// for all columns of matrix A
for j from 1 to n do
  // for all elements above or on the diagonal
  for i from 1 to j do
    u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}
  // for all elements below the diagonal
  for i from j + 1 to n do
    l_{ij} = \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)

Although when we begin the algorithm none of the terms \( u_{kj} \) or \( l_{ik} \) is known, they are all computed just in time to be known when they appear on the right side of an equation in the algorithm.

Furthermore, once a term \( a_{ij} \) has been used, it will not be used anymore. As a consequence it is possible to store the \( a \) and \( l \) terms in the “old” A matrix by overwriting \( a_{ij} \) with \( l_{ij} \) if \( i > j \) or \( u_{ij} \) if \( i \leq j \) to give the matrix

\[
\begin{pmatrix}
  u_{11} & u_{12} & \cdots & u_{1n} \\
  l_{21} & u_{22} & \cdots & u_{2n} \\
  \vdots & \ddots & \ddots & \vdots \\
  l_{n1} & \cdots & l_{n,n-1} & u_{nn}
\end{pmatrix}
\]

Of course, one drawback of overwriting \( A \) is that you lose your input data. If you do this, better make sure that you are working with a copy of your matrix, on the other hand, you will see in the next subsection that this makes the pivoting a lot easier. Even if you do not overwrite \( A \), there is no point returning \( L \) and \( U \) in two separate matrices. You only compute them to send them to your forward and backward substitution modules anyway.

### 2.3 Scaled partial pivoting

#### 2.3.1 The problem with small pivots

What has been left out of the above algorithm is pivoting. What do we do when the term \( u_{jj} \) is zero? We have already seen that we should swap rows. Unfortunately, this is not sufficient. As a general rule, when a procedure fails for some values of the data, it generally behaves unreliably near the failing values. Let us study a classical example of such an erratic behavior:

\[
\begin{align*}
  \varepsilon x_1 + x_2 &= 1, \\
  x_1 + x_2 &= 2,
\end{align*}
\]

in which \( \varepsilon \) is a small number \( \neq 0 \). If we follow the above algorithm, we get the following upper-triangular SLE:

\[
\begin{align*}
  \varepsilon x_1 + x_2 &= 1, \\
  \left(1 - \frac{1}{\varepsilon}\right) x_2 &= \left(2 - \frac{1}{\varepsilon}\right).
\end{align*}
\]
The backsubstitution algorithm gives us

\[
x_2 = \frac{2 - 1/\varepsilon}{1 - 1/\varepsilon},
\]
\[
x_1 = \frac{1 - x_2}{\varepsilon}.
\]

For \(\varepsilon\) small (say, about \(10^6\) if we work in single precision), then \(1/\varepsilon\) is large, much larger than 1 or 2. In fact, we will get \(2 - 1/\varepsilon = -1/\varepsilon\) and \(1 - 1/\varepsilon = -1/\varepsilon\).

Therefore the computer will compute \(x_2 = 1\) and \(x_1 = 0\), while the correct values would be \(x_1 \approx x_2 \approx 1\). In this case the relative error for \(x_1\) is about 100 percent!

### 2.3.2 Relation to the condition number

Considering the poor quality of the above result, one could expect the condition number of this SLE’s matrix \(A\) to be high. To verify this conjecture we must compute \(A\)‘s condition number, and for this we need to determine first the norm of \(A\) and that of \(A^{-1}\). I will use the \(\|\cdot\|_{\infty}\) norm for these calculations\(^5\).

\[
\|A\|_{\infty} = \max (|\varepsilon| + 1, 1 + 1) = 2.
\]

The inverse of \(A\) is \(A^{-1} = \begin{pmatrix} -1/(1 - \varepsilon) & 1/(1 - \varepsilon) \\ 1/(1 - \varepsilon) & -\varepsilon/(1 - \varepsilon) \end{pmatrix}\). From this we get

\[
\|A^{-1}\|_{\infty} = \max \left(\frac{1 + |\varepsilon|}{|1 - \varepsilon|}, \frac{2}{|1 - \varepsilon|}\right) = \frac{2}{|1 - \varepsilon|}.
\]

\(A\)’s condition number is therefore

\[
\text{cond } A = \frac{4}{|1 - \varepsilon|},
\]

which for small values of \(\varepsilon\) is not large at all! In fact, as \(\varepsilon\) gets closer to zero, the value of the condition number decreases, tending to a limit value of 4. Doesn’t this contradict the error bounds we were given in Subsection 1.13.2?

No, it does not. What the error bounds gives us is a sort of “minimax” result: With the best algorithm for computing the solution, we get an upper bound on the relative forward error for a given amount of relative error in the data. This does not, however, tell us anything about the effect of applying a bad algorithm. Using as our pivot a very small number, as we did in the this section, definitely qualifies as applying a bad algorithm

\(^5\)This is an arbitrary choice. I could have chosen any other norm. I would get qualitatively similar results, if not exactly identical values.
2.3.3 The solution

So what is the solution to this problem? The same as for a null pivot: We swap rows. If we swap the two rows of our SLE and apply the Gaussian elimination algorithm, we get

\[
\begin{align*}
&x_1 + x_2 = 2, \\
&\varepsilon x_1 + x_2 = 1,
\end{align*}
\]

\[\Rightarrow \begin{align*}
&x_1 + x_2 = 2, \\
&(1 - \varepsilon)x_2 = 1 - 2\varepsilon
\end{align*}\]

Backsubstitution then gives us the following (correct) result:

\[
\begin{align*}
&x_2 = \frac{1 - 2\varepsilon}{1 - \varepsilon} \approx 1, \\
&x_1 = 2 - x_2 \approx 1.
\end{align*}
\]

So the problem only occurs when we have a small pivot, and the solution is simply to swap the unsuitable row with a better one, right?

Wrong! The problem does not occur because the pivot is small, but because it is small relative to the other elements of the same row. We would have the same problem for the following SLE, which is mathematically equivalent to the one we have been studying in this example:

\[
\begin{align*}
&x_1 + \frac{1}{\varepsilon}x_2 = \frac{1}{\varepsilon}, \\
&x_1 + x_2 = 2.
\end{align*}
\]

Our identification of the problem was incorrect, but the solution was the right one: swapping rows will lead to a good triangular system (solution of that SLE left as an exercise to the reader).

2.3.4 To swap or not to swap: One of many questions

The above discussion brings up a few questions:

1. How small is small? When should we decide that a pivot is small enough to call for a row swap?

2. When we swap rows, how do we choose which row to swap with?

3. If swapping rows is a very fast operation (is it?), why not do it all the time, and always choose the “best” pivot?

4. At what point in the algorithm can we compare the pivots and swap rows?

We answer first the third question. The answer is: Indeed, why not? This is exactly what we will do. This makes Question 1 moot. Remain Questions 2 and 4. The former is somewhat tricky. We will address it in the next subsection.
When we look carefully in the algorithm without pivoting at the expression for $a_{ij}$ and that of the terms $l_{ij}$ we see that it is identical, except for the $u_{jj}$ scaling factor. In other words, we can do the following

1. For $i = j \ldots n$, compute $hu_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$,

2. Decide which of the above terms (say, the one at row $i^*$) would constitute a good pivot,

3. If $i^* \neq j$, swap rows $i^*$ and $j$ (this defines the new pivot $u_{jj}$),

4. For all rows $i = j + 1 \ldots n$ after the row swap, divide row elements by the term computed in step one to obtain $l_{ij}$.

In the above algorithm, I wrote $hu_{ij}$ to refer to yet-undifferentiated elements (not sure yet whether they belong to $L$ or $U$). This notation only makes sense because $L$ and $U$ are returned in the same matrix, as explained in Subsection 2.2.

### 2.3.5 Choice of a pivot: scaled partial pivoting

We will discuss implementation issues regarding Step 3 (row swapping) in Subsection 2.3.6, but for the moment, let us consider more carefully Step 2. Obviously, we need to swap rows when the candidate pivot computed at Step 1, $u_{jj}$ is null or very small relative to the rest of its row. To pick a new pivot, we have two possible choices: “first fit” or “best fit.” The first strategy consists in selecting the first element computed in Step 1 whose absolute value is larger than some predefined threshold. The second strategy consists in estimating a “quality” criterion for each pivot candidate and selecting the one with the highest score. The two most common quality estimates are

- **absolute value** $|hu_{ij}|$: We simply select the candidate pivot with the largest absolute value;
- **relative value** $|hu_{ij}| / \max_{k=1 \ldots n} |a_{i,k}|$: We select the candidate pivot that is the largest relative to the original row in matrix $A$.

The first strategy looks simple, but it will run into trouble for some matrices, as we have already seen in our example.

The second strategy works generally much better, so this is the one we will use in the following algorithm:
// for all rows of matrix A
for i from 1 to n do
    \( v_i = \max_{j=1 \ldots n} |a_{ij}| \)
// for all columns of matrix A
for j from 1 to n do
    // for all elements above the diagonal
    for i from 1 to \( j - 1 \) do
        \( u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \)
    // for all elements on or below the diagonal
    for i from j to n do
        \( lu_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \)
    determine \( i^* \) such that \( |lu_{i^*j}| = \max_{i=j \ldots n} |lu_{ij}| / v_i \)
    if \( i^* \neq j \) then
        swap rows \( j \) and \( i^* \) in A, L, U (set \( u_{jj} = lu_{i^*j} \))
        \( s = 1 / u_{jj} \) (if \( u_{jj} = 0 \), set \( s = 0 \))
    // for all elements below the (possibly new) diagonal
    for i from \( j + 1 \) to n do
        \( l_{ij} = s \times lu_{ij} \)

Note that when there is no choice but to use a zero pivot, \( u_{jj} = 0 \), then we set \( 1 / u_{jj} \) to zero! What is the meaning of this strange operation? Since when does \( 0 = \infty \)? The explanation is that from this point on, the LU factorization has failed because the matrix is singular. It is however possible to get a solution for the SLE that is not completely absurd if we set the problematic element equal to zero. Of course, our SLE solving module should report the problem and let the user verify the validity of the solution \( x \) returned by computing the difference \( Ax - b \). In fact it is never a bad idea to perform this verification, even when the LU factorization has not run into such problems.

### 2.3.6 How to swap rows

Regarding the swapping operation, we must be careful about one thing: we are (or could be) performing the factorization of \( A \) before we know the right hand term \( b \). Therefore, be cannot simply swap the rows. We have to keep track of the swaps we perform so that we can apply them to the right-side term when we try to solve the SLE.

The simplest way to do this is to maintain an array of row indexes that will store the permutations applied to the rows of our matrix. In other words, we replace in our algorithms any any reference to row index \( i \) by \( p_i \). Initially, \( p = \{1, 2, \ldots, n\} \). Each time the algorithm calls for a row swap, we simply invert the corresponding elements of \( p \). At the end of the factorization, \( p \) represents the permutation that has been applied to \( A \) and must now be applied to any right hand
term $b$ of an SLE we want to solve. So now, instead of manipulating element $LU_{i,j}$, we will work with $LU_{s_{i,j}}$, or if you prefer to write this in “array” notation, $LU[p[i]][j]$.

Now, where you have to be very careful is that the way you apply this row permutation depends on the way you decided to store your matrices. If you overwrite your $A$ matrix with the terms of the $L$ and $U$ matrices, then you should always use $p[i]$ instead of $i$ as the first index whenever you access the matrix. You should of course understand that this permutation vector should be sent back as a result of the factorization since you will need it in your substitution modules.

### 2.3.7 What is “partial” about partial pivoting

Or simply: What is full pivoting? Full pivoting does not only swap rows of the SLE but also columns. It is more complex because now instead of finding the maximum value in a 1D array (the scaling terms) we would have to find the maximum value in a 2D array, which poses additional storage problems when the matrix is large. Although on paper there would be gains to expect from a full pivoting strategy, the jury is still out on that one, and most software packages only implement partial pivoting.

### 2.4 Solving for more than one right-side term

If we must solve multiple SLEs with the same matrix

\[
\begin{align*}
A x^{(1)} &= b^{(1)}, \\
A x^{(2)} &= b^{(2)}, \\
&\vdots \\
A x^{(n)} &= b^{(n)},
\end{align*}
\]

where the superscript indicates the index of the SLE’s right-side term or solution, then we can define the $n \times m$ matrices $X$ and $B$ as follows:

\[
X = (x^{(1)}, x^{(2)}, \ldots, x^{(n)}), \\
B = (b^{(1)}, b^{(2)}, \ldots, b^{(n)}).
\]

We can now write our $m$ SLEs as a single matrix SLE:

\[
A X = B.
\]

Note that the number $m$ of columns is completely unrelated to the dimension of $A$: We can treat as many or as few SLEs as we want. The beauty of the $LU$ factorization method is that it still applies in this case because the factorization algorithm itself is still applied on matrix $A$, as in the case of a single right-side term, while the backsubstitution and forward substitution algorithm still “work” when the right-side term and the unknown are matrices (go back to Subsections 2.1.3 and 2.1.4 to convince yourself of that).
2.5 Inverting a matrix

Computing the inverse of an $n \times n$ matrix $A$ is the same as solving the SLE $AX = B$, where $I_n$ is the $n \times n$ identity matrix.

2.6 Algorithmic complexity

The LU decomposition takes about $\frac{1}{3}n^3$ flops. Backsubstitution and forward substitution both total $n^2$ executions of an inner loop containing one multiply and one add. Note that if we want to solve for $n$ right sides and the same matrix, we only need to perform the LU decomposition once, while the backsubstitution and forward substitution times will both be $\frac{1}{2}n^3$.

In the particular case where the right-side matrix is the identity matrix, the forward substitution is accelerated by a factor 3, for a computation time of about $\frac{1}{6}n^3$, while backsubstitution keeps the same computation time of $\frac{1}{2}n^3$.

2.7 Computation of the determinant

A “bonus” of the LU factorization is that we can now compute very easily the determinant of $A$. Indeed

$$\det A = \det \begin{pmatrix} L & U \end{pmatrix} = \det L \det U = \prod_{i=1}^{n} u_{ii},$$

since $L$ has a unit diagonal. So we can now compute the determinant of an $n \times n$ matrix in $O(n^3)$. Compare that with the $O(n!)$ of the recursive algorithm!

3 Improvements and Particular Cases

3.1 Iterative improvement of the solution

Having applied the LU algorithm (or any other alternative algorithm) to solve the SLE $A \mathbf{x} = \mathbf{b}$, we obtain a solution $\mathbf{x}$, which we hope to be close to the real solution $\mathbf{x}$. Let us call $\delta \mathbf{x}$ the absolute error of our solution: $\delta \mathbf{x} = \mathbf{x} - \mathbf{x}$. If we multiply our solution $\mathbf{x}$ by the SLE’s matrix, $A$, we get

$$A \mathbf{x} = A (\mathbf{x} + \delta \mathbf{x}) = A \mathbf{x} + A \delta \mathbf{x}.$$ 

Since $A \mathbf{x} = \mathbf{b}$, we see that

$$A \delta \mathbf{x} = A \mathbf{x} \mathbf{x} - \mathbf{b},$$

This defines a new SLE whose matrix is still $A$. The right-side term is known, and therefore we can solve the SLE (using the $LU$ factorization of $A$ previously computed, to get an estimate of
the error term $\delta x$. Having done this, we can get a better approximation of the correct solution by computing

$$x = \hat{x} - \delta x.$$  

We can therefore improve our solution by calculating an estimate of the error term and subtracting it from the solution $\hat{x}$. It is highly recommended to apply this extra step at least once because

- it improves solutions that were already good,
- it can sometimes salvage completely incorrect solution (by at least putting them back in the right range of values),
- its algorithmic complexity is $O(n^2)$, which for large values of $n$ is negligible compared to the $O(n^3)$ of the LU factorization + substitution.

It is possible to apply this step more than once, twice being generally considered to be sufficient. It is also possible to keep applying the iteration until the solution doesn’t change anymore.

### 3.2 Case of a positive definite matrix: Cholesky’s algorithm

A positive definite matrix is a symmetric $n \times n$ matrix that verifies the following condition:

For any nonzero $n \times 1$ column vector $v$,  

$$v^T A v > 0.$$  

Obviously, such a matrix cannot be singular.

It is fairly easy to prove that an positive definite matrix $A$ can be factorized under the following form:

$$A = L L^T.$$  

Note that in this equation we cannot impose on $L$ the constraint of having a unit diagonal, as we did with Crout’s algorithm. Other than that, it is fairly straightforward to adapt the LU factorization algorithm to handle this particular type of matrix. We get the following algorithm.

```c
// for all columns of matrix A
for j from 1 to n do
  // compute the pivot
  l_{jj} = \left( a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2}
  // fix the elements below the diagonal
  for i from j + 1 to n do
    l_{ij} = \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) / l_{jj}
```

Of course, we would have to add scaled partial pivoting to get a “complete” working algorithm. It is possible to avoid the computation of the square root if instead of decomposing $A$ as $L L^T$ we decomposed it as the product of three matrices, $A = L D L^T$, in which $D$ is a diagonal matrix and $L$ has a unit diagonal.
3.3 Band matrices and other special cases

Researchers have developed a number of optimized algorithms to handle matrices with special forms that are commonly encountered in scientific computing problems: tridiagonal matrices, pentadiagonal matrices, block-diagonal matrices, block-triangular matrices, and other sparse matrices. Most of these algorithms are adaptations of the Gaussian elimination to the particular geometry of the matrix.

Note that we can’t waste computation time to determine if the matrix has one of these classical special forms in order to apply a more optimal algorithm. We apply the best algorithm because we have properly studied and modeled our problem beforehand, and we know the shape of the matrix before we start writing a line of code.

4 Iterative Solution of an SLE

This is not at all the same thing as the iterative improvement. The latter assumed that a reasonably good solution had been computed and the idea behind these techniques is to compute the solution of an SLE as the limit of a converging sequence.

4.1 The Basic algorithm: Jacobi iteration

Let us start from our standard SLE

\[ Ax = b. \]  

(15)

Now let us decompose matrix \( A \) into the sum of three matrices \( L, D, \) and \( U \) such that \( L \) is strictly lower-diagonal (diagonal elements are all zero), \( D \) is diagonal, and \( U \) is strictly upper-diagonal. Now we can rewrite Eq. (15) as

\[ Dx = -(L + U)x + b, \]

or, if we multiply by \( D^{-1} \) on both sides of the equality sign,

\[ x = -D^{-1}(L + U)x + D^{-1} b. \]  

(16)

So what have we gained by changing Eq. (15) into Eq. (16)? Simply it offers up the possibility to try to compute \( x \) as the limit of the following recursive sequence:

\[ x^{(k+1)} = \left(-D^{-1}(L + U)\right)x^{(k)} + D^{-1} b. \]  

(17)

where the \( ^{(k)} \) superscripts indicate the order in the iteration. Why should it work? Does it always work? Even if it does work, what is so good about that? Let me answer the last question first. Since \( D \) is so easy to invert, we can compute \( D^{-1}(L + U) \) and \(-D^{-1}b\) once and for all, and all we have to do in our iteration is repeatedly multiply a matrix and a vector and add two vectors.
4.2 Get some insight from the scalar case

To understand why this should work, let’s study the following iteration

\[ x^{(k+1)} = a \cdot x^{(k)} + b, \]  

\( (18) \)

where \( a \) and \( b \) are real numbers and \( x^{(j)} \) represents the value of \( x \) (a real number or a complex number) after \( j \) steps of the iteration. At priori, it would seem that the result of the iteration will depend on three parameters: \( a, b, \) and the initial value of \( x \) (the starting point of the iteration). Let us see what happens for 10 steps of iteration and combinations of values of \( a, b, \) and \( x^{(0)} \). The table below shows some results.

Table 1: Tests of 10 iterations of Equation (18) for different sets of parameters.

<table>
<thead>
<tr>
<th>( a = 0.5 )</th>
<th>( a = 0.5 )</th>
<th>( a = 0.5 )</th>
<th>( a = -0.3 )</th>
<th>( a = -0.3 )</th>
<th>( a = -0.3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
</tr>
<tr>
<td>( x^{(0)} = 10 )</td>
<td>( x^{(0)} = -20 )</td>
<td>( x^{(0)} = 10 )</td>
<td>( x^{(0)} = 1 )</td>
<td>( x^{(0)} = -20 )</td>
<td></td>
</tr>
<tr>
<td>-2.</td>
<td>-4.25</td>
<td>-9.5</td>
<td>-1.2</td>
<td>-2.01</td>
<td>-3.9</td>
</tr>
<tr>
<td>-4.</td>
<td>-5.125</td>
<td>-7.75</td>
<td>-2.64</td>
<td>-2.397</td>
<td>-1.83</td>
</tr>
<tr>
<td>-5.</td>
<td>-5.5625</td>
<td>-6.875</td>
<td>-2.208</td>
<td>-2.2809</td>
<td>-2.451</td>
</tr>
<tr>
<td>-5.5</td>
<td>-5.78125</td>
<td>-6.4375</td>
<td>-2.3376</td>
<td>-2.31573</td>
<td>-2.2647</td>
</tr>
<tr>
<td>-5.75</td>
<td>-5.89063</td>
<td>-6.21875</td>
<td>-2.29872</td>
<td>-2.30528</td>
<td>-2.32059</td>
</tr>
<tr>
<td>-5.875</td>
<td>-5.94531</td>
<td>-6.10938</td>
<td>-2.31038</td>
<td>-2.30842</td>
<td>-2.30382</td>
</tr>
<tr>
<td>-5.9375</td>
<td>-5.97266</td>
<td>-6.05469</td>
<td>-2.30688</td>
<td>-2.30748</td>
<td>-2.30885</td>
</tr>
<tr>
<td>-5.96875</td>
<td>-5.98633</td>
<td>-6.02734</td>
<td>-2.30793</td>
<td>-2.30776</td>
<td>-2.30734</td>
</tr>
<tr>
<td>-5.98438</td>
<td>-5.99316</td>
<td>-6.01367</td>
<td>-2.30762</td>
<td>-2.30767</td>
<td>-2.3078</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( a = 0.2 )</th>
<th>( a = -0.2 )</th>
<th>( a = 0.8 )</th>
<th>( a = 1.2 )</th>
<th>( a = -1.2 )</th>
<th>( a = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
<td>( b = -3. )</td>
</tr>
<tr>
<td>( x^{(0)} = 10 )</td>
<td>( x^{(0)} = -10 )</td>
<td>( x^{(0)} = -10 )</td>
<td>( x^{(0)} = -10 )</td>
<td>( x^{(0)} = -10 )</td>
<td>( x^{(0)} = -10 )</td>
</tr>
<tr>
<td>-5.</td>
<td>-1.</td>
<td>-11.</td>
<td>-15.</td>
<td>9.</td>
<td>-23</td>
</tr>
<tr>
<td>-4.</td>
<td>-2.8</td>
<td>-11.8</td>
<td>-21.</td>
<td>-13.8</td>
<td>-49</td>
</tr>
<tr>
<td>-3.8</td>
<td>-2.44</td>
<td>-12.44</td>
<td>-28.2</td>
<td>13.56</td>
<td>-101</td>
</tr>
<tr>
<td>-3.76</td>
<td>-2.512</td>
<td>-12.952</td>
<td>-36.84</td>
<td>-19.272</td>
<td>-205</td>
</tr>
<tr>
<td>-3.75</td>
<td>-2.5</td>
<td>-14.3289</td>
<td>-113.995</td>
<td>43.1981</td>
<td>-6653</td>
</tr>
<tr>
<td>-3.75</td>
<td>-2.5</td>
<td>-14.4631</td>
<td>-139.793</td>
<td>-54.8377</td>
<td>-13309</td>
</tr>
</tbody>
</table>

From the first row of the table we learn two things: First, the starting point of the iteration has no effect on the success of the iteration, nor on its result. Second, the value of \( b \) only affects the value of the limit, not its existence. If the limit exists, it is equal to \( b/(1 - a) \). The second
row of the table shows that it is the absolute value of $a$ that controls convergence: if $|a| < 1$, the sequence converges, otherwise it diverges. The smaller $|a| < 1$, the faster the convergence, the larger $|a| > 1$, the faster the divergence.

The value of the limit, when it exists, should not be a surprise. Looking back at Equation 18, we see that for $k = \infty$ we get

$$x^{(\infty)} = a x^{(\infty)} + b,$$

a linear equation in $x^{(\infty)}$ that gives us the solution $x^{(\infty)} = b/(1 - a)$.

### 4.2.1 Back to the $n$-dimensional case

So what can we learn from the 1D case to apply to the matrix iteration? What condition relative to matrices could correspond to the simple condition $|a| < 1$? We can see $|a|$ in the 1D case as an “amplification” or “stretching” applied to the number $x$. In the case of matrices and vectors, the norm of the matrix captures a similar notion. For a given matrix $G$, we can conjecture that if $\|G\| < 1$ then the following iteration will converge:

$$x^{(k+1)} = G x^{(k)} + b,$$

and this, regardless of the value of $b$ and of the starting point of the iteration $x^{(0)}$. The limit of the iteration will be

$$x = (I - G)^{-1} b.$$

Note that we can use any norm for the condition $\|G\| < 1$, but that this condition is only sufficient.

Returning to the Jacobi iteration, we can see that a sufficient condition for the iteration to converge is that $\| - D^{-1}(L + U) \| < 1$. We can also see that the objective and challenge of any iteration scheme will be to produce an iteration matrix $G$ whose norm is as small as possible while keeping $I - G$ as well-conditioned as possible.

### 4.3 Gauss-Seidel iteration

The Gauss-Seidel iteration uses the same “trick” as the Jacobi iteration, except that it does not pass the $L$ matrix on the right side of the equation as in Eq. 16. Instead we write:

$$(D + L) x = -U x + b,$$  \hspace{1cm} (19)

so that our iteration now becomes

$$(D + L) x^{(k+1)} = -U x^{(k)} + b,$$  \hspace{1cm} (20)

that is, a lower-triangular SLE. Instead of inverting the matrix on the left side, it is much preferable to solve it by forward substitution.

So why is this good, when it looks more complicated that the Jacobi version? To see this you have to express both equations at the level of the components of the $x$ unknown vector.
For the Jacobi iteration

\[ x_i^{(k+1)} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right]. \]

For the Gauss-Seidel iteration

\[ x_i^{(k+1)} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right]. \]

Note that the Gauss-Seidel algorithms uses elements of the \( k + 1 \)th iterations to compute other elements of the \( k + 1 \)th iteration, whereas the Jacobi iteration uses only elements of the \( k \)th iteration. We say that Jacobi iteration is a simultaneous correction method, while Gauss-Seidel is a successive correction method.

### 4.4 One more iterative technique

This modified version of the Gauss-Seidel iteration is called successive over-relaxation, or SOR. The idea here is just to add \( x_i^{(k)} \) in one place of the Gauss-Seidel iteration and remove it in another place. We obtain

\[
x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right],
\]

\[ = x_i^{(k)} + \delta_i^{(k)}. \]

So what we are doing is just adding a correcting term to the current estimate \( x_i^{(k)} \). Obviously this makes it easier to check if the iteration is converging.