# Numerical Linear Algebra - Notes - v0.3.0

260236

November 2024

# Preface

Every theory section in these notes has been taken from the sources:

- Course slides. [1]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on numerical linear algebra. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

# Contents

# 1 Preliminaries

This section introduces some of the basic topics used throughout the course.

## 1.1 Notation

We try to use the same notation for anything.

- **Vectors**. With $\mathbb{R}$ is a set of real numbers (scalars) and $\mathbb{R}^n$ is a space of column vectors with $n$ real elements.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

  Vectors with all zeros and all ones:

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \mathbf{1} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

- **Matrices**. With $\mathbb{R}^{m \times n}$ is a space of $m \times n$ matrices with real elements:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

  Identity matrix $\mathbf{I} \in \mathbb{R}^{n \times n}$:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_n \end{bmatrix}$$

  Where $\mathbf{e}_i$, $i = 1, 2, \ldots, n$ are the canonical vectors.

$$\mathbf{e}_i = \begin{bmatrix} 0 & 0 & \cdots & 1 & \cdots & 0 & 0 \end{bmatrix}^T$$

  Where 1 is the $i$-th entry.

## 1.2 Matrix Operations

Some basic matrix operations:

- **Inner products**. If $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ then:

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1,\ldots,n} x_i y_i$$

For real vectors, the commutative property is true:

$$\mathbf{x}^T\mathbf{y} = \mathbf{y}^T\mathbf{x}$$

Furthermore, the vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are **orthogonal** if:

$$\mathbf{x}^T\mathbf{y} = \mathbf{y}^T\mathbf{x} = \mathbf{0}$$

And finally, some useful properties of matrix multiplication:

1. Multiplication by the *identity* changes nothing.

$$A \in \mathbb{R}^{n\times m} \;\Rightarrow\; \mathbf{I}_n A = A = A\mathbf{I}_m$$

2. Associativity:

$$A\,(BC) = (AB)\,C$$

3. Distributive:

$$A\,(B + D) = AB + AD$$

4. No commutativity:

$$AB \neq BA$$

5. Transpose of product:

$$(AB)^T = B^T A^T$$

- **Matrix powers**. For $A \in \mathbb{R}^{n\times n}$ with $A \neq \mathbf{0}$:

$$A^0 = \mathbf{I}_n \qquad A^k = \underbrace{A \cdots A}_{k \text{ times}} = AA^{k-1} \qquad k \geq 1$$

Furthermore, $A \in \mathbb{R}^{n\times n}$ is:

  - **Idempotent** (projector) $A^2 = A$
  - **Nilpotent** $A^k = \mathbf{0}$ for some integer $k \geq 1$

- **Inverse**. For $A \in \mathbb{R}^{n\times n}$ is **non-singular** (**invertible**), if exists $A^{-1}$ with:

$$AA^{-1} = \mathbf{I}_n = A^{-1}A \tag{1}$$

Inverse and transposition are interchangeable:

$$A^{-T} \triangleq \left(A^T\right)^{-1} = \left(A^{-1}\right)^T$$

Furthermore, an inverse of a product for a matrix $A \in \mathbb{R}^{n\times n}$ can be expressed as:

$$(AB)^{-1} = B^{-1}A^{-1}$$

Finally, remark that if $\mathbf{0} \neq \mathbf{x} \in \mathbb{R}^n$ and $A\mathbf{x} = 0$, then $A$ is **singular**.

- **Orthogonal matrices**. Given a matrix $A \in \mathbb{R}^{n \times n}$ that is *invertible*, the matrix $A$ is said to be **orthogonal** if:

$$A^{-1} = A^T \Rightarrow A^T A = \mathbf{I}_n = A A^T$$

- **Triangular matrices**. There are two types of triangular matrices:

  1. **Upper triangular matrix**:

$$\mathbf{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{bmatrix}$$

     $\mathbf{U}$ is **non-singular** if and only if $u_{ii} \neq 0$ for $i = 1, \ldots, n$.

  2. **Lower triangular matrix**:

$$\mathbf{L} = \begin{bmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \cdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{bmatrix}$$

     $\mathbf{L}$ is **non-singular** if and only if $l_{ii} \neq 0$ for $i = 1, \ldots, n$.

- **Unitary triangular matrices**. Are matrices similar to the lower and upper matrices, but they have the main diagonal composed of ones.

  1. **Unitary upper triangular matrix**:

$$\mathbf{U} = \begin{bmatrix} 1 & u_{1,2} & \cdots & u_{1,n} \\ 0 & 1 & \cdots & u_{2,n} \\ \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

  2. **Unitary lower triangular matrix**:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \cdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{bmatrix}$$

## 1.3 Basic matrix decomposition

In the Numerical Linear Algebra course, we will use three main decomposition:

- **LU factorization with (partial) pivoting**. If $A \in \mathbb{R}^{n \times n}$ is a non-singular matrix, then:
$$PA = LU$$
Where:
  - $P$ is a permutation matrix
  - $L$ is an unit lower triangular matrix
  - $U$ is an upper triangular matrix

  Note that the linear system solution:
$$A\mathbf{x} = \mathbf{b}$$
  Can be solved directly by calculation:
$$PA = LU$$
  This way the complexity is equal to $O\left(n^3\right)$. So a smarter way to reduce complexity is to use the *divide et impera* (or *divide and conquer*) technique. Then solve the system:
$$\begin{cases} L\mathbf{y} = P\mathbf{b} & \rightarrow \text{ unit lower triangular system, complexity } O\left(n^2\right) \\ U\mathbf{x} = \mathbf{y} & \rightarrow \text{ upper triangular system, complexity } O\left(n^2\right) \end{cases}$$

- **Cholesky decomposition**. If $A \in \mathbb{R}^{n \times n}$ is a symmetric[1] and positive definite[2], then:
$$A = L^T L$$
  Where $L$ is a lower triangular matrix (with positive entries on the diagonal). Also note that the linear system solution:
$$A\mathbf{x} = \mathbf{b}$$
  Can be solved directly by calculation:
$$A = L^T L$$
  This way the complexity is equal to $O\left(n^3\right)$. So a smarter way to reduce complexity is to use the *divide et impera* (or *divide and conquer*) technique. Then solve the system:
$$\begin{cases} L^T\mathbf{y} = \mathbf{b} & \rightarrow \text{ lower triangular system, complexity } O\left(n^2\right) \\ L\mathbf{x} = \mathbf{y} & \rightarrow \text{ upper triangular system, complexity } O\left(n^2\right) \end{cases}$$

[1] $A^T = A$
[2] $\mathbf{z}^T A\mathbf{z} > 0 \qquad \forall \mathbf{z} \neq 0$

- **QR decomposition**. If $A \in \mathbb{R}^{n \times n}$ is a non-singular matrix, then:

$$A = QR$$

Where:

  - $Q$ is an orthogonal matrix
  - $R$ is an upper triangular

Note that the linear system solution:

$$A\mathbf{x} = \mathbf{b}$$

Can be solved directly by calculation:

$$A = QR$$

This way the complexity is equal to $O\left(n^3\right)$. So a smarter way to reduce complexity is to use the *divide et impera* (or *divide and conquer*) technique. Then:

1. Multiply $\mathbf{c} = Q^T\mathbf{b}$, complexity $O\left(n^2\right)$
2. Solve the lower triangular system $R\mathbf{x} = \mathbf{c}$, complexity $O\left(n^2\right)$

## 1.4 Determinants

We will assume that the determinant topic is well known. However, in the following enumerated list there are some useful properties about the determinant of a matrix:

1. If a general matrix $T \in \mathbb{R}^{n \times n}$ is upper- or lower-triangular, then the determinant is computed as:

$$\det(T) = \prod_{i=1}^{n} t_{i,i}$$

2. Let $A, B \in \mathbb{R}^{n \times n}$, then is true:

$$\det(AB) = \det(A) \cdot \det(B)$$

3. Let $A \in \mathbb{R}^{n \times n}$, then is true:

$$\det\left(A^T\right) = \det(A)$$

4. Let $A \in \mathbb{R}^{n \times n}$, then is true:

$$\det(A) \neq 0 \iff A \text{ is non-singular}$$

5. **Computation**. Let $A \in \mathbb{R}^{n \times n}$ be non-singular, then:
   (a) Factor $PA = LU$
   (b) $\det(A) = \pm \det(U) = \pm u_{1,1} \dots u_{n,n}$

## 1.5 Sparse matrices

A **sparse matrix** is a matrix in which most of the elements are zero; roughly speaking, given $A \in \mathbb{R}^{n \times n}$, the number of non-zero entries of $A$ (denoted $\text{nnz}(A)$) is $O(n)$, we say that $A$ is **sparse**.

Sparse matrices are so important because when we try to solve:

$$A\mathbf{x} = \mathbf{b}$$

The $A$ matrix is often sparse, especially when it comes from the discretization of partial differential equations.

Finally, note that the iterative methods (explained in the next section) only use a sparse matrix $A$ in the context of the matrix-vector product. Then we only need to provide the matrix-vector product to the computer.

---

### 1.5.1 Storage schemes

Unfortunately, storing a sparse matrix is a waste of memory. Instead of storing a dense array (with many zeros), the main idea is to **store only the non-zero entries, plus their locations**.

This technique allows to save data storage because it will be from $O(n^2)$ to $O(\text{nnz})$.

The most common sparse storage types are:

- **Coordinate format (COO)**. The data structure consists of three arrays (of length $\text{nnz}(A)$):
  - `AA`: all the values of the non-zero elements of $A$ in any order.
  - `JR`: integer array containing their row indices.
  - `JC`: integer array containing their column indices.

  For **example**:

$$A = \begin{bmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{bmatrix}$$

$$\texttt{AA} = [12.\ \ 9.\ \ 7.\ \ 5.\ \ 1.\ \ 2.\ \ 11.\ \ 3.\ \ 6.\ \ 4.\ \ 8.\ \ 10.]$$

$$\texttt{JR} = [\ 5\ \ 3\ \ 3\ \ 2\ \ 1\ \ 1\ \ 4\ \ 2\ \ 3\ \ 2\ \ 3\ \ 4\ ]$$

$$\texttt{JC} = [\ 5\ \ 5\ \ 3\ \ 4\ \ 1\ \ 4\ \ 4\ \ 1\ \ 1\ \ 2\ \ 4\ \ 3\ ]$$

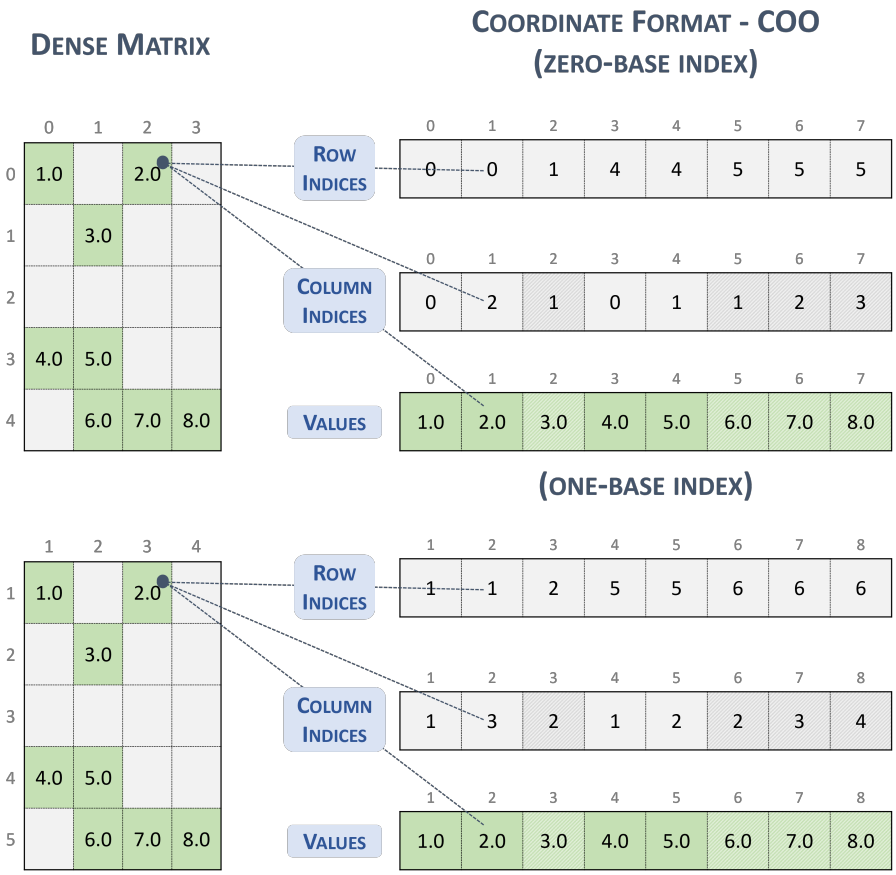


Figure 1: Graphical representation of the coordinate format (COO) technique. From the figure we can see the representation of the `AA` array, called *values*, the `JR`, called *row indices*, and finally the `JC`, called *column indices*. The algorithm is very simple. The figures are taken from the NVIDIA Performance Libraries Sparse, which is part of the NVIDIA Performance Libraries.

- **Coordinate Compressed Sparse Row format (CSR)**. If the elements of $A$ are listed by row, the array `JC` might be replaced by an array that points to the beginning of each row.
  - `AA`: all the values of the non-zero elements of $A$, stored row by row from $1, \ldots, n$.
  - `JA`: contains the column indices.
  - `IA`: contains the pointers to the beginning of each row in the arrays $A$ and `JA`. Thus `IA`$(i)$ contains the position in the arrays `AA` and `JA` where the $i$-th row starts. The length of `IA` is $n+1$, with `IA`$(n+1)$ containing the number $A(1) + \mathrm{nnz}(A)$. Remember that $n$ is the number of rows.

For **example**:

$$A = \begin{bmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{bmatrix}$$

$$\texttt{AA} = [1.\ \ 2.\ \ 3.\ \ 4.\ \ 5.\ \ 6.\ \ 7.\ \ 8.\ \ 9.\ \ 10.\ \ 11.\ \ 12.]$$

$$\texttt{JA} = [1\ \ 4\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\,]$$

$$\texttt{IA} = [1\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\,]$$

To retrieve each position of the matrix, the algorithm is quite simple. Consider the `IA` arrays.

1. We start at position one of the array, then the value 1:

   $$\texttt{AA} = [1.\ \ 2.\ \ 3.\ \ 4.\ \ 5.\ \ 6.\ \ 7.\ \ 8.\ \ 9.\ \ 10.\ \ 11.\ \ 12.]$$
   $$\texttt{JA} = [1\ \ 4\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\,]$$
   $$\texttt{IA} = [\textcircled{1}\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\,]$$

2. We use the value one to see the first (index one) position of the array JA, and the value is 1:

   $$\texttt{AA} = [1.\ \ 2.\ \ 3.\ \ 4.\ \ 5.\ \ 6.\ \ 7.\ \ 8.\ \ 9.\ \ 10.\ \ 11.\ \ 12.]$$
   $$\texttt{JA} = [\textcircled{1}\ \ 4\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\,]$$
   $$\texttt{IA} = [1\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\,]$$

3. But with the same index of `IA`, you also check the array `AA`, which has a value of 1:

   $$\texttt{AA} = [\textcircled{1}.\ \ 2.\ \ 3.\ \ 4.\ \ 5.\ \ 6.\ \ 7.\ \ 8.\ \ 9.\ \ 10.\ \ 11.\ \ 12.]$$
   $$\texttt{JA} = [1\ \ 4\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\,]$$
   $$\texttt{IA} = [1\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\,]$$

4. Now we can check the next row of the matrix. So we check the array `IA` at position 2 and get the value 3. But be careful! From 1 (the previously calculated value) to 3 (the value just taken) there is the value 2 in between. So we can assume that the value 2 is also in the first row.

   $$\texttt{AA} = [1.\ \ \textcircled{2}.\ \ 3.\ \ 4.\ \ 5.\ \ 6.\ \ 7.\ \ 8.\ \ 9.\ \ 10.\ \ 11.\ \ 12.]$$
   $$\texttt{JA} = [1\ \ \textcircled{4}\ \ 1\ \ 2\ \ 4\ \ 1\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\,]$$
   $$\texttt{IA} = [1\ \ 3\ \ 6\ \ 10\ \ 12\ \ 13\,]$$

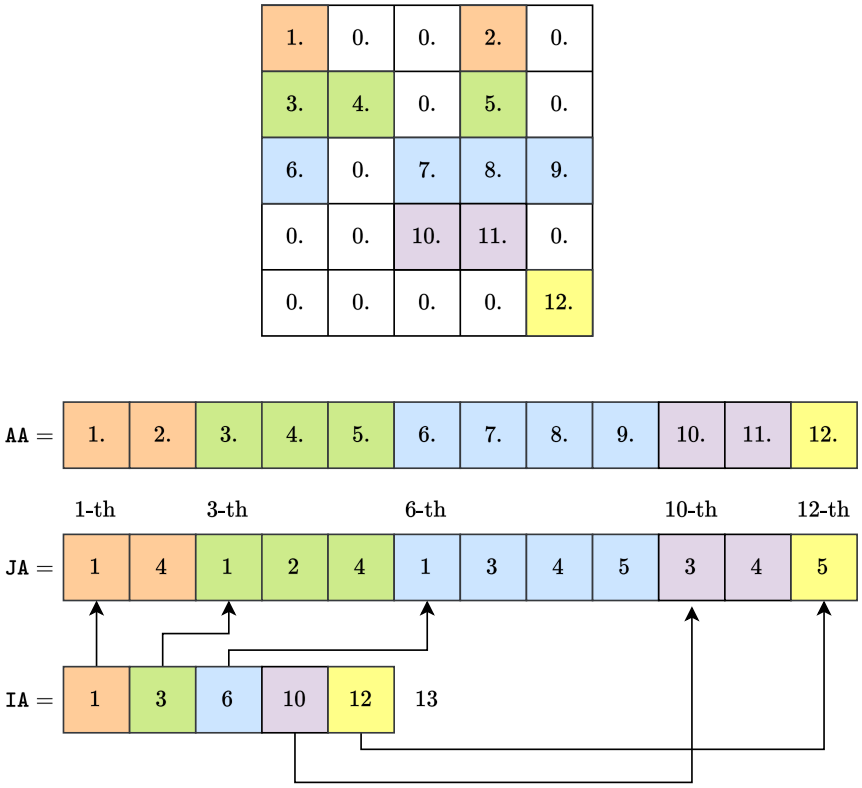


Figure 2: View an illustration of the CRS technique using colors to improve readability.

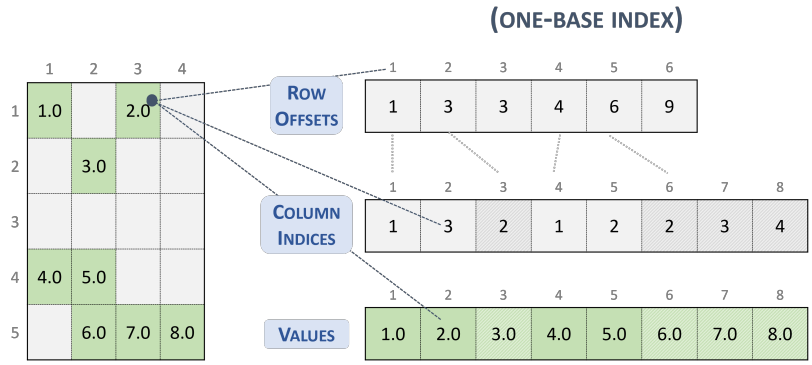


Figure 3: Graphical representation of the coordinate compressed sparse row (CSR) technique. From the figure we can see the representation of the `AA` array, called *values*, the `IA`, called *row offset*, and finally the `JA`, called *column indices*. It's interesting to see how the empty line case is handled. It copies the previous value of the array. The figures are taken from the NVIDIA Performance Libraries Sparse, which is part of the NVIDIA Performance Libraries.

# 2 Iterative methods for linear systems of equations

## 2.1 Why not use the direct methods?

Let us considering the following linear system of equations:

$$Ax = b$$

Where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x \in \mathbb{R}^n$ and $\det(A) \neq 0$. In general, direct methods are **not very suitable whenever**:

- $n$ **is large**. Typically, the average cost of direct methods scales as $n^3$, except in selected cases. As a trivial example, if peak performance is 1 PetaFLOPS ($10^{15}$ floating point operations per second), then

$$n = 10^7 \rightarrow \approx 10^6 \text{ seconds} \approx 11 \text{ days}$$

- **Matrix** $A$ **is sparse**. Direct methods suffer from the *fill-in* phenomenon[3] (see later). Unfortunately, sparse matrices are very popular in many application problems and we cannot consider them.

**Definition 1: Sparse Matrix**

Let $A \in \mathbb{R}^{n \times n}$ we say that $A$ is **sparse** the number of non-zero elements (abbreviated as $\text{nnz}(A)$) is approximately equal to the number of rows/columns $n$, i.e. $\text{nnz}(A) \sim n$.

**What is an iterative method?**

It is clear that iterative methods are usually better than direct methods. An **iterative method** is a **mathematical procedure that uses an initial value to generate a sequence of improving approximate solutions to a class of problems**, where the $i$-th approximation (called an "*iteration*") is derived from the previous ones.

More precisely, we introduce a sequence $\mathbf{x}^{(k)}$ of vectors determined by a recursive relation that identifies the method.

$$\mathbf{x}^{(0)} \rightarrow \mathbf{x}^{(1)} \rightarrow \cdots \rightarrow \mathbf{x}^{(k)} \rightarrow \mathbf{x}^{(k+1)} \rightarrow \cdots$$

To "*initialize*" the iterative process, it is necessary to provide a starting point (*initial vector*, also called *initial guess*) $\mathbf{x}^{(0)}$, e.g. based on physical/engineering applications.

[3] The fill-in of a matrix are those entries that change from an initial zero to a non-zero value during the execution of an algorithm. To reduce the memory requirements and the number of arithmetic operations used during an algorithm, it is useful to minimize the fill-in.

After initialization, the core of the process should, sooner or later, produce a result. It is a very complex and long topic, but in general it refers to the process by which an iterative algorithm approaches a fixed point or a solution to a problem after several iterations. An **iterative method must satisfy the convergence property**:

$$\lim_{k \to +\infty} \mathbf{x}^{(k)} = \mathbf{x} \tag{2}$$

It is important to note that the **convergence does not depend on the choice of the initial vector** $x^{(0)}$.

From the property 2, it should be clear that **convergence is guaranteed only after an $\infty$ number of iterations**. From a practical point of view, we need to stop the iteration process after a finite number of iterations when we are *sufficiently close* to the solution.

In addition to the *problem of convergence* and "*when should we stop our convergence method*", we have to deal with the *numerical error* inevitably introduced by our method.

These topics will be explained and faced in the following pages.

## 2.2 Linear iterative methods

### 2.2.1 Definition

In general, we consider linear iterative methods of the following form:

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{f} \qquad k \geq 0$$

Where $B \in \mathbb{R}^{n\times n}$, $\mathbf{f} \in \mathbb{R}^n$ and the matrix $B$ is called **iteration matrix**. The choice of the iteration matrix and $\mathbf{f}$ uniquely identifies the method.

The question is now automatic. **How to choose** an intelligent iteration matrix and $\mathbf{f}$? There are two main factors to consider:

- **Consistency**. This is a necessary condition, but not sufficient to guarantee the convergence. If $\mathbf{x}^{(k)}$ es the exact solution $\mathbf{x}$, then $\mathbf{x}^{(k+1)}$ is again equal to $\mathbf{x}$ (no update if the exact solution is found):

  $$\mathbf{x} = B\mathbf{x} + \mathbf{f} \longrightarrow \mathbf{f} = (I - B)\,\mathbf{x} = (I - B)\,A^{-1}\mathbf{b}$$

  The former identity gives a relationship between $B$ and $\mathbf{f}$ as a function of the data.

- **Convergence**. To study the convergence we need the error and the spectral radius:

  - **Error**. Let us introduce the error at step $(k+1)$:

    $$\mathbf{e}^{(k+1)} = \mathbf{x} - \mathbf{x}^{(k+1)}$$

    And an appropriate vector norm, such as the Euclidean norm $||\cdot||$. Then we have:

    $$\begin{aligned}
    \left|\left|\mathbf{e}^{(k+1)}\right|\right| &= \left|\left|\mathbf{x} - \mathbf{x}^{(k+1)}\right|\right| \\
    &= \left|\left|\mathbf{x} - \left(B\mathbf{x}^{(k)} + \mathbf{f}\right)\right|\right| \\
    &= \left|\left|\mathbf{x} - B\mathbf{x}^{(k)} - \mathbf{f}\right|\right| \\
    &= \left|\left|\mathbf{x} - B\mathbf{x}^{(k)} - (I - B)\,\mathbf{x}\right|\right| \\
    &= \left|\left|\mathbf{x} - B\mathbf{x}^{(k)} - I\mathbf{x} + B\mathbf{x}\right|\right| \\
    &= \left|\left|\mathbf{x} - B\mathbf{x}^{(k)} - \mathbf{x} + B\mathbf{x}\right|\right| \\
    &= \left|\left|-B\mathbf{x}^{(k)} + B\mathbf{x}\right|\right| \\
    &= \left|\left|B\left(\mathbf{x} - \mathbf{x}^{(k)}\right)\right|\right| \\
    &= \left|\left|B\mathbf{e}^{(k)}\right|\right| \\
    &\leq ||B|| \cdot \left|\left|\mathbf{e}^{(k)}\right|\right|
    \end{aligned}$$

    Note that $||B||$ is the matrix norm induced by the vector norm $||\cdot||$.

Using recursion, we get:

$$\begin{aligned}
\left|\left|\mathbf{e}^{(k+1)}\right|\right| &\leq ||B|| \cdot \left|\left|\mathbf{e}^{(k)}\right|\right| \\
&\leq ||B|| \cdot ||B|| \cdot \left|\left|\mathbf{e}^{(k-1)}\right|\right| \\
&\leq ||B|| \cdot ||B|| \cdot ||B|| \cdot \left|\left|\mathbf{e}^{(k-2)}\right|\right| \\
&\leq \dots \\
&\leq ||B||^{(k+1)} \cdot \left|\left|\mathbf{e}^{(0)}\right|\right| \\
\lim_{k\to\infty} \left|\left|\mathbf{e}^{(k+1)}\right|\right| &\leq \left(\lim_{k\to\infty} ||B||^{(k+1)}\right) \cdot \left|\left|\mathbf{e}^{(0)}\right|\right|
\end{aligned}$$

And here is the key. The **sufficient condition for convergence is to choose a matrix $B$ that has the norm less than** 1:

$$||B|| < 1 \Longrightarrow \lim_{k\to\infty} \left|\left|\mathbf{e}^{(k+1)}\right|\right| = 0$$

We recall that the *Euclidean norm* (commonly used) of a matrix is calculated by taking the square root of the sum of the absolute squares of its elements. Let $A$ be a matrix of size $m \times n$, the Euclidean norm:

$$||A||_2 \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

– **Spectral radius**. The spectral radius of a matrix is the **largest absolute value of its eigenvalues**. We define:

$$\rho(B) = \max_j |\lambda_j(B)|$$

Where $\lambda_j(B)$ are the eigenvalues of $B$.
Why is the spectral radius useful? Well, if the matrix $B$ is symmetric positive definite (SPD)[4], then the spectral radius is equal to the Euclidean norm of the matrix.

$$B \text{ is SPD } \Rightarrow ||B||_2 = \rho(B) \wedge \rho(B) < 1 \iff \text{method convergences}$$

And this is a very big help to us for many reasons.

* **Balance and Predictability**. When the norm is equal to the spectral, it means that the influence of the matrix is well distributed. In other words, this uniformity can help make our iterative methods more predictable, reducing the possibility of non-convergence.
* **Efficiency**. It avoids scenarios where the matrix might have hidden large entries affecting convergence or stability.

[4] **SPD (Symmetric Positive Definite)** is a matrix:
* Symmetric: $A = A^T$
* Positive Definite: $x^T AX > 0$, $\forall x \in \mathbb{R}^n \setminus \{0\}$

Let $C \in \mathbb{R}^{n\times n}$ then the spectral radius of a matrix is equal to the infimum (lower bound) of its matrix norm:

$$\rho(C) = \inf\{||C|| \ \ \forall \text{ induced matrix norm } ||\cdot||\} \tag{3}$$

It follows from this property that:

$$\rho(B) \leq ||B|| \qquad \forall \text{induced matrix norm } ||\cdot|| \tag{4}$$

Note that thanks to 4 we can observe that if:

$$\exists\, ||\cdot|| \text{ such that } ||B|| < 1 \implies \rho(B) < 1$$

The convergence of the method is guaranteed by the following theorem.

**Theorem 1** (**necessary and sufficient condition for convergence**). *A* ***consistent*** *iterative method with iteration matrix $B$ converges if and only if $\rho(B) < 1$.*

### 2.2.2 Jacobi method

Let the problem of solve $Ax = b$, where $A$ is a square matrix, $x$ is the vector of unknowns, and $b$ is the result vector.

We start from the $i$-th line of the linear system:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \;\rightarrow\; a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

Formally the solution $x_i$ for each $i$ si given by:

$$x_i = \frac{b_i - \sum_{j \neq i} a_{ij}x_j}{a_{ii}} \tag{5}$$

Obviously the previous identity cannot be used in practice because we do not know $x_j$, for $j \neq i$. And here is the **magic idea** of Jacobi: we could think of introducing an iterative method (Jacobi) that **updates** $x_i^{(k+1)}$ **step** $k+1$ **using the other** $x_j^{(k)}$ **obtained in the previous step** $k$.

$$x_i = \frac{b_i - \sum_{j \neq i} a_{ij}x_j}{a_{ii}} \xrightarrow{\text{as } x_j \text{ is not well known}} x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}}{a_{ii}} \tag{6}$$

Where $\forall i = 1, \ldots, n$.

#### Algorithm

1. **Start with an initial guess** $\mathbf{x}^{(0)}$, also zero.
2. **Update each component** $\mathbf{x}_i^{(k+1)}$ using the equation 6.
3. **Repeat until the changes are less than a specified tolerance** or we haven't found the exact solution (in practice very difficult, almost impossible).

#### How much does it cost?

It depends on the matrix used:

- **Dense matrix** (bad choice). Each iteration costs $\approx n^2$ operations, so the Jacobi method is competitive if the number of iteration is less than $n$.
- **Sparse matrix** (good choice). Each iteration costs only $\approx n$ operations.

#### Can it be parallelized?

The parallelization of the Jacobi method is actually **one of its main advantages** on modern computers. Each update of $x_i$ depends only on the previous values of the other $x_j$, not on the current iteration values. This independence makes it easy to distribute the work across multiple processors.

### 2.2.3 Gauss-Seidel method

Given the Jacobi method, the Gauss Seidel method is similar, but with one clever difference: it uses the latest available values during iterations.

$$x_i^{(k+1)} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}}{a_{ii}} \tag{7}$$

At iteration $(k+1)$, let's consider the computation of $x_i^{(k+1)}$. We observe that for $j < i$ (with $i \geq 2$), $x_j^{(k+1)}$ is known (we have already calculated it). We can therefore think of using the quantities at step $(k+1)$ if $j < i$ and, as in the Jacobi method, those at the previous step $k$ if $j > i$.

#### Algorithm

1. **Start with an initial guess $\mathbf{x}^{(0)}$**, also zero.
2. **Iteration**. For each row $i$ from 1 to $n$ calculate the value of the equation 7.
3. **Repeat until the changes are less than a specified tolerance**.

#### How much does it cost?

The cost is comparable to the Jacobi method explained on page 19.

#### Can it be parallelized?

Unlike the Jacobi method, the Gauss-Seidel method relies on the most recent updates within the same iteration. This sequential dependency **makes it more difficult to parallelize, as each update depends on the previous ones**.

While it's harder to parallelize due to its inherent sequential nature, we can still achieve some degree of parallelism with clever strategies such as red-black ordering. This makes the Gauss-Seidel method less straightforward to parallelize than Jacobi, but not impossible.

### 2.2.4 Convergence of Jacobi and Gauss-Seidel methods

Let be a general matrix $A$, and :

- $D$ the **diagonal part** of $A$
- $-E$ **lower triangular part** of $A$
- $-F$ **upper triangular part** of $A$

$$A = \begin{bmatrix} \ddots & & -F \\ & D & \\ -E & & \ddots \end{bmatrix}$$

The previous Jacobi and Gauss-Seidel methods can be rewritten as:

- Jacobi:
  - Method:
  $$D\mathbf{x}^{(k+1)} = (E+F)\,\mathbf{x}^{(k)} + \mathbf{b}$$
  - Iteration matrix:
  $$B_J = D^{-1}(E+F) = D^{-1}(D-A) = I - D^{-1}A$$
- Gauss-Seidel
  - Method:
  $$(D-E)\,\mathbf{x}^{(k+1)} = F\mathbf{x}^{k} + \mathbf{b}$$
  - Iteration matrix:
  $$B_{GS} = (D-E)^{-1}F$$

We present a theorem which gives us the **sufficient condition for convergence** of the Jacobi and Gauss-Seidel methods.

**Theorem 2** (**sufficient condition for convergence of Jacobi and Gauss–Seidel**). *The following conditions are sufficient for convergence:*

- *If a matrix $A$ is **strictly diagonally dominant by rows**:*
  $$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \qquad i = 1, \ldots, n$$
  *Then Jacobi and Gauss-Seidel converge.*
- *If a matrix $A$ is **strictly diagonally dominant by columns**:*
  $$|a_{ii}| > \sum_{j \neq i} |a_{ji}| \qquad i = 1, \ldots, n$$
  *Then Jacobi and Gauss-Seidel converge.*
- *If a matrix $A$ is SPD (symmetric positive and definite), then the Gauss-Seidel method is convergent.*

- *If a matrix $A$ is tridiagonal*[5]*, then the square spectral value of the Jacobi iteration matrix is equal to the spectral value of the Gauss-Seidel iteration matrix.*

$$\rho^2 (B_J) = \rho (B_{GS})$$

[5] A matrix is **tridiagonal** when it has non-zero elements only on the main diagonal, the diagonal above the main diagonal, and the diagonal below the main diagonal.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,3} & a_{4,4} \end{bmatrix}$$

### 2.2.5 Stationary Richardson method

The stationary Richardson method is a way of refining a guess for solving the general problem $Ax = b$. We **start with an initial guess for the solution**, then we **keep adjusting that guess based on how far it is from the actual answer**. The **adjustments depend on a parameter we choose**, which can speed up or slow down how quickly we get to the right answer. We **keep doing this until our guess is close enough to the actual solution**.

Mathematically, given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$, the stationary Richardson method is based on the following recursive update:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \cdot \underbrace{\left(\mathbf{b} - A\mathbf{x}^{(k)}\right)}_{\text{residual } \mathbf{r}^{(k)}} \tag{8}$$

The idea is to update the numerical solution by adding a quantity proportional to the residual. Indeed, it is expected that if the residual is *large* (*small*), the solution at step $k$ should be corrected *much* (*little*). Where $\alpha$ is a weighted version of the residual.

- Iteration matrix $B_\alpha$:
$$B_\alpha = I - \alpha A$$
- $\mathbf{f}$:
$$\mathbf{f} = \alpha \mathbf{b}$$

We now ask ourselves **which value of the parameter** $\alpha$, among those that **guarantee convergence**, **maximizes the speed of convergence**. We introduce the following $A$-induced norm where A is SPD:

$$||\mathbf{z}||_A = \sqrt{\sum_{i,j=1}^{n} a_{ij} z_i z_j} \iff ||\mathbf{z}||_A = \sqrt{(A\mathbf{z}, \mathbf{z})} = \sqrt{\mathbf{z}^T A \mathbf{z}}$$

We look for $0 < \alpha_{\text{opt}} < \dfrac{2}{\lambda_{\max(A)}}$ such that $\rho(B_\alpha)$ is minimum. That is:

$$\alpha_{\text{opt}} = \underset{0<\alpha<\frac{2}{\lambda_{\max(A)}}}{\operatorname{argmin}} \left\{ \max_i |1 - \alpha \lambda_i(A)| \right\}$$

To understand which $\alpha$ to choose, we plot the problem. On the *x-axis* are the values of $\alpha$ and on the *y-axis* is the spectral radius equal to $|1 - \alpha\lambda_i(A)|$, with $i = 1, \ldots, n$.

In the figure 4 we can see that the upper bound of the spectral radius is equal to 1 (no convergence). Each line represents the possible value of the spectral radius for different values of $\alpha$. In **green** we see the **spectral radius equal to** $\rho(B_\alpha)$; it is important because its intersection with the upper bound of $\rho$ represents the right bound of the interval where the **values of** $\alpha$ **guarantee convergence**. It can also be seen by the **red arrow**. The **lowest point of the curve is where the spectral radius is minimized, indicating the best** $\alpha$ **for convergence**.

In other words, the optimal value is given by the intersection between the curves:

$$|1 - \alpha\lambda_1(A)| \cap |1 - \alpha\lambda_n(A)|$$

That gives us the perfect formula:

$$\alpha_{\text{opt}} = \frac{2}{\lambda_{\min}(A) + \lambda_{\max}(A)} \tag{9}$$

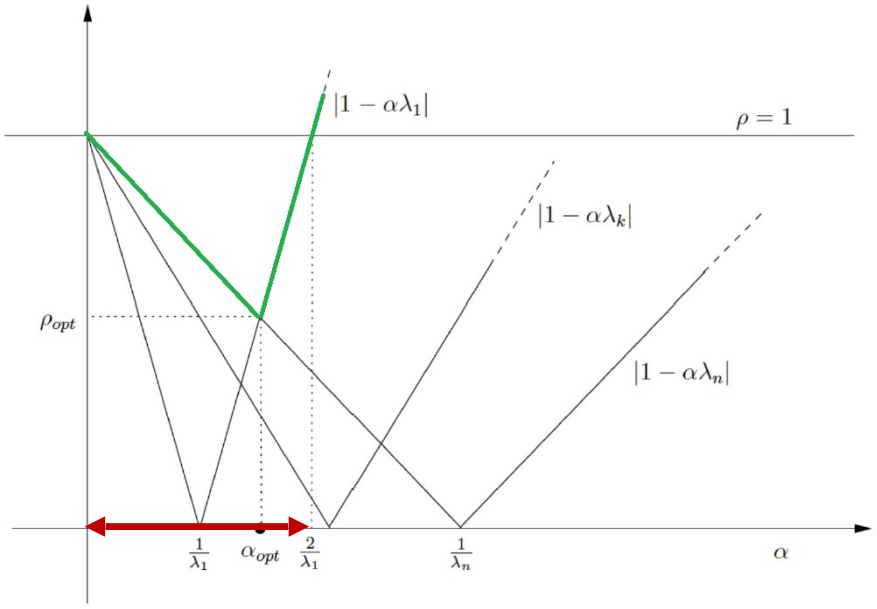


Figure 4: Graphical representation of the optimal alpha to choose in the stationary Richardson method.

If A is SPD, the eigenvalues of $A$ (real and positive) are:

$$\lambda_{\max}(A) = \lambda_1(A) \geq \lambda_2(A) \geq \cdots \geq \lambda_n(A) = \lambda_{\min}(A) > 0$$

**Theorem 3.** *Let $A$ be a symmetric and positive definite matrix. The* ***stationary Richardson method is convergent if and only if:***

$$0 < \alpha < \frac{2}{\lambda_{\max}(A)} \tag{10}$$

Since there is a strong correlation between the optimal $\alpha$ and the optimal spectral radius, we can obtain

$$\begin{aligned} \rho_{\text{opt}} &= \rho\left(B_{\alpha_{\text{opt}}}\right) \\ &= -1 + \alpha_{\text{opt}}\lambda_{\max}(A) \\ &= 1 - \alpha_{\text{opt}}\lambda_{\min}(A) \\ &= \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{\lambda_{\max}(A) + \lambda_{\min}(A)} \end{aligned}$$

Finally, since $A$ is SPD, we have the Euclidean norm equal to the maximum eigenvalue of A: $||A||_2 = \lambda_{\max}(A)$. Moreover, $\lambda_i\left(A^{-1}\right) = \frac{1}{\lambda_i(A)}$, $i = 1, \ldots, n$:

$$\rho_{\text{opt}} = \frac{K(A) - 1}{K(A) + 1} \tag{11}$$

### Algorithm

1. **Start with an initial guess $\mathbf{x}^{(0)}$** and **select a parameter** $\alpha$.
2. **Iteration**. For each $k$ calculate the value of the equation 8.
3. **Repeat until the changes are less than a specified tolerance**.

### How much does it cost?

The cost of each iteration depends by type of matrix:

- **Dense matrix**: the cost of each iteration is about $n^2$ **operations**, where $n$ is the number of unknowns in the linear system.
- **Sparse matrix**: the cost of each iteration is only about $n$ **operations**.

### Can it be parallelized?

The stationary Richardson method is not as easily parallelizable as the Jacobi method. Richardson uses the entire solution vector from the previous iteration in each step. This dependency makes it **more difficult to parallelize**.

## 2.3 Stopping Criteria

A practical test is needed to determine when to stop the iteration. The **main idea** is that we stop iterations when:

$$\frac{\left|\left|\mathbf{x}-\mathbf{x}^{(\mathbf{k})}\right|\right|}{\left|\left|\mathbf{x}^{(k)}\right|\right|} \leq \varepsilon$$

Where $\varepsilon$ is a **user defined tolerance**. Meanwhile, the error (left side of the equation) is unknown! There are two criteria we can use to replace it:

- **Residual-based stopping criteria**. It looks at the *residual*, which is the difference between the current solution and the one obtained by reapplying the method's equation:

  $$r^{(k)} = b - Ax^{(k)}$$

  This residual gets smaller as the solution gets closer to the exact answer. When it's small enough, the iteration stops. This approach works because the residual essentially tracks the behaviour of the error. When the residual is small, the error is usually small.

  From a mathematical point of view:

  $$\frac{\left|\left|\mathbf{x}-\mathbf{x}^{(\mathbf{k})}\right|\right|}{\left|\left|\mathbf{x}^{(k)}\right|\right|} \leq K\left(A\right)\frac{\left|\left|\mathbf{r}^{(k)}\right|\right|}{||\mathbf{b}||} \Longrightarrow \frac{\left|\left|\mathbf{r}^{(k)}\right|\right|}{||\mathbf{b}||} \leq \varepsilon$$

  Where $K\left(A\right)$ is the **condition number** of $A$. It is a measure of **how sensitive the solution of a system of linear equations is to errors in the data or errors in the solution process**.

  - A **low condition number** (close to 1) means that the matrix is well conditioned, and **small errors in the data will cause only small errors in the solution**.
  - A **high condition number** indicates that the matrix is poorly conditioned, and even **small errors in the data can lead to large errors in the solution**.

  To reduce the condition number and the error, we need to use a preconditioner on the main matrix $A$. So instead of solving the general problem $Ax = b$ directly, we choose a preconditioner $P$ and solve $P^{-1}Ax = P^{-1}b$:

  $$\frac{\left|\left|\mathbf{x}-\mathbf{x}^{(\mathbf{k})}\right|\right|}{\left|\left|\mathbf{x}^{(k)}\right|\right|} \leq K\left(P^{-1}A\right)\frac{\left|\left|\mathbf{z}^{(k)}\right|\right|}{||\mathbf{b}||} \Longrightarrow \frac{\left|\left|\mathbf{z}^{(k)}\right|\right|}{||\mathbf{b}||} \leq \varepsilon \qquad \mathbf{z}^{(k)} = P^{-1}\mathbf{r}^{(\mathbf{k})}$$

- **Distance between consecutive iterates criteria**. It looks at **how much the current iterate (solution) changes compared to the previous one. When this difference becomes small enough, it's a signal that the method is converging and can be stopped**.

  Mathematically, define:

  $$\delta^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \Longrightarrow \left|\left|\delta^{(k)}\right|\right| \leq \varepsilon \Longrightarrow \left|\left|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right|\right|$$

With some manipulation, we can also demonstrate the relation between the true error and $\delta^{(k)}$:

$$\left|\left|\mathbf{e}^{(k)}\right|\right| \leq \frac{1}{1-\rho\left(B\right)} \cdot \left|\left|\delta^{(k)}\right|\right|$$

Indeed:

$$\begin{aligned} \left|\left|\mathbf{e}^{(k)}\right|\right| &= \left|\left|\mathbf{x}-\mathbf{x}^{(k)}\right|\right| \\ &= \left|\left|\mathbf{x}-\mathbf{x}^{(k+1)}+\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right|\right| \\ &= \left|\left|\mathbf{e}^{(k+1)}+\delta^{(k)}\right|\right| \\ &\leq \rho\left(B\right)\cdot\left|\left|\mathbf{e}^{(k)}\right|\right|+\left|\left|\delta^{(k)}\right|\right| \end{aligned}$$

## 2.4 Preconditioning techniques

Preconditioning techniques are used to **improve the convergence rate** of iterative methods for solving linear systems.

The optimal spectral radius $\rho_{\text{opt}}$ (equation 11, page 25) expresses the maximum convergence speed that can be achieved with a stationary Richardson method. Unfortunately, **badly conditioned matrices** (where $K(A) \gg 1$) are characterized by a **very low convergence rate**. So how can we improve the convergence rate?

The main idea is to introduce a symmetric positive definite matrix $P^{-1}$, called a **preconditioner**. Then the solution of the general problem is equivalent to the following preconditioned system:

$$A\mathbf{x} = \mathbf{b} \equiv P^{-\frac{1}{2}} A P^{-\frac{1}{2}} \mathbf{z} = P^{-\frac{1}{2}} \mathbf{b} \tag{12}$$

Where $\mathbf{x} = P^{-\frac{1}{2}}\mathbf{z}$. In general, the rule of thumb is to use a $P^{-1}$ such that $K\left(P^{-\frac{1}{2}} A P^{-\frac{1}{2}}\right) \ll K(A)$.

Suppose that $P^{-1}A$ has real and positive eigenvalues. We apply the stationary Richardson method to $P^{-1}A$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha P^{-1}\left(\mathbf{b} - A\mathbf{x}^{(k)}\right) = \mathbf{x}^{(k)} + \alpha P^{-1}\mathbf{r}^{(k)} \tag{13}$$

We obtain the same convergence results as in the non-preconditioned case, provided we replace $A$ with $P^{-1}A$:

- **Preconditioned convergence**:

$$0 < \alpha < \frac{2}{\lambda_{\max}\left(P^{-1}A\right)} \tag{14}$$

- **Preconditioned optimal values**:
  - Optimal alpha:

$$\alpha_{\text{opt}} = \frac{2}{\lambda_{\min}\left(P^{-1}A\right) + \lambda_{\max}\left(P^{-1}A\right)} \tag{15}$$

  - Optimal spectral radius:

$$\rho_{\text{opt}} = \frac{K\left(P^{-1}A\right) - 1}{K\left(P^{-1}A\right) + 1} \tag{16}$$

Since $K\left(P^{-1}A\right) \ll K(A)$ we obtain a higher convergence rate, we can conclude that the preconditioner method is faster than the non-preconditioned case? Well, the topic is little more complicated. **Preconditioning usually makes iterative methods converge faster** because it improves the condition number of the system. However, the effectiveness of preconditioning depends on the specific problem and the preconditioner chosen. In **some cases**, the **overhead of applying the preconditioner can offset its benefits**, so while preconditioning generally helps, it's not a guaranteed speedup every time.

### 2.4.1 Preconditioned Richardson method

The stationary Richardson method explained on page 23 is the same in this case, but we also choose to apply a preconditioner.

Remember that:

- The core of the stationary Richardson method defined on page 28 is:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha P^{-1}\left(\mathbf{b} - A\mathbf{x}^{(k)}\right) = \mathbf{x}^{(k)} + \alpha P^{-1}\mathbf{r}^{(k)}$$

- The preconditioned residual:

$$\mathbf{z}^{(k)} = P^{-1}\mathbf{r}^{(k)}$$

We define the pseudo-algorithm as follows. For any $k = 0, 1, 2, \ldots$:

1. **Compute**
$$\alpha_{\text{opt}} = \frac{2}{\lambda_{\min}\left(P^{-1}A\right) + \lambda_{\max}\left(P^{-1}A\right)}$$
2. **Update**
$$\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$$
3. **Solve**
$$P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$$
4. **Update**
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_{\text{opt}}\mathbf{z}^{(k)}$$

## 2.5 Gradient method

The Gradient method **uses the gradient to find the most efficient path to the minimum**. Although the gradient of a function gives the direction to the maximum of a function, if we go the opposite way, we find the minimum. This is the most basic and general idea.

### Algorithm

1. **Start with an initial guess** $\mathbf{x}^{(0)}$ and an **initial residual** as $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$.
2. **Iteration**. For each $k$ calculate:
   (a) The parameter $\alpha_k$:
   $$\alpha_k = \frac{\left(\mathbf{r}^{(k)}\right)^T \mathbf{r}^{(k)}}{\left(\mathbf{r}^{(k)}\right)^T A\mathbf{r}^{(k)}} \tag{17}$$
   (b) The step $k+1$:
   $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)} \tag{18}$$
   (c) The next residual:
   $$\mathbf{r}^{(k+1)} = (I - \alpha_k A)\,\mathbf{r}^{(k)} \tag{19}$$
3. **Repeat until the changes are less than a specified tolerance**.

Where the **convergence rate** is:

$$\left\|\mathbf{e}^{(k)}\right\|_A \leq \left(\frac{K(A)-1}{K(A)+1}\right)^k \cdot \left\|\mathbf{e}^{(0)}\right\|_A \tag{20}$$

### $ How much does it cost?

The cost of each iteration depends by type of matrix:

- **Dense matrix**: the cost of each iteration is about $n^2$ **operations**.
- **Sparse matrix**: the cost of each iteration is only about $n$ **operations**.

### Can it be parallelized?

Parallelizing the gradient method involves distributing the computation of gradients and their applications across multiple processors. Then, yes, it is possible.

## 2.6 Conjugate Gradient method

The **Conjugate Gradient method (GC)** is essentially an iterative algorithm used to solve large linear systems. It is similar to the gradient method, but instead of just following the steepest path, it chooses directions that are conjugate to each other. This avoids backtracking and converges more quickly.

**Theorem 4.** *In exact arithmetic the Conjugate Gradient method (GC) converges to the exact solution in at most n iterations. At each iteration k, the error* $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$ *can be bounded by:*

$$\left\|\mathbf{e}^{(k)}\right\|_A \leq \frac{2c^k}{1+c^{2k}} \cdot \left\|\mathbf{e}^{(0)}\right\|_A \tag{21}$$

*With:*

$$c = \frac{\sqrt{K(A)}-1}{\sqrt{K(A)}+1} \tag{22}$$

**Conjugate Gradient Algorithm**

1. **Start with an *initial guess*** $\mathbf{x}^{(0)}$, an ***initial residual*** as $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, and the ***initial direction*** $\mathbf{d}^{(0)} = \mathbf{r}^{(0)}$.
2. **Iteration**. For each $k$ calculate:
   (a) The parameter $\alpha_k$:
   $$\alpha_k = \frac{\left(\mathbf{d}^{(k)}\right)^T \mathbf{r}^{(k)}}{\left(\mathbf{d}^{(k)}\right)^T A\mathbf{d}^{(k)}} \tag{23}$$
   (b) The step $k+1$ along the direction $k$:
   $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)} \tag{24}$$
   (c) The next residual $k+1$:
   $$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{d}^{(k)} \tag{25}$$
   (d) The parameter $\beta_k$:
   $$\beta_k = \frac{\left(A\mathbf{d}^{(k)}\right)^T \mathbf{r}^{(k+1)}}{\left(A\mathbf{d}^{(k)}\right)^T \mathbf{d}^{(k)}} \tag{26}$$
   (e) The new direction $k+1$:
   $$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{d}^{(k)} \tag{27}$$
3. **Repeat until the changes are less than a specified tolerance**.

Each new direction is orthogonal (or conjugate) to all previous directions. This orthogonality ensures that each step optimally reduces the error without undoing the progress made in previous steps.

### Preconditioned Conjugate Gradient Algorithm

The CG method is modified by introducing $A$ and $P$ as symmetric, positive and definite matrices. The preconditioned system is:

$$\underbrace{P^{-1}AP^{-T}}_{\widehat{A}}\underbrace{P^{T}\mathbf{x}}_{\widehat{\mathbf{x}}}=\underbrace{P^{-1}\mathbf{b}}_{\widehat{\mathbf{b}}}$$

1. **Start with an *initial guess*** $\mathbf{x}^{(0)}$, an ***initial residual*** as $\mathbf{r}^{(0)}=\mathbf{b}-A\mathbf{x}^{(0)}$, and the ***initial direction*** $\mathbf{d}^{(0)}=\mathbf{r}^{(0)}$.
2. **Iteration**. For each $k$ calculate:
   (a) The parameter $\alpha_k$:
   $$\alpha_k=\frac{\left(\mathbf{z}^{(k)}\right)^T\mathbf{r}^{(k)}}{\left(A\mathbf{d}^{(k)}\right)^T A\mathbf{d}^{(k)}} \tag{28}$$
   (b) The step $k+1$ along the direction $k$:
   $$\mathbf{x}^{(k+1)}=\mathbf{x}^{(k)}+\alpha_k\mathbf{d}^{(k)} \tag{29}$$
   (c) The next residual $k+1$:
   $$\mathbf{r}^{(k+1)}=\mathbf{r}^{(k)}-\alpha_k A\mathbf{d}^{(k)} \tag{30}$$
   (d) Compute the action of the preconditioner $P$ on $\mathbf{r}^{(k+1)}$:
   $$P\mathbf{z}^{(k+1)}=\mathbf{r}^{(k+1)} \tag{31}$$
   (e) The parameter $\beta_k$:
   $$\beta_k=\frac{\left(A\mathbf{d}^{(k)}\right)^T\mathbf{z}^{(k+1)}}{\left(A\mathbf{d}^{(k)}\right)^T\mathbf{d}^{(k)}} \tag{32}$$
   (f) The new direction $k+1$:
   $$\mathbf{d}^{(k+1)}=\mathbf{z}^{(k+1)}-\beta_k\mathbf{d}^{(k)} \tag{33}$$
3. **Repeat until the changes are less than a specified tolerance**.

With the equations 21 and 22, the **preconditioner is considered good if**:

$$\frac{\sqrt{K\left(P^{-1}A\right)}-1}{\sqrt{K\left(P^{-1}A\right)}+1}<\frac{\sqrt{K\left(A\right)}-1}{\sqrt{K\left(A\right)}+1} \tag{34}$$

### $ How much does it cost?

The cost of each iteration depends by type of matrix:

- **Dense matrix**: the cost of each iteration is about $n^2$ **operations**.
- **Sparse matrix**: the cost of each iteration is only about $n$ **operations**.

### Can it be parallelized?

The Conjugate Gradient method has some parts that can be parallelized, such as: matrix-vector products, dot products, and vector updates. However, **some operations** (such as dot products) **require global synchronization**, which can **limit the efficiency of parallelization**. So while we can parallelize parts of it, the method as a whole isn't perfectly parallelizable.

## 2.7 Krylov-space

Krylov space methods are a group of iterative techniques used to solve large linear systems or eigenvalue problems. These methods construct a sequence of subspaces, called Krylov subspaces, which are iteratively expanded to approximate the solution.

**Definition 2: Krylov (sub)space**

Given a nonsingular $A \in \mathbb{R}^{n\times n}$ and $\mathbf{y} \in \mathbb{R}^n$, $\mathbf{y} \neq \mathbf{0}$, the $k$th Krylov (sub)space $\mathcal{K}_k(A, \mathbf{y})$ generated by $A$ from $\mathbf{y}$ is:

$$\mathcal{K}_k(A, \mathbf{y}) = \operatorname{span}\left(\mathbf{y}, A\mathbf{y}, \ldots, A^{k-1}\mathbf{y}\right) \tag{35}$$

Clearly, it holds:

$$\mathcal{K}_1(A, \mathbf{y}) \subseteq \mathcal{K}_2(A, \mathbf{y}) \subseteq \cdots$$

It seems clever to choose the $k$th approximate solution $\mathbf{x}^{(k)}$:

$$\mathbf{x}^{(k)} \in \mathbf{x}^{(0)} + \mathcal{K}_k\left(A, \mathbf{r}^{(0)}\right)$$

But can we expect to find the exact solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{b}$ in one of those affine space?

**Lemma 5.** *Let $\mathbf{x}$ be the solution of $A\mathbf{x} = \mathbf{b}$ and let $\mathbf{x}^{(0)}$ be any initial approximation of it and $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$ the corresponding residual. Moreover, let $v = v\left(\mathbf{r}^{(0)}, A\right)$ be the so called **grade of $\mathbf{r}^{(0)}$ with respect to** $A$. Then:*

$$\mathbf{x} \in \mathbf{x}^{(0)} + \mathcal{K}_v\left(A, \mathbf{r}^{(0)}\right)$$

**Lemma 6.** *There is a positive integer $\nu = \nu\left(\mathbf{r}^{(0)}, A\right)$ called **grade of $\mathbf{y}$ with respect to** $A$, such that:*

$$\begin{aligned} \dim\left(\mathcal{K}_s(A, y)\right) &= s \text{ if } s \leq \nu \\ \dim\left(\mathcal{K}_s(A, y)\right) &= \nu \text{ if } s \geq \nu \end{aligned}$$

*$\mathcal{K}_\nu(A, y)$ is the smallest $A$-invariant subspace that contains $\mathbf{y}$.*

**Lemma 7.** *The nonnegative integer $\nu = \nu(\mathbf{y}, A)$ of $\mathbf{y}$ with respect to $A$ satisfies:*

$$\nu(\mathbf{y}, A) = \min\left\{s \mid A^{-1}\mathbf{y} \in \mathcal{K}_s(A, y)\right\}$$

The idea behind Krylov space solvers is to **generate a sequence of approximate solutions $\mathbf{x}^{(k)} \in \mathbf{x}^{(0)} + \mathcal{K}_k\left(A, \mathbf{r}^{(0)}\right)$ of $A\mathbf{x} = \mathbf{b}$ so that the corresponding residuals $\mathbf{r}^{(k)} \in \mathcal{K}_{k+1}\left(A, \mathbf{r}^{(0)}\right)$ *converge* to the zero vector $\mathbf{0}$**.

The *converge* may also **mean that after a finite number of steps, $\mathbf{r}^{(k)} = \mathbf{0}$**, so that $\mathbf{x}^{(k)} = \mathbf{x}$ and the process stops. This is especially true (in exact arithmetic) if a **method ensures that the residuals are linearly independent**: then $\mathbf{r}^{(\nu)} = \mathbf{0}$. In this case, we say that the **method has the property of finite termination**.

**Definition 3: (standard) Krylov space**

A (standard) Krylov space method for solving a linear system $A\mathbf{x} = \mathbf{b}$ or, briefly, a Krylov space solver is an iterative method starting from some initial approximation $\mathbf{x}^{(0)}$ and the corresponding residual $\mathbf{r}^{(0)}$ and generating for all, or at least most $k$, until it possibly finds the exact solution, iterates $\mathbf{x}^{(k)}$ such that:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(0)} + p_{k-1}(A)\,\mathbf{r}^{(0)} \tag{36}$$

With a polynomial $p_{k-1}(A)$ of exact degree $k-1$. For some $k$, $\mathbf{x}^{(k)}$ may not exist or $p_{k-1}(A)$ may have lower degree.

The conjugate gradient method is a Krylov space solver.

# 3 Solving large scale eigenvalue problems

## 3.1 Eigenvalue problems

Eigenvalue problems involve **finding scalar values (eigenvalues) and corresponding vectors (eigenvectors) that satisfy the equation** $A\mathbf{x} = \lambda\mathbf{x}$, where $A$ is a square matrix, $x$ is the eigenvector, and $\lambda$ is the eigenvalue.

Mathematically, the algebraic eigenvalue problem reads as follows. Given a matrix $A \in \mathbb{C}^{n\times n}$, find $(\lambda, \mathbf{v}) \in \mathbb{C} \times \mathbb{C}^n \setminus \{\mathbf{0}\}$ such that:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{37}$$

Where:

- $\lambda$ is an eigenvalue of $A$
- $\mathbf{v}$ (non-zero) is the corresponding eigenvector

Thus, equation 37 represents the **equation that must be satisfied to solve the eigenvalue problem**. Some features:

- The **set of all the eigenvalues** of a matrix $A$ is called the **spectrum** of $A$ and is represented as $\sigma(A)$
- The **maximum modulus of all the eigenvalues** is called the **spectral radius** of $A$:

$$\rho(A) = \max\{|\lambda| \ : \ \lambda \in \lambda(A)\} \tag{38}$$

**√× Mathematical background**

Here is a list of some mathematical concepts that are useful for studying the following chapter.

- The problem (equation 37) $A\mathbf{v} = \lambda\mathbf{v}$ is equivalent to $(A - \lambda I)\mathbf{v} = 0$.
- The equation 37 has a nonzero solution $\mathbf{v}$ if and only if its matrix is singular, that is the eigenvalues of $A$ are the values $\lambda$ such that $\det(A - \lambda I) = 0$.
- The $\det(A - \lambda I) = 0$ is a polynomial of degree $n$ in $\lambda$. It is called the **characteristic polynomial** of $A$ and its roots are the eigenvalues of $A$.
- From the Fundamental Theorem of Algebra, an $n \times n$ matrix $A$ always has $n$ eigenvalues $\lambda_i$, $i = 1, \ldots, n$.
- Each $\lambda_i$ may be real but in general is a complex number.
- The eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ may not all have distinct values.
- Rayleigh quotient: let $(\lambda_i, \mathbf{v}_i)$ be an eigenpair of $A$, then:

$$\lambda_i = \frac{\mathbf{v}_i^H A \mathbf{v}_i}{\mathbf{v}_i^H \mathbf{v}_i}$$

### Similarity transformations to simplify eigenvalue problems

Similarity transformations are crucial in eigenvalue problems because they simplify matrices, making it easier to find eigenvalues. Of course, they don't change the fundamental nature of the original matrix.

**Definition 1: Similar matrices**

The matrix $B$ is **similar** to the matrix $A$ if there exists a nonsingular matrix $T$ such that $B = T^{-1}AT$. Note that a matrix is nonsingular if there exists another matrix $C$ such that $TC = CT = I$.

*Proof.* The above definition is indeed true:

$$\begin{aligned} & B\mathbf{y} = \lambda\mathbf{y} \\ \implies \quad & T^{-1}AT\mathbf{y} = \lambda\mathbf{y} \\ \implies \quad & A\left(T\mathbf{y}\right) = \lambda\left(T\mathbf{y}\right) \end{aligned}$$

So that $A$ and $B$ have the same eigenvalues, and if $\mathbf{y}$ is an eigenvector of $B$, then $\mathbf{v} = T\mathbf{y}$ is an eigenvector of $A$. QED

A square matrix $A$ is called **diagonalizable** if it is similar to a diagonal matrix.

### Similarity transformations limitations

The **similarity transformations preserve only the eigenvalues but not the eigenvectors**. This is not so bad because they can be easily recovered.

Furthermore, the eigenvalue problems using the similarity transformation are simplified when we use diagonal matrices. Unfortunately, **some matrices cannot be transformed into diagonal form by a similarity transformation**.

However, the similarity transformation is only a small tool. In the following pages, we present three powerful methods that attempt to simplify the eigenvalue problem.

## 3.2 Power method

The **Power method** is an iterative technique used to **find the largest eigenvalue** (in absolute value) of a matrix and **its corresponding eigenvector**.

### Algorithm

Assume that the matrix $A$ has a unique eigenvalue $\lambda_1$ of maximum modulus:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$$

With corresponding eigenvector $\mathbf{v}_1$. The algorithm is:

1. **Start with an initial guess**, a nonzero vector $\mathbf{x}^{(0)}$ such that its norm is one $\left|\left|\mathbf{x}^{(0)}\right|\right| = 1$.
2. **Iteration**. For each $k \geq 0$:
   (a) Multiply the current vector by the matrix:
   $$\mathbf{y}^{(k+1)} = A\mathbf{x}^{(k)}$$
   (b) After each multiplication, normalize the vector to prevent it from growing too large:
   $$\mathbf{x}^{(k+1)} = \frac{\mathbf{y}^{(k+1)}}{\left|\left|\mathbf{y}^{(k+1)}\right|\right|}$$
   (c) Computes the Rayleigh quotient. It is computed to approximate the eigenvalue corresponding to the eigenvector $\mathbf{x}^{(k+1)}$. It provides an estimate of the eigenvalue associated with the current eigenvector approximation.
   We can think of it as a checkpoint that tells us how close our current vector is to being an actual eigenvector, and thus how close our estimate is to the actual eigenvalue. This helps us understand the convergence of the iterative process, and ensures that we are on the right track.
   $$\nu^{(k+1)} = \left[\mathbf{x}^{(k+1)}\right]^H A\mathbf{x}^{(k+1)}$$
3. **Repeat until we meet a specific stopping criteria**.

It can be shown that the **iteration scheme converges to a multiple of $\mathbf{v}_1$**, the **eigenvector corresponding to the dominant eigenvalue** $\lambda_1$.

The **convergence rate** of the power method depends on the ratio of the largest absolute eigenvalue $|\lambda_1|$ to the second largest absolute eigenvalue $|\lambda_2|$.

- $\dfrac{\lambda_2}{\lambda_1} \lll 1$, convergence rate high, the method converges **quickly**.
- $\dfrac{\lambda_2}{\lambda_1} \approx 1$, convergence rate low, the method converges **slowly**.

### $ How much does it cost?

It depends on the matrix used:

- **Dense matrix**. Each iteration costs $\approx n^2$ operations,.
- **Sparse matrix**. Each iteration costs only $\approx n$ operations.

### Can it be parallelized?

The power method **can be parallelized to increase its efficiency**, **especially for large matrices**. This is one of the reasons it is used to solve large eigenvalue problems. A simple introduction to parallelization:

- *Matrix-Vector Multiplication.* The main computational task, multiplying the matrix $A$ by the vector $\mathbf{x}$, can be distributed across multiple processors. Each processor handles a portion of the matrix and vector and performs the multiplication in parallel.
- *Normalization.* Vector norming and scaling can also benefit from parallel processing. The norm calculation is a sum of squares that can be computed in parallel.
- *Rayleigh Quotient.* Computing the Rayleigh quotient for eigenvalue approximation can be parallelized similarly to matrix-vector multiplication.

### 3.2.1 Deflation method

**Deflation** is a technique used in conjunction with the Power Method to **find multiple eigenvalues and eigenvectors of a matrix**. This approach helps isolate and find successive eigenvalues by progressively "deflating" the influence of previously found eigenpairs.

#### Mathematical point of view

Suppose we have computed an eigenvalue $\lambda_1$ and corresponding eigenvector $\mathbf{v}_1$ (eigenpair) for a matrix $A$. We can compute additional eigenvalues $\lambda_2, \ldots, \lambda_n$ of $A$ using deflation, which removes the known eigenvalue. The *main idea* is: construct a new matrix $B$ with eigenvalues $\lambda_2, \ldots, \lambda_n$, i.e. deflate the matrix $A$ by removing $\lambda_1$. Then $\lambda_2$ can be obtained by the power method.

Now the interesting question is, how can we compute the new matrix $B$? We help us the similarity transformation. Let $S$ be any nonsingular matrix such that $S\mathbf{v}_1 = \alpha\mathbf{e}_1$, that is $S$ is a scalar multiple of the first column $\mathbf{e}_1$ of the identity matrix $I$. Then, the similarity transformation determined by $S$ transforms $A$ into the form:

$$SAS^{-1} = \begin{bmatrix} \lambda_1 & b^T \\ 0 & B \end{bmatrix} \tag{39}$$

We use $B$ to compute next eigenvalue $\lambda_2$ and eigenvector $\mathbf{z}_2$. Given $\mathbf{z}_2$ eigenvector of $B$, we want to compute the second eigenvector $\mathbf{v}_2$ of the matrix $A$. We need to add an element to vector $\mathbf{z}_2$ (that consist of $n-1$ elements), that is

$$\mathbf{v}_2 = S^{-1}\begin{pmatrix} \alpha \\ \mathbf{z}_2 \end{pmatrix} \qquad \alpha = \frac{\mathbf{b}^H \mathbf{z}_2}{\lambda_1 - \lambda_2}$$

Hence, $\mathbf{v}_2$ is an eigenvector corresponding to $\lambda_2$ for the original matrix $A$. The process can be repeated to find additional eigenvalues and eigenvectors.

#### Algorithm

1. **Find the Dominant Eigenvalue**. We use the Power Method to find the largest eigenvalue $\lambda_1$ and its corresponding eigenvector $\mathbf{v}_1$.
2. ***Deflate* the Matrix**. We modify the matrix to *remove* the influence of the found eigenvalue and eigenvector.
3. **Repeat**. Apply the Power Method to the deflated matrix to find the next largest eigenvalue.

## 3.3 Inverse power method

The **Inverse Power method** is used to **find the smallest eigenvalues of a matrix**, rather than the largest as its brother the Power Method does.

### Algorithm

We use the fact that the eigenvalues of $A^{-1}$ are the reciprocals of those of $A$. Hence the **smallest eigenvalue of $A$ is the reciprocal of the largest eigenvalue of $A^{-1}$**.

1. **Start with an initial guess**, nonzero vector $\mathbf{q}^{(0)}$ such that its norm is one $\left|\left|\mathbf{q}^{(0)}\right|\right| = 1$.
2. **Iteration**. For each $k \geq 0$:
   (a) Solve the system:
   $$A\mathbf{z}^{(k+1)} = \mathbf{q}^{(k)}$$
   (b) After each system solution, normalize the vector to prevent it from growing too large:
   $$\mathbf{q}^{(k+1)} = \frac{\mathbf{z}^{(k+1)}}{\left|\left|\mathbf{z}^{(k+1)}\right|\right|}$$
   (c) Computes the Rayleigh quotient (see page 38 for more details).
   $$\sigma^{(k+1)} = \left[\mathbf{q}^{(k+1)}\right]^H A\mathbf{q}^{(k+1)}$$
3. **Repeat until we meet a specific stopping criteria**.

### How much does it cost?

It depends on the matrix used:

- **Dense matrix**. Each iteration costs $\approx n^3$ operations.
- **Sparse matrix**. Each iteration costs only $\approx n \cdot m$, where $n$ is the number of rows or columns of the square matrix and $m$ the number of non-zero elements.

### Can it be parallelized?

The overall convergence of the method may be sequential because the result of one iteration is needed to compute the next. Therefore, while some components of the algorithm can be parallelized, the entire method isn't inherently parallel.

### 3.3.1 Inverse power method with shift

The **Inverse Power method with shift** *extends* the standard inverse power method by improving convergence to certain eigenvalues near a chosen shift value $\mu$. This is particularly useful for **finding the eigenvalues closest to a given value**.

#### Algorithm

1. **Start with an initial guess**, nonzero vector $\mathbf{q}^{(0)}$ such that its norm is one $\left|\left|\mathbf{q}^{(0)}\right|\right| = 1$.

   **Choose a shift** $\mu$ close to the desired eigenvalue.

   **Compose shifted matrix**:

   $$M_\mu = A - \mu I \tag{40}$$

2. **Iteration**. For each $k \geq 0$:

   (a) Solve the system:

   $$M_\mu \mathbf{z}^{(k+1)} = \mathbf{q}^{(k)}$$

   (b) After each system solution, normalize the vector to prevent it from growing too large:

   $$\mathbf{q}^{(k+1)} = \frac{\mathbf{z}^{(k+1)}}{\left|\left|\mathbf{z}^{(k+1)}\right|\right|}$$

   (c) Computes the Rayleigh quotient (see page 38 for more details).

   $$\nu^{(k+1)} = \left[\mathbf{q}^{(k+1)}\right]^H A\mathbf{q}^{(k+1)}$$

3. **Repeat until we meet a specific stopping criteria**.

We observe that the eigenvalue $\lambda$ of $A$ which is the closes to $\mu$ is the **minimum eigenvalue** of $M_\mu$.

#### How much does it cost?

It depends on the matrix used, the system to solve ($M_\mu \mathbf{z}^{(k+1)} = \mathbf{q}^{(k)}$) is the main cost:

- **Dense matrix**. Each iteration costs $\approx n^3$ operations.
- **Sparse matrix**. Each iteration costs only $\approx n \cdot m$, where $n$ is the number of rows or columns of the square matrix and $m$ the number of non-zero elements.

#### Can it be parallelized?

The inverse power method with shift can be difficult to parallelize efficiently due to the nature of its iterative steps, but there are parts of the algorithm that can benefit from parallel processing. These include solving the linear system, normalization, and the Rayleigh quotient.

## 3.4 QR Factorization

**QR Factorization** is a method to **decompose a matrix into two simpler matrices**: an *orthogonal* matrix $Q$ and an *upper triangular* matrix $R$. We use this method when we want to **find the eigenvalues and the corresponding eigenvectors of a matrix** $A$.

### Required prerequisites

- The **rank of a matrix is the maximum number of linearly independent rows or columns** in the matrix. Essentially, it tells us the dimension of the vector space spanned by the rows or columns. When we do Gaussian elimination, the number of non-zero rows represents the rank!
- An **orthogonal matrix** is a square matrix $Q$ with the property that its transpose is also its inverse. This means that $Q^TQ = QQ^T = I$, where $I$ is the identity matrix. In simpler terms, the rows and columns of an orthogonal matrix are orthonormal vectors, each row and column is orthogonal to the others, and each has a length of 1 (norm equal to one).
- A **vector is orthogonal to another vector if their dot product is zero**. If this is true, we say that the orthogonal vectors are ***perpendicular*** *to each other.*
- An **orthonormal vector** is a vector that is both orthogonal to other vectors in a set and normalized (meaning it has a unit length of 1, norm equal to one). In a collection of orthonormal vectors, each vector is perpendicular to the others, and each has a length of one.
- The **span of a set of orthonormal vectors** is the set of all possible linear combinations of those vectors. If we have a set of orthonormal vectors $\{v_1, v_2, \ldots, v_k\}$, their span is **every vector that can be written as**:

  $$c_1v_1 + c_2v_2 + \cdots + c_kv_k$$

  Where $c_1, c_2, \ldots, c_k$ are scalar coefficients.

### Mathematical point of view

Find orthonormal vectors $[\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n]$ that span the successive spaces spanned by the columns of $A = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n]$:

$$< \mathbf{a}_1 > \subseteq < \mathbf{a}_1, \mathbf{a}_2 > \ldots \subseteq < \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n >$$

This means that (for full rank $A$):

$$< \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_j > = < \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j > \qquad \forall j = 1, \ldots, n$$

A matrix of the previous form will appear:

$$[\mathbf{a}_1 \mid \mathbf{a}_2 \mid \cdots \mid \mathbf{a}_n] = [\mathbf{q}_1 \mid \mathbf{q}_2 \mid \cdots \mid \mathbf{q}_n] \cdot \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & \vdots \\ 0 & 0 & \ddots & r_{nn} \end{bmatrix}$$

That is:

$$A = \widehat{Q}\widehat{R}$$

This is called the **reduced QR factorization**.

Let $A$ be an $m \times n$ matrix. The **full QR factorization** of $A$ is the factorization $A = QR$, where:

- $Q$ is $m \times m$ orthogonal $QQ^T = I$
- $R$ is $m \times n$ upper-trapezoidal

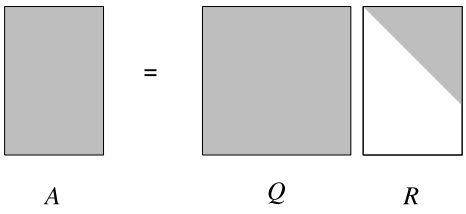


Figure 5: Full QR Factorization.

Let $A$ be an $m \times n$ matrix. The **reduced QR factorization** of $A$ is the factorization $A = \widehat{Q}\widehat{R}$, where:

- $\widehat{Q}$ is $m \times m$
- $\widehat{R}$ is $m \times n$ upper-trapezoidal

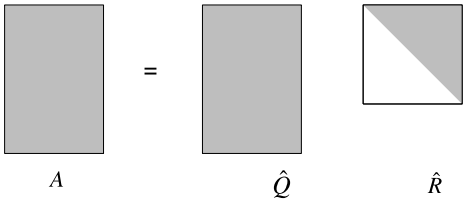


Figure 6: Reduced QR Factorization.

Every matrix $A \in \mathbb{C}^{m \times n}$ $(m \geq n)$ has a **full QR factorization and a reduced QR factorization**. Also, every $A$ of **full rank has a unique reduced QR factorization** with $r_{jj} > 0$, $j = 1, \ldots, n$.

### What is Gram-Schmidt orthogonalization and why is it important?

After a long mathematical introduction to the full and reduced QR factorization methods, the question is *how can we apply this in practice*? Well, finding a special set of vectors that satisfies some properties cannot be very easy. Fortunately, **Gram-Schmidt orthogonalization** is one of the primary **methods used to find the orthogonal (or orthonormal) vectors** necessary for QR factorization.

The Gram-Schmidt orthogonalization takes as:

- **Input**. A set of vectors (typically the columns of the matrix $A$).
- **Output**. An orthogonal set of vectors, which can then be normalized to form an orthonormal set.

Mathematically, the Gram-Schmidt orthogonalization works as follows. Given the columns of $A$ $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$; find new $\mathbf{q}_j$ (the $j$-th column of $\widehat{Q}$) orthogonal to $\mathbf{q}_1, \ldots, \mathbf{q}_{j-1}$ by subtracting components along previous vectors:

$$\mathbf{w}_j = \mathbf{a}_j - \sum_{k=1}^{j-1} \left(\overline{\mathbf{q}}_k^T \mathbf{a}_j\right) \mathbf{q}_k$$

Normalize to get $\mathbf{q}_j = \dfrac{\mathbf{w}_j}{||\mathbf{w}_j||}$, we then obtain a reduced QR factorization with:

$$r_{ij} = \overline{\mathbf{q}}_i^T \mathbf{a}_j \qquad i \neq j \tag{41}$$

And:

$$r_{jj} = \left|\left| \mathbf{a}_j - \sum_{i=1}^{j-1} r_{ij} \mathbf{q}_i \right|\right|$$

Since the previous equation $r_{ij}$ is numerically unstable because it is too sensitive to rounding errors, the following modification ensures more stability. The previous one is called **Classical Gram-Schmidt (CGS, or simply GS)**, and the following one is called **Classical Gram-Schmidt (CGS, or simply GS)**:

$$r_{ij} = \overline{\mathbf{q}}_i^T \mathbf{w}_j \tag{42}$$

### 3.4.1 Schur decomposition applied to QR algorithm

Instead of analyzing the classical QR algorithm, which is very general and applicable to any mathematical problem, here we present the powerful **Schur decomposition**, which is applied with the aim of finding a QR decomposition.

**Why do we need a variant of the QR decomposition algorithm?**

Before presenting and explaining how to apply it, we think that the motivations are fundamental:

- *What is the purpose of using the QR algorithm with the Schur variant?* To transform a matrix into an upper triangular form with eigenvalues on the diagonal.
- *And why should this be useful? We could get the same result using the theoretical QR decomposition (e.g. Gram-Schmidt).* Obviously, but the Schur decomposition provides **more numerical stability**. In addition, it is very useful for analyzing eigenvalues and eigenvectors, and it simplifies the computation of matrix functions.
- *So the Schur decomposition is the best! We will only use that.* Not at all. After explaining the algorithm, we will see why there are other better alternatives.

**Required prerequisites**

- **Schur decomposition** is a mathematical concept used to transform a square matrix into a quasi-upper triangular form. If $A \in \mathbb{C}^{n\times n}$ then there is a unitary matrix $U \in \mathbb{C}^{n\times n}$ such that:

  $$U^H A U = T$$

  And $U$ is upper triangular. The diagonal elements of $T$ are the eigenvalues of $A$. The Schur vectors are $U = |\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n|$ and they are in general not eigenvectors.

- The $k$-th column of $U^H A U = T$ read:

  $$A\mathbf{u}_k = \lambda_k \mathbf{u}_k + \sum_{i=1}^{k-1} t_{ik}\mathbf{u}_i$$

  That is:

  $$A\mathbf{u}_k \in \text{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_k\} \qquad \forall k$$

  The first **Schur vector** $\mathbf{u}_1$ is an eigenvector of $A$. The first $k$ Schur vectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ form an invariant subspace for $A$. The Schur decomposition is not unique.

### Algorithm

**Goal**: let $A \in \mathbb{C}^{n \times n}$, the QR algorithm computes an upper triangular matrix $T$ and a unitary matrix $U$ such that $A = UTU^H$ is the Schur decomposition of $A$.

1. **Initialization**. $A$ is the *original matrix* we start with; at the beginning, the initial guess $A^{(0)}$ is equal to the original $A^{(0)} = A$. It is transformed iteratively by the QR decompositions and updates. Meanwhile, $U$ is the *accumulation of orthogonal transformations* applied to $A$. Initially, $U$ is set to the identity matrix $U^{(0)} = I$.

$$\begin{aligned} A^{(0)} &= A \\ U^{(0)} &= I \end{aligned}$$

2. **Iteration**. For each $k \geq 1$:

   (a) **QR Decomposition**. Decompose the matrix $A^{(k-1)}$ into the product of an orthogonal matrix $Q^{(k)}$ and an upper triangular matrix $R^{(k)}$:

$$A^{(k-1)} = Q^{(k)} R^{(k)}$$

   (b) **Update the matrix** $A$ to be used in next iteration by multiplying $R^{(k)}$ and $Q^{(k)}$:

$$A^{(k)} = R^{(k)} Q^{(k)}$$

   (c) **Update the Transformations matrix** $U$ to keep track of the cumulative orthogonal transformations:

$$U^{(k)} = U^{(k-1)} Q^{(k)}$$

3. **Repeat until we meet a specific stopping criteria**.

4. **Results**. If a certain stopping criterion is met, we have the upper triangular matrix $A^{(k)}$ and the orthogonal matrix $U^{(k)}$. The Schur decomposition gives us an important result:

$$T = A^{(k)} \wedge U^{(k)} = U \implies A = UTU^H \equiv U^H AU = T$$

   In other words, in the end we get:

   - The **unitary matrix** $U$ ($U^H U = I$), where the **columns are the orthonormal eigenvectors** of the original matrix $A$.
   - The **upper triangular matrix** $T$, where the elements of the **diagonal are the eigenvalues** of the original matrix $A$.

About the convergence, we need to show some interesting details. Let us assume that all the eigenvalues are isolated:

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

Then the elements of $A^{(k)}$ below the diagonal converge to zero:

$$\lim_{k \to \infty} a_{ij}^{(k)} = 0 \qquad \forall i > j$$

Moreover, it can be shown that:

$$a_{ij}^{(k)} = O\left(\left|\frac{\lambda_i}{\lambda_i}\right|^k\right) \qquad i > j$$

Thus, **convergence is low when the eigenvalues are close**.

### $ How much does it cost?

The QR algorithm enhanced with Schur decomposition is powerful for finding eigenvalues and eigenvectors, but the **high iteration cost** of $\approx n^3$ operations is a tradeoff for its robustness and accuracy.

### Can it be parallelized?

The Schur decomposition applied to the QR algorithm is **difficult** to parallelize due to its sequential dependencies.

### 3.4.2 Hessenberg applied to QR algorithm

A matrix $H \in \mathbb{C}^{n\times n}$ is called a **Hessenberg matrix** if its elements below the lower off-diagonal are zero:

$$h_{ij} = 0 \qquad i > j+1$$

For example:

$$H = \begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{bmatrix}$$

**Why do we use Hessenberg?**

Apply the QR method to a Hessenberg matrix can be decrease the number of operations from $n^3$ (Schur decomposition, page 46) to $n^2$ **operations**.

**Algorithm**

**Goal**: compute a Hessenberg matrix $H$ and an orthogonal matrix $U$ such that $A = UHU^H$ is the QR decomposition of $A$. Such a reduction can be done with a finite number of operations.

1. **Initial Transformation to Hessenberg Form**. Take as input the matrix $A$, we convert $A$ to a Hessenberg matrix $H$ using similarity transformations techniques.

2. **Initial guess and initial accumulation of orthogonal transformations**. The first guess is the first Hessenberg form we got from the previous step, and for the $U^{(0)}$ we take the identity as always:

$$\begin{aligned} H^{(0)} &= H \\ U^{(0)} &= I \end{aligned}$$

3. **Iteration**. For each $k \geq 1$:

   (a) **Hessenberg QR Decomposition**. Decompose the matrix $H^{(k-1)}$ into the product of an orthogonal matrix $Q^{(k)}$ and an upper triangular matrix $R^{(k)}$:

$$H^{(k-1)} = Q^{(k)}R^{(k)}$$

   (b) **Update the Hessenberg matrix** $H$ to be used in next iteration by multiplying $R^{(k)}$ and $Q^{(k)}$:

$$H^{(k)} = R^{(k)}Q^{(k)}$$

   (c) **Update the Transformations matrix** $U$ to keep track of the cumulative orthogonal transformations:

$$U^{(k)} = U^{(k-1)}Q^{(k)}$$

4. **Repeat until we meet a specific stopping criteria**.

5. **Results**. If a certain stopping criterion is met, we have the upper triangular matrix $H^{(k)}$ and the orthogonal matrix $U^{(k)}$. The Schur decomposition using the Hessenberg matrix gives us an important result:

$$H = H^{(k)} \wedge U^{(k)} = U \implies A = UHU^H \equiv U^H AU = H$$

In other words, in the end we get:

- The **unitary matrix** $U$ ($U^H U = I$), where the **columns are the orthonormal eigenvectors** of the original matrix $A$.
- The **upper triangular matrix** $H$, where the elements of the **diagonal are the eigenvalues** of the original matrix $A$.

### $ How much does it cost?

As we have already said, the Hessenberg matrix **reduces the computational cost to** $n^2$, which is more competitive than the Schur decomposition ($n^3$).

### Can it be parallelized?

As we have seen with the other QR methods, parallelization is still **difficult**. It can be achieved with some very optimized libraries, but in general it is complicated due to its dependencies.

## 3.5 Lanczos method

The **Lanczos algorithm** is an iterative method for **finding the eigenvalues and eigenvectors** of a large, sparse, symmetric (or Hermitian) matrix. It's particularly **useful for computing the extremal (largest or smallest) eigenvalues and their corresponding eigenvectors**. The algorithm generates a sequence of vectors, called *Lanczos vectors*, which are used to form a tridiagonal matrix that approximates the original matrix. Finally, this method is also used to **find a low-rank approximation** of the input matrix; by low-rank, we mean a **technique** used in numerical linear algebra to simplify a matrix while preserving its most important properties. It is particularly useful **for reducing the complexity of large data sets, compressing information, and speeding up computations**.

### ✔ Good prerequisites of the matrix

Some good prerequisites necessary to get the best performance with the Lanczos algorithm are:

- Sparse matrix;
- Symmetric (or Hermitian) matrix;
- Square matrix, then a size of $n \times n$.

### √× Mathematical point of view

Let a symmetric matrix $A$ of size $n \times n$, the Lanczos algorithm is based on computing the following decomposition of $A$:

$$A = QTQ^T \tag{43}$$

Where $Q$ is an orthonormal basis of vectors $\mathbf{q}_1, \ldots, \mathbf{q}_n$ and $T$ is tri-diagonal:

$$Q = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n] \qquad T = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \beta_{n-1} \\ 0 & \cdots & 0 & \beta_{n-1} & \alpha_n \end{bmatrix}$$

The **decomposition always exists** and is **unique** if $\mathbf{q}_1$ was specified.

Since we know that $T = Q^T AQ$ which gives:

$$\alpha_k = \mathbf{q}_k^T A \mathbf{q}_k \qquad \beta_k = \mathbf{q}_{k+1}^T A \mathbf{q}_k$$

The full decomposition is obtained by imposing $AQ = QT$:

$$[A\mathbf{q}_1, A\mathbf{q}_2, \ldots, A\mathbf{q}_n] =$$
$$[\underbrace{\alpha_1\mathbf{q}_1 + \beta_1\mathbf{q}_2}_{\text{1st row}}, \underbrace{\beta_1\mathbf{q}_1 + \alpha_2\mathbf{q}_2 + \beta_2\mathbf{q}_3}_{\text{2nd row}}, \ldots, \underbrace{\beta_{n-1}\mathbf{q}_{n-1} + \alpha_n\mathbf{q}_n}_{n \text{ row}}]$$

### Algorithm

Note that at iteration $k$, the algorithm generates intermediate matrices $Q_k$ and $T_k$ satisfying $T_k = Q_k^T A Q_k$.

1. **Residual, Lanczos vector and scalar initialization**. We set the residual to the value of the lanczos vector $\mathbf{q}_1$ which is set randomly; the Lanczos vector is set to zero and finally the scalar $\beta$ is set to one.

$$\mathbf{r}_0 = \mathbf{q}_1 \qquad \mathbf{q}_0 = \mathbf{0} \qquad \beta = 1$$

2. **Iteration**. For each $k = 1, \ldots, n$:

   (a) **Check if the previously calculated $\beta$ is zero**. If zero, stop the algorithm, otherwise continue the iteration.

   (b) **Compute Lanczos vector $\mathbf{q}_k$**:

$$\mathbf{q}_k = \frac{\mathbf{r}_{k-1}}{\beta_{k-1}}$$

   (c) **Compute scalar $\alpha_k$**:

$$\alpha_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k$$

   (d) **Compute the residual $\mathbf{r}_k$**:

$$\mathbf{r}_k = (\mathbf{A} - \alpha_k)\, \mathbf{q}_k - \beta_{k-1} \mathbf{q}_{k-1}$$

   (e) **Compute scalar $\beta_k$**:

$$\beta_k = |\mathbf{r}_k|$$

3. **Results**. It produces the tridiagonal symmetric matrix $T$ that is an approximation of the original matrix $A$ and the orthonormal basis $Q_k$.

At iteration $k$, the $k$-th Lanczos vector $\mathbf{q}_k$ is proven to maximize the left hand side of:

$$\max_{\mathbf{y} \neq \mathbf{0}} \frac{\mathbf{y}^T \left(Q_k^T A Q_k\right) \mathbf{y}}{\mathbf{y}^T \mathbf{y}} = \lambda_1 \left(T_k\right) \leq \lambda_1 \left(A\right) = \lambda_1 \left(T\right)$$

And to simultaneously minimize the left hand side of:

$$\min_{\mathbf{y} \neq \mathbf{0}} \frac{\mathbf{y}^T \left(Q_k^T A Q_k\right) \mathbf{y}}{\mathbf{y}^T \mathbf{y}} = \lambda_n \left(T_k\right) \leq \lambda_n \left(A\right) = \lambda_n \left(T\right)$$

Where:

- $\lambda_1 \left(A\right)$ is the ***maximum*** **eigenvalue** of $A$;
- $\lambda_n \left(A\right)$ is the ***minimum*** **eigenvalue** of $A$.

### $ How much does it cost?

Although the algorithm is quite complex to understand, the computational cost is very competitive. If we respect all the prerequisites that we have said, then for **large**, **symmetric**, **sparse** and **square matrices**, the primary cost is proportional to the **number of non-zero elements** in the matrix. Thus, the cost of each iteration is only $\approx \text{nnz}(A)$ **operations** (where $A$ is the input matrix).

The reasoning changes for **dense matrices**, although still feasible, the cost can be higher due to the $\approx n^2$ **operations**.

### Can it be parallelized?

The Lanczos method is widely used in practice, and obviously it **fits very well with parallel patterns**. The Lanczos parallelization focuses on matrix-vector multiplication and orthogonalization steps. If the reader wants to delve deeper into this parallelization, we suggest an interesting scientific paper:

Parallelization of the Lanczos Algorithm on Multi-core Platforms

Link to the paper

# References

[1] Antonietti Paola Francesca. Numerical Linear Algebra. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

# Index

## S



## T



## U