Numerical Linear Algebra - Notes - v1.2.0

260236

January 2025

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.

Correlated Projects

During the Numerical Linear Algebra for HPC course, I was part of a team where we created a project that included two challenges related to the course. See more details in the corresponding repository:

GitHub repository

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**:

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

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

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

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

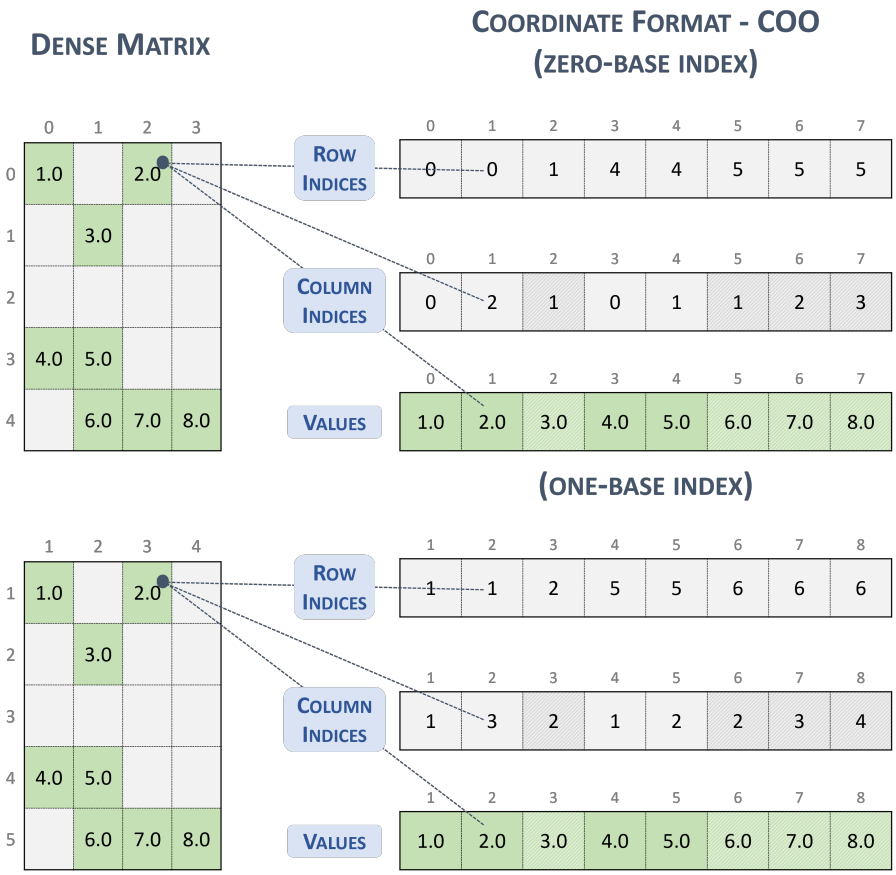


Figure 1: Graphical representation of the coordinate format (COO) technique. From the figure we can see the representation of the `AA` array, called *values*, the `JR`, called *row indices*, and finally the `JC`, called *column indices*. The algorithm is very simple. The figures are taken from the NVIDIA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

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

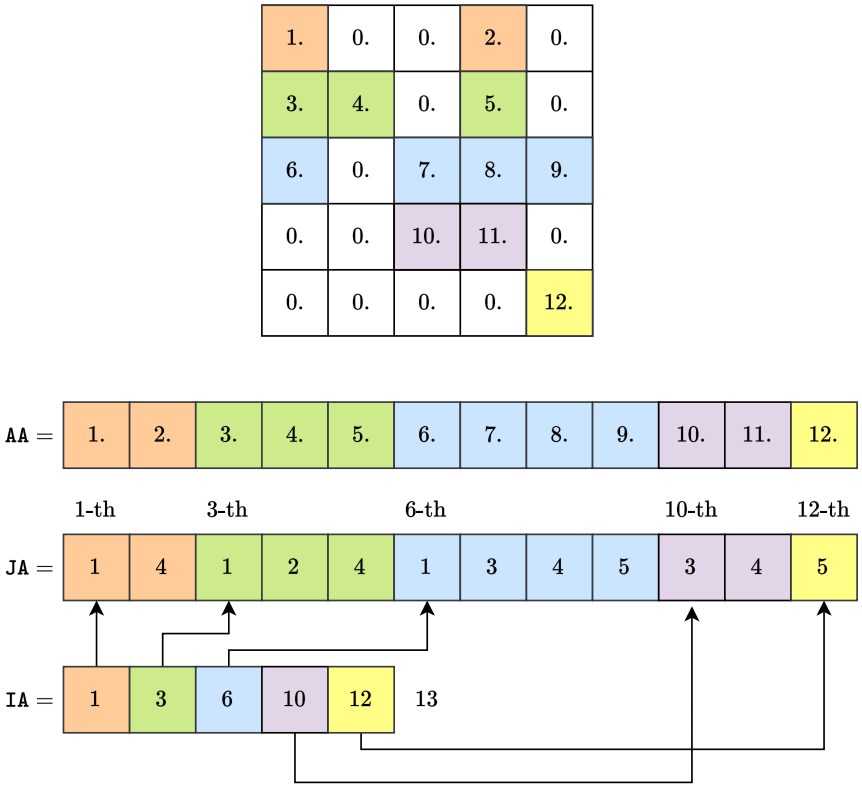


Figure 2: View an illustration of the CRS technique using colors to improve readability.

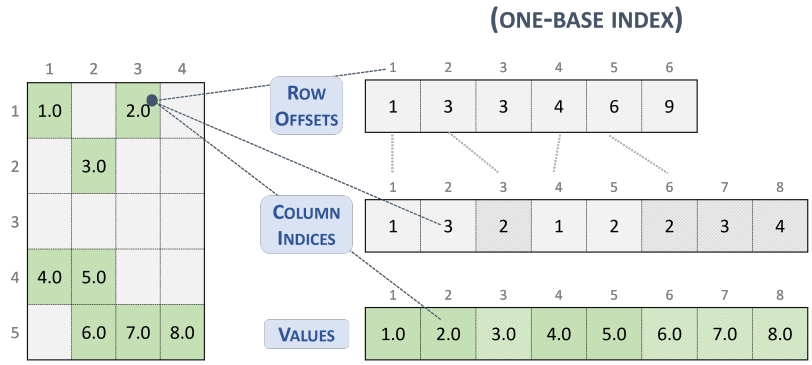


Figure 3: Graphical representation of the coordinate compressed sparse row (CSR) technique. From the figure we can see the representation of the `AA` array, called *values*, the `IA`, called *row offset*, and finally the `JA`, called *column indices*. It's interesting to see how the empty line case is 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 ∞ 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 20.

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 α is a weighted version of the residual.

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

We now ask ourselves **which value of the parameter** α, 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 α to choose, we plot the problem. On the *x-axis* are the values of α 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 α. In **green** we see the **spectral radius equal to** $\rho(B_\alpha)$; it is important because its intersection with the upper bound of ρ represents the right bound of the interval where the **values of** α **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** α **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:

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

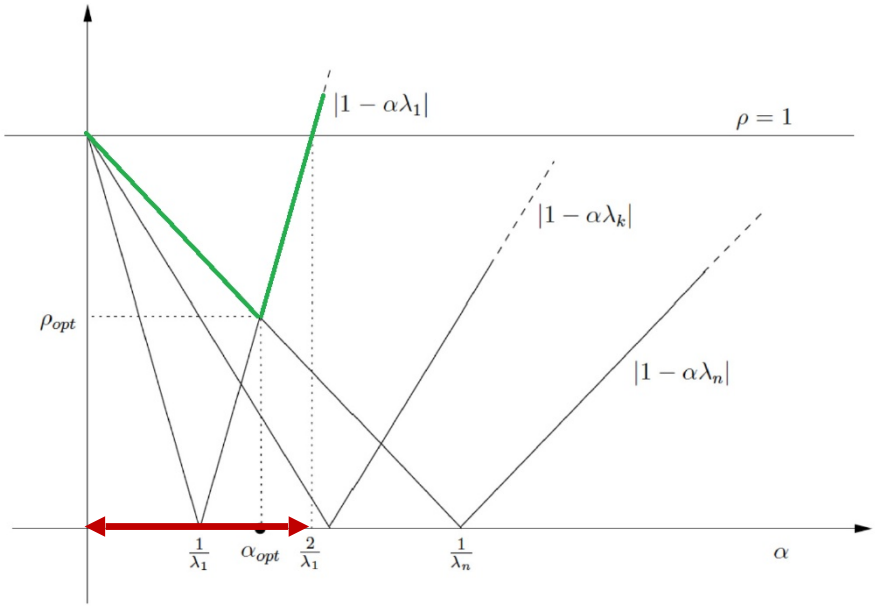


Figure 4: Graphical representation of the optimal alpha to choose in the stationary Richardson method.

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 α 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** α.
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:

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

Where ε 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:

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

  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 $P^{-1}Ax = P^{-1}b$:

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

- **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:

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

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 ρ_{opt} (equation 11, page 26) 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 24 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 29 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 α_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 α_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 β_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 α_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 β_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 kth 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 kth 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.

Solving nonsymmetric linear systems iteratively with Krylov space solvers is considerably more difficult and costly than symmetric systems. There are two different ways to generalize the Conjugate Gradient:

- Maintain the orthogonality of the projection and the related minimality of the error by constructing either orthogonal residuals $\mathbf{x}^{(k)}$. Then, the recursions involve all previously constructed residuals or search directions and all previously constructed iterates.
- (Preferred) Maintain short recurrence formulas for residuals, direction vectors and iterates (BiConjugate Gradient (BiCG) method, Lanczos-type product methods (LTPM)). The resulting methods are at best oblique projection methods. There is no minimality property of error or residuals vectors.

2.7.1 BiConjugate Gradient (BiCG) and BiCGSTAB method

The **BiConjugate Gradient (BiCG) method** is an iterative algorithm used to **solve non-symmetric linear systems of equations**, $A\mathbf{x} = \mathbf{b}$. It **extends the Conjugate Gradient (CG) method to handle matrices that are not symmetric or positive definite**.

BiCG has the peculiarity of **simultaneously solving** the original system $A\mathbf{x} = \mathbf{b}$ (where A is a square matrix and $\mathbf{x}, \mathbf{b}$ are column vectors) and a dual system $\widehat{\mathbf{x}}A^T = \widehat{\mathbf{b}}$ (where the $A^T \neq A$ and $\widehat{\mathbf{x}}, \widehat{\mathbf{b}}$ are row vectors).

While CG has mutually orthogonal residual $\mathbf{r}^{(k)}$, BiCG constructs in the same spaces residuals that are orthogonal to a dual Krylov space spanned by "shadow residuals":

$$\begin{aligned} \tilde{\mathbf{r}}^{(k)} &= p_k\left(A^T\right)\tilde{\mathbf{r}}^{(0)} \in \operatorname{span}\left\{\tilde{\mathbf{r}}^{(0)}, A^T\tilde{\mathbf{r}}^{(0)}, \ldots, \left(A^T\right)^k\tilde{\mathbf{r}}^{(0)}\right\} \\ &= \mathcal{K}_{k+1}\left(A^T, \tilde{\mathbf{r}}^{(0)}\right) \equiv \tilde{\mathcal{K}}_{k+1} \end{aligned} \tag{37}$$

The initial shadow residual $\tilde{\mathbf{r}}^{(0)}$ can be chosen freely. So, BiCG requires two matrix-vector multiplications to extend $\mathcal{K}_k$ and $\tilde{\mathcal{K}}_k$: one multiplication by A (the original system) and one by A^T (the dual system).

BiCG Algorithm

1. **Initial guess**. Start with an initial guess $\mathbf{x}^{(0)}$ (column vector), $\widehat{\mathbf{x}}^{(0)}, \widehat{\mathbf{b}}$ (row vectors).
2. **Compute initial residual**. Define the residual $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$ (column vector) and the shadow residual $\widehat{\mathbf{r}}^{(0)} = \widehat{\mathbf{b}} - \widehat{\mathbf{x}}^{(0)}A^T$ (row vector).
3. **Initial direction**. The direction is equal to the residual $\mathbf{d}_0 = \mathbf{r}^{(0)}$ (column vector), and the shadow direction is equal to the shadow residual $\widehat{\mathbf{d}}_0 = \widehat{\mathbf{r}}^{(0)}$ (row vector).
4. **Iteration**. Continue to iterate until the stopping criteria is met:
   (a) Parameter α_k:
   $$\alpha_k = \frac{\widehat{\mathbf{r}}^{(k)}\mathbf{r}^{(k)}}{\widehat{\mathbf{d}}_k A\mathbf{d}_k}$$
   (b) Update the solution for both systems:
   $$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k\mathbf{d}_k \\ \widehat{\mathbf{x}}^{(k+1)} &= \widehat{\mathbf{x}}^{(k)} + \alpha_k\widehat{\mathbf{d}}_k \end{aligned}$$
   (c) Update the residual for both systems:
   $$\begin{aligned} \mathbf{r}^{(k+1)}\left(\equiv \mathbf{b} - A\mathbf{x}^{(k+1)}\right) &= \mathbf{r}^{(k)} + \alpha_k A\mathbf{d}_k \\ \widehat{\mathbf{r}}^{(k+1)}\left(\equiv \widehat{\mathbf{b}} - \widehat{\mathbf{x}}^{(k+1)}A^T\right) &= \widehat{\mathbf{r}}^{(k)} - \alpha_k\widehat{\mathbf{d}}_k A^T \end{aligned}$$

(d) Parameter β_k:

$$\alpha_k = \frac{\widehat{\mathbf{r}}^{(k+1)}\mathbf{r}^{(k+1)}}{\widehat{\mathbf{r}}^{(k)}\mathbf{r}^{(k)}}$$

(e) Update the direction:

$$\begin{aligned}\mathbf{d}_{k+1} &= \mathbf{r}^{(k+1)} + \beta_k + \mathbf{d}_k \\ \widehat{\mathbf{d}}_{k+1} &= \widehat{\mathbf{r}}^{(k+1)} + \beta_k + \widehat{\mathbf{d}}_k\end{aligned}$$

In practice the $\widehat{\mathbf{x}}^{(0)} = \left[\mathbf{x}^{(0)}\right]^T$ and $\widehat{\mathbf{b}} = \mathbf{b}^T$. We also need to make sure that $\widehat{\mathbf{r}}^{(0)}\mathbf{r}^{(0)} \neq 0$.

$ How much does it cost and why do we need to use BiCGSTAB?

Each iteration costs twice as much as a CG iteration:

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

It also has a **big problem**: **numerical stability**. BiCG uses duality, which introduces a level of complexity that can lead to numerical instability, especially because of the **multiplication of** A and A^T. Fortunately, the **BiConjugate Gradient Stabilized (BiCGSTAB) method** is a variant of BiCG and has **faster and smoother convergence than the original BiCG**. The main idea in BiCGSTAB is not to keep track of residuals and search directions, but to use techniques to stabilize the convergence and improve the robustness of the method.

Can it be parallelized?

BiCGSTAB can be **implemented on GPUs using frameworks like CUDA**. This allows for massive parallelism, as GPUs have thousands of cores that can perform computations simultaneously. BiCGSTAB can also be **parallelized on distributed memory systems using MPI** (Message Passing Interface). This involves partitioning the matrix and distributing the computations across multiple processors. The communication between processors is managed efficiently to minimize overhead and maximize performance.

2.7.2 Generalized Minimum Residual (GMRES) method

The **Generalized Minimum Residual (GMRES) method** is an iterative technique used to **solve non-symmetric linear systems** of the form $A\mathbf{x} = \mathbf{b}$. It is particularly **effective** for systems where A is **non-symmetric** or even **non-square**.

This method is a projection method. It approximates the solution by the vector in a Krylov subspace with minimal residual. It uses the Arnoldi process to generate an orthonormal basis for the Krylov subspace. This process involves a modified Gram-Schmidt orthogonalization to ensure the basis vectors are orthogonal. The main idea is that approximates the exact solution of $A\mathbf{x} = \mathbf{b}$ by the vector:

$$\mathbf{x}^{(k)} \in \mathbf{x}^{(0)} + \mathcal{K}_k\left(A, \mathbf{r}^{(0)}\right) \tag{38}$$

That minimizes the Euclidean norm of the residual $\mathbf{r}^{(k)}$.

GMRES Algorithm

1. **Initialization**. Choose an initial guess $\mathbf{x}^{(0)}$ and the initial residual $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$.
2. **Initialize orthonormal vector**. Set $\mathbf{q}_1 = \dfrac{\mathbf{r}^{(0)}}{\left|\left|\mathbf{r}^{(0)}\right|\right|_2}$.
3. **Iteration**. Continue to iterate until the stopping criteria is met:
   (a) Compute the orthonormal k vector $\mathbf{q}_k$ with a suitable method.
   (b) Form $\mathbf{Q}_k$ as the $n \times k$ matrix formed by $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k$.
   (c) Find $\mathbf{y}^{(k)}$ which minimize $\left|\left|\mathbf{r}^{(k)}\right|\right|_2$.
   (d) Compute the result $\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + Q_k\mathbf{y}^{(k)}$.

About the **convergence**:

- If $A_S = \dfrac{\left(A + A^T\right)}{2}$ is SPD, then:

$$\left|\left|\mathbf{r}^{(k)}\right|\right|_2 \leq \left[1 - \frac{\lambda^2_{\min}\left(A_S\right)}{\lambda_{\max}\left(A^T A\right)}\right]^{\frac{k}{2}} \left|\left|\mathbf{r}^{(0)}\right|\right|_2 \tag{39}$$

- If A is SPD, then:

$$\left|\left|\mathbf{r}^{(k)}\right|\right|_2 \leq \left[\frac{\left[K_2\left(A\right)\right]^2 - 1}{\left[K_2\left(A\right)\right]^2}\right]^{\frac{k}{2}} \left|\left|\mathbf{r}^{(0)}\right|\right|_2 \tag{40}$$

$ 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**.

In addition to the matrix-vector product, $k \cdot n$ **operations** must be computed at the k-th iteration. Furthermore, the k-th iterate minimize the residual in the Krylov subspace $\mathcal{K}_k\left(A, \mathbf{r}^{(0)}\right)$. In exact arithmetic, since every subspace is contained in the next subspace, the residual does not increase. Therefore, after $n = \text{size}\,(A)$ iterations, the Krylov space $\mathcal{K}_n\left(A, \mathbf{r}^{(0)}\right)$ is the whole of $\mathbb{R}^n$, hence the GMRES method has **finite termination property**. This, unfortunately, does not happen in practice.

Can it be parallelized?

GMRES **can be parallelized** on multi-core and many-core architectures, such as CPUs and GPUs.

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 λ 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:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{41}$$

Where:

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

Thus, equation 41 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:

$$\rho(A) = \max\{|\lambda| \ : \ \lambda \in \lambda(A)\} \tag{42}$$

Mathematical background

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

- The problem (equation 41) $A\mathbf{v} = \lambda\mathbf{v}$ is equivalent to $(A - \lambda I)\mathbf{v} = 0$.
- The equation 41 has a nonzero solution $\mathbf{v}$ if and only if its matrix is singular, that is the eigenvalues of A are the values λ such that $\det(A - \lambda I) = 0$.
- The $\det(A - \lambda I) = 0$ is a polynomial of degree n in λ. 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 λ_i, $i = 1, \ldots, n$.
- Each λ_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 λ_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** λ_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 $|\lambda_2|$.

- $\frac{|\lambda_2|}{|\lambda_1|} \ll 1$, convergence rate high, the method converges **quickly**.
- $\frac{|\lambda_2|}{|\lambda_1|} \approx 1$, 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 λ_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 λ_1. Then λ_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:

$$SAS^{-1} = \begin{bmatrix} \lambda_1 & b^T \\ 0 & B \end{bmatrix} \tag{43}$$

We use B to compute next eigenvalue λ_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 λ_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 λ_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 43 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 μ. 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 μ close to the desired eigenvalue.

   Compose shifted matrix:

   $$M_\mu = A - \mu I \tag{44}$$

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 43 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 λ of A which is the closes to μ is the **minimum eigenvalue** of M_μ.

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 $\{v_1, v_2, \dots, v_k\}$, their span is **every vector that can be written as**:

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

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

Mathematical point of view

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

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

This means that (for full rank A):

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

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

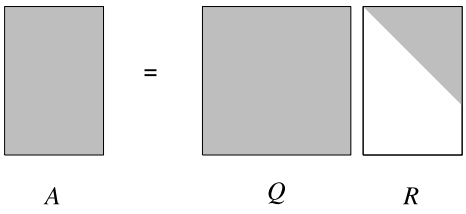


Figure 5: Full QR Factorization.

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

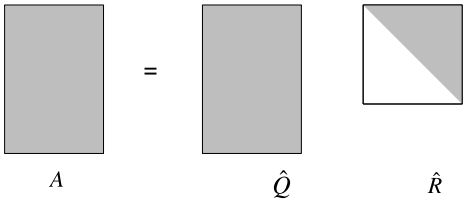


Figure 6: Reduced QR Factorization.

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:

$$r_{ij} = \overline{\mathbf{q}}_i^T \mathbf{a}_j \qquad i \neq j \tag{45}$$

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 (45) is called **Classical Gram-Schmidt (CGS)**, and the following one is called **Modified Gram-Schmidt (MGS)**:

$$r_{ij} = \overline{\mathbf{q}}_i^T \mathbf{w}_j \tag{46}$$

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 51) 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:

$$A = QTQ^T \tag{47}$$

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 β 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 β 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 α_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 β_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

4 Numerical methods for overdetermined linear systems and SVD

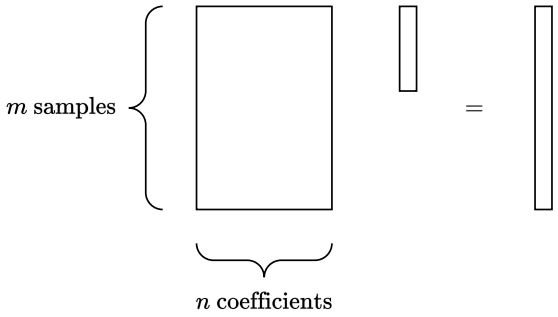
4.1 Overdetermined systems and Least Squares

An **Overdetermined linear system** is a **system of linear equations in which there are more equations than unknowns**. In other words, there are more constraints than variables, which often makes it *impossible to satisfy all the equations simultaneously*. This often happens in practical applications where we have more measurements or constraints than variables.

The solution method is **Least Squares**; it finds an approximate **solution by minimizing the sum of the squares of the residuals** (the differences between the left and right sides of the equations). A practical implementation is Singular Value Decomposition (SVD).

√× Mathematical point of view

The mathematical problem reads: given $A \in \mathbb{R}^{m \times n}$, with $m \geq n$ and $\mathbf{b} \in \mathbb{R}^m$, find $\mathbf{x} \in \mathbb{R}^n$ such that: $A\mathbf{x} = \mathbf{b}$.



Note that the above problems generally have no solution unless the right side $\mathbf{b}$ is an element of range (A) (all possible linear combinations of the columns of A). So the basic approach is to look for a $\mathbf{x}$ that makes $A\mathbf{x}$ "close" to $\mathbf{b}$.

We compute the solution using least-squares. Given $A \in \mathbb{R}^{m \times n}$, $m \geq n$, we say that $\mathbf{x}^* \in \mathbb{R}^n$ is a solution of the linear system $A\mathbf{x} = \mathbf{b}$ in the least-squares sense if:

$$\Phi(\mathbf{x}^*) = \min_{\mathbf{y} \in \mathbb{R}^n} \Phi(\mathbf{y}) \tag{48}$$

Where:

$$\Phi(\mathbf{y}) = ||A\mathbf{y} - \mathbf{b}||_2^2 \tag{49}$$

The problem thus consists of minimizing the Euclidean norm of the residual. The **solution $\mathbf{x}^*$** can be **found by imposing the condition that the gradient of the function $\Phi(\cdot)$ must be equal to zero at $\mathbf{x}^*$**. From the definition we have:

$$\begin{aligned} \Phi(\mathbf{y}) &= (A\mathbf{y} - \mathbf{b})^T (A\mathbf{y} - \mathbf{b}) \\ &= \mathbf{y}^T A^T A\mathbf{y} - 2\mathbf{y}^T A\mathbf{b} + \mathbf{b}^T \mathbf{b} \end{aligned}$$

Therefore:

$$\nabla\Phi(\mathbf{y}) = 2A^TA\mathbf{y} - 2A^T\mathbf{b}$$

From which it follows that $\mathbf{x}^*$ must be the solution of the square system:

$$A^TA\mathbf{x}^* = A^T\mathbf{b} \tag{50}$$

The system of normal equations is nonsingular if A has **full rank** and, in such a case, the least-squares **solution exists and is unique**.

Theorem 8. *Let $A \in \mathbb{R}^{m\times n}$, with $m \geq n$, be a full rank matrix. Then the **unique solution** in the least-square sense $\mathbf{x}^*$ of $A\mathbf{x}^* = \mathbf{b}$ is given by $\mathbf{x}^* = \widehat{R}^{-1}\widehat{Q}^T\mathbf{b}$, where $\widehat{R} \in \mathbb{R}^{n\times n}$ and $\widehat{Q} \in \mathbb{R}^{m\times n}$ are the matrices of the reduced QR factorization of A. Moreover, the minimum of $\Phi(\cdot)$ is given by:*

$$\Phi(\mathbf{x}^*) = \sum_{i=n+1}^{m} \left[\left(Q^Tb\right)_i\right]^2$$

If A has full rank, then since the solution exists in the least squares sense and is unique, it must necessarily have **minimal Euclidean norm**:

$$||A\mathbf{x}^* - \mathbf{b}||_2^2 \leq \min_{\mathbf{x}\in\mathbb{R}^n} ||A\mathbf{x} - \mathbf{b}||_2^2 \tag{51}$$

In other words, given an overdetermined system $A\mathbf{x} = \mathbf{b}$, the least squares method finds $\mathbf{x}$ that minimizes the quantity $||A\mathbf{x} - \mathbf{b}||_2^2$. These problems can be solved using the SVD method.

4.2 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) method is a factorization of a matrix into three other matrices. For any $m \times n$ matrix A, the SVD is given by:

$$A = U\Sigma V^T \tag{52}$$

It **provides a solution to Least Squares techniques**. Where:

- U is an $m \times m$ orthogonal matrix, called **left singular vectors**. These vectors form an *orthonormal basis* for the column space of A.
- Σ is a $m \times n$ diagonal matrix with non-negative real numbers on the diagonal, called **singular values**. These values are sorted in **descending order** (from largest to smallest), and the number of values is guaranteed by the minimum between the number of columns and the number of rows; if A is $m \times n$, there are $\min(m, n)$ singular values.

  These values are important because keeping only the largest singular values can reduce the dimensions of the data while preserving important features. It also compresses the image, if the matrix represents an image, and filters out noise.
- V is an $n \times n$ orthogonal matrix, called **right singular vectors**. These vectors form an orthonormal basis for the row space of A.

Theorem 9. *Let $A \in \mathbb{R}^{m\times n}$. There exist two orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that:*

$$U^T AV = \Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n} \tag{53}$$

With $p = \min(m, n)$ and $\sigma_1 \geq \cdots \geq \sigma_p \geq 0$.

This method is a robust mathematical tool commonly employed in machine learning for tasks such as dimensionality reduction, data compression and feature extraction. It is especially effective in handling high-dimensional datasets, helping to lower computational complexity and enhance the efficiency of machine learning algorithms.

✓ Singular Value Decomposition (SVD) is an **alternative to Eigenvalue Decomposition**, which is **generally better** for rank-deficient and ill-conditioned matrices.

✓ Computing the **SVD is always numerically stable** for any matrix but is typically **more expensive than other decompositions**.

✓ SVD can be used to **compute low-rank approximations** to a matrix via Principal Component Analysis (PCA has many practical applications, and usually large sparse matrices arise).

SVD features

- If A is a real-valued matrix, U and V will also be real-valued and in the equation 53, U^T must be written instead of U^H.
- The singular values holds:

$$\sigma_i(A) = \sqrt{\lambda_i(A^T A)} \qquad i = 1, \ldots, p \tag{54}$$

- Since AA^T and $A^T A$ are symmetric matrices, the columns of U turn out to be the eigenvectors of $A^T A$ and, therefore, they are not uniquely defined. The same holds for the columns of V, which are the right singular vectors of A.
- As far as the rank (A) is concerned, if:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 \qquad \text{and} \qquad \sigma_{r+1} = \cdots = \sigma_p = 0$$

  Then the rank of A is r, the kernel of A is the span of the column vectors of V, $\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n\}$, and the range of A is the span of the column vectors of U, $\{\mathbf{u}_1, \ldots, \mathbf{u}_p\}$.

Generalized inverse

The **Generalized Inverse of a matrix** A is a matrix that can provide **solutions to systems of linear equations** that may not have unique solutions or may not be solvable using the regular inverse (such as least squares problems). There are different types of generalized inverses, but one of the most commonly used is the **Moore-Penrose pseudo-inverse**, denoted as $A^\dagger$.

Definition 1: Moore-Penrose

Suppose that $A \in \mathbb{R}^{m \times n}$ has rank equal to r and that it admits a SVD of the type $U^T A V = \Sigma$. The matrix:

$$A^\dagger = V \Sigma^\dagger U^T \tag{55}$$

Is called the **Moore-Penrose pseudo-inverse matrix**, being:

$$\Sigma^\dagger = \operatorname{diag}\left\{\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_p}, 0, \ldots, 0\right\} \tag{56}$$

The matrix $A^\dagger$ is also called the **generalized inverse of** A. Also, if $n = m = \operatorname{rank}(A)$, then $A^\dagger = A^{-1}$.

The Moore-Penrose pseudo-inverse matrix is used in the SVD method to **solve the overdetermined systems** using the least squares technique.

Theorem 10. *Let $A \in \mathbb{R}^{m \times n}$ with SVD given by $A = U \Sigma V^T$. Then the unique solution to the equation 51 is:*

$$\mathbf{x}^* = A^\dagger \mathbf{b} \tag{57}$$

Where $A^\dagger$ is the pseudo-inverse of A.

How to calculate SVD

The **Householder reflection** (or Householder transformation) is a method used in linear algebra to zero out the subdiagonal elements of a matrix, transforming it into a simpler form such as an upper triangular matrix or a bidiagonal matrix.

The use of Householder reflections is **recommended** because they provide a **numerically stable and efficient way to reduce a matrix to bidiagonal form**. This reduction makes the subsequent **steps of the SVD calculation easier and more computationally efficient**.

$$U_1^T A V_1 = B = \begin{bmatrix} d_1 & f_1 & \cdots & \cdots & 0_n \\ 0 & d_2 & f_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & d_{n-1} & f_{n-1} \\ 0 & 0 & \cdots & 0 & d_n \end{bmatrix}$$

It follows that $T = B^T B$ is symmetric and tridiagonal. We could then apply the QR algorithm directly to B.

5 Multigrid methods

5.1 Idea of MG methods

The **Multigrid (MG) methods** are efficient **algorithms for solving large systems of linear equations**, particularly those **arising from the discretization of partial differential equations** (PDEs). They're especially useful for problems that exhibit behavior on multiple scales.

A multigrid (MG) method is an iterative algorithm of the form:

$$\mathbf{x}^{(k+1)} = \mathrm{MG}\left(\mathbf{x}^{(k)}\right) \qquad k \geq 0 \tag{58}$$

For solving the (typically) sparse linear systems of equations stemming from the numerical discretization of differential equations. The **MG methods are based on**:

- ***Hierarchy of levels*** (associated with a hierarchy of discretization):
  - **Fine Grid**. The finest grid **captures the most detailed features** of the problem. This is where the original problem is defined and where the final solution needs to be accurate.
  - **Coarse Grids**. They are lower resolution versions of the fine grid. They **capture broader**, **large-scale features** of the problem. The coarser the grid, the **fewer the details**, but **computations are cheaper and faster**.

    Coarse grids help in correcting the errors that are hard to eliminate on finer grids due to their global nature.
- ***MG cycles*** reduce all error components by a fixed amount (bounded well below one), regardless of the dimension n of the system.

The **main idea** of MG is to **accelerate the convergence of a basic iterative method** by a global correction of the fine grid solution approximation accomplished by solving a coarse problem. The coarse-level problem should be *similar* to the fine grid problem. The cost of (direct) solution of the coarse problem should be negligible compared to the cost of one relaxation sweep on the fine grid.

In other words, the main goal of the multigrid method is to speed up the convergence of an iterative method for solving systems of linear equations. This acceleration is achieved by globally correcting the solution approximation on the fine grid by solving a similar problem on a coarser grid.

5.2 How it works

The multigrid method is divided into **seven parts** that make the MG method work.

1. **Coarse Grids** (page 66)
2. **Correction** (page 69)
3. **Interpolation Operator** (page 70)
4. **Restriction Operator** (page 74)
5. **Two-Grid Scheme** (page 76)
6. **V-Cycle Scheme** (page 78)

These elements **work together to handle errors at different scales**, making the method **highly effective for solving large and complex systems** of linear equations.

Note that this **is <u>not</u> an algorithm!** We can think of the MG method as a toolbox filled with powerful tools, each designed to address different aspects of solving complex problems efficiently.

√× Notation used in MG methods

We will use the subscript h to indicate the Grid Spacing. The variable h represents the **distance between two successive grid points on the fine grid**. For example, if the domain is divided into N intervals, the grid spacing h is typically $\frac{1}{N}$.

- **Residual $\mathbf{r}_h$** represents the residual calculated on the fine grid with spacing h. It's the difference between the current solution and the exact solution on this grid.
- **Solution $\mathbf{x}_h$** indicates the approximate solution on the fine grid. This solution is updated iteratively using the Multigrid method.
- **Operator A_h** is the matrix or operator that represents the system of equations on the fine grid. This operator acts on the solution $\mathbf{x}_h$.
- **Right-Hand Side $\mathbf{b}_h$** is the right-hand side vector of the system of equations on the fine grid. It's what the solution $\mathbf{x}_h$ should ideally satisfy when acted upon by A_h.
- **Error on the Fine Grid $\mathbf{e}_h$** is the error estimate or correction term calculated on the fine grid. It represents the difference between the true solution and the current approximate solution on the fine grid with spacing h.

What's more, **if we move to a coarser grid**, the grid spacing will be greater, indicated by $2h$ or even $4h$ if we make a significant jump to a coarser level.

5.2.1 Coarse Grids

Purpose. Simplifies the problem by **reducing the number of grid points**, capturing broad features, and addressing low-frequency errors. In other words, reduce the grids with fewer points and greater spacing between them compared to the fine grid.

Before we go any further, we need to understand the **difference between Coarse and Fine Grid**. This can be done from an image point of view, for example, an image where we can see the simplification of details:

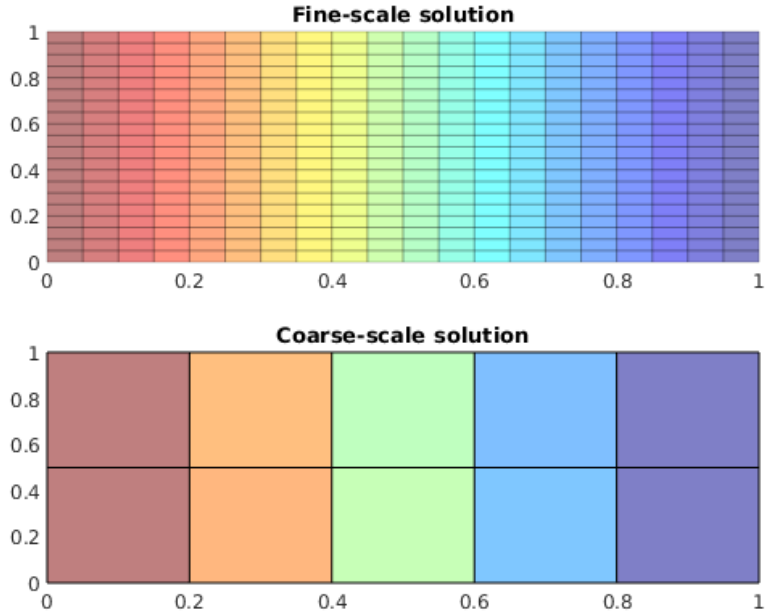
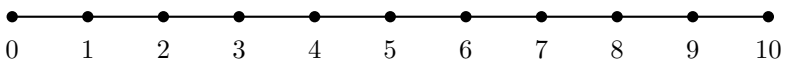


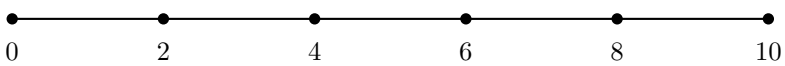
Figure 7: Difference between Coarse and Fine Grid.

But to understand frequency, we have to look at the problem from a one-dimensional point of view, looking at frequencies.

- **Fine Grid** has a high resolution, then many closely spaced points.

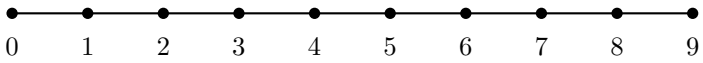


- **Coarse Grid** has a lower resolution, then fewer points spaced farther apart.

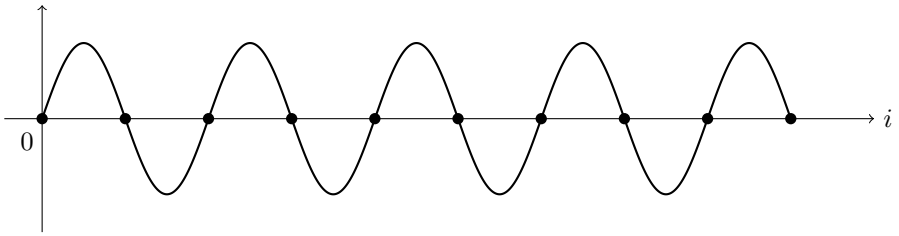


As we **move from the fine grid to the coarse grid, the mode becomes more oscillatory because the same error pattern spans fewer points, increasing its apparent frequency**. The term "mode" refers to the different error patterns or components in the solution. See the following illustration for a 100% understanding.

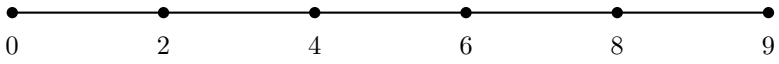
Consider a wave function on the fine grid $w_j = \sin\left(\frac{j\pi}{n+1}i\right)$ (where j determines the frequency, n is the number of points, and i is the index of the grid point), its 1D representation, and the signal:



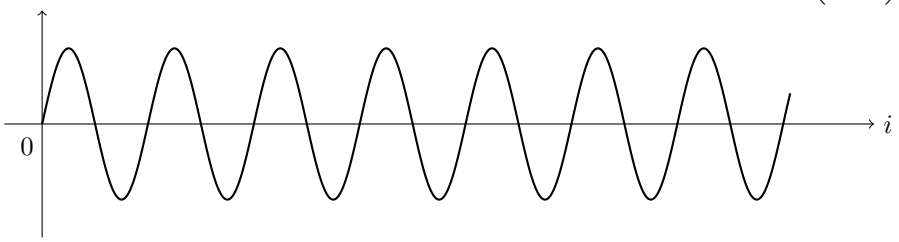
$j = 10 \qquad n = 10 \qquad w_{10} = \sin\left(\frac{10\pi}{11}i\right)$

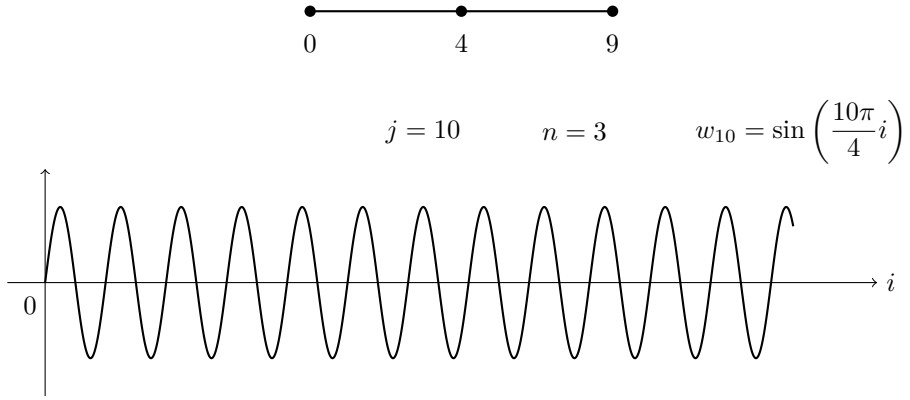


If we pass from the Fine Grid to Coarse Grid reducing the number of points, for example from 10 to 6, we obtain an increase of the oscillatory and also the same error pattern is repeated several times:



$j = 10 \qquad n = 6 \qquad w_{10} = \sin\left(\frac{10\pi}{7}i\right)$





Obviously, **smooth modes on a Fine Grid will look less smooth on a Coarse Grid**.

5.2.2 Correction

Purpose. It is a critical part of the process that **ensures efficient error reduction across multiple grid levels**. The steps are as follows:

1. **Pre-Smoothing**. Relax ν_1 times on $A_h\mathbf{x}_h = \mathbf{b}_h$ to obtain an approximation $\mathbf{x}_h^{(k+\nu_1)}$.

   This step aims to **reduce high frequency errors on the fine grid**. Smoothing (relaxation) techniques such as Gauss-Seidel or Jacobi iterations are typically used.

2. **Compute Residual**. Compute $\mathbf{r}_h^{(k+\nu_1)} = \mathbf{b}_h - A_h\mathbf{x}_h^{(k+\nu_1)}$

   The residual represents the error in the current solution and is used to determine the correction required.

3. **Restriction to Coarse Grid**. Move the residual $\mathbf{r}_h^{(k+\nu_1)}$ from (the fine grid) $\mathcal{F}_h$ to (the coarse grid) $\mathcal{F}_{2h}$ to obtain $\mathbf{r}_{2h}^{(k+\nu_1)}$.

   This step transfers the error information to a coarser grid where it is easier to handle low frequency errors.

4. **Solve on Coarse Grid**. Solve the residual equations $A_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}^{(k+\nu_1)}$ on $\mathcal{F}_{2h}$. Where $\mathbf{e}_{2h}$ is the error estimate.

   Coarse grid solving addresses the lower frequency errors that are more difficult to smooth on the fine grid.

5. **Prolongation to Fine Grid**. Move the error calculated previously $\mathbf{e}_{2h}$ from (the coarse grid) $\mathcal{F}_{2h}$ to (the fine grid) $\mathcal{F}_h$ to obtain $\mathbf{e}_h$.

   This step transfers the correction back to the fine grid, where it can be applied to improve the solution.

6. **Correction**. Correct the approximation obtained on (the fine grid) $\mathcal{F}_h$ with the error estimate obtained on (the coarse grid) $\mathcal{F}_{2h}$, i.e., $\mathbf{x}_h^{(k+1)} = \mathbf{x}_h^{(k+\nu_1)} + \mathbf{e}_h$.

   Applying this correction refines the solution on the fine grid, reducing the overall error.

We can summarize these steps as follows:

1. **Pre-Smoothing**. Reduces high-frequency errors on the fine grid.
2. **Compute Residual**. Identifies remaining errors.
3. **Coarse Grid Correction**. Targets lower-frequency errors by solving on a coarser grid.
4. **Prolongation and Correction**. Transfers and applies corrections to refine the fine grid solution.
5. **Post-Smoothing**. Further smooths any remaining errors.

5.2.3 Interpolation Operator

Purpose. The Interpolation Operator transfers corrections from the coarse grid back to the fine grid, refining the fine grid solution with broader adjustments. In other words, it is a powerful operator for **mapping values from a coarse grid $\mathcal{F}_{2h}$ to a fine grid $\mathcal{F}_h$**. This process is essential to transfer the error corrections or residuals from a coarse grid back to a fine grid, thereby increasing the accuracy of the solution.

Mathematically, the interpolation operator is a linear operator and it is **denoted** as a matrix I_{2h}^h:

$$\begin{aligned} I_{2h}^h : \mathcal{F}_{2h} &\longrightarrow \mathcal{F}_h \\ \mathbb{R}^n &\longrightarrow \mathbb{R}^m \end{aligned} \tag{59}$$

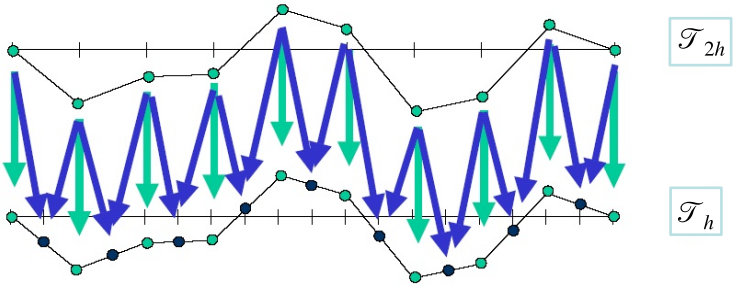
And it is multiplied by the coarse grid $\mathbf{v}_{2h}$ to get the fine grid with the interpolated values $\mathbf{v}_h$:

$$I_{2h}^h \mathbf{v}_{2h} = \mathbf{v}_h \tag{60}$$

It isn't a simply multiplication, because each position of the fine grid is given by:

$$\mathbf{v}_{h,i} = \begin{cases} \mathbf{v}_{2h,i} & \text{if the node } i \text{ is common node of both } \mathcal{F}_h \text{ and } \mathcal{F}_{2h} \\ \dfrac{\mathbf{v}_{2h,i}^+ + \mathbf{v}_{2h,i}^-}{2} & \text{if the node } i \text{ in } \mathcal{F}_h \text{ is not a node in } \mathcal{F}_{2h} \end{cases}$$

Perhaps this discussion is easier to understand graphically. As we can see, for nodes that exist only in the fine grid and not in the coarse grid, the value is interpolated as the average of the neighboring coarse grid nodes.



⚖ Smooth vs Oscillatory errors

To fully understand the MG method, it is important to understand when to use it. The interpolation operator highlights when and why the method can be effective or ineffective. **If the exact error on the fine grid $\mathcal{F}_h$** is:

- **Smooth**: an interpolation of the coarse grid error $\mathbf{e}_{2h}$ should give a **good representation** of the exact error.

  The smooth errors are errors that change gradually over the grid. So if we interpolate a smooth error from a coarse grid to a fine grid (using the interpolation operator), the interpolation will be accurate. This is because the changes in the error are well captured by the averaging, so that the interpolated fine grid values are very similar to the original smooth error. See the Figure 8 to see why this is a good representation of the exact error.

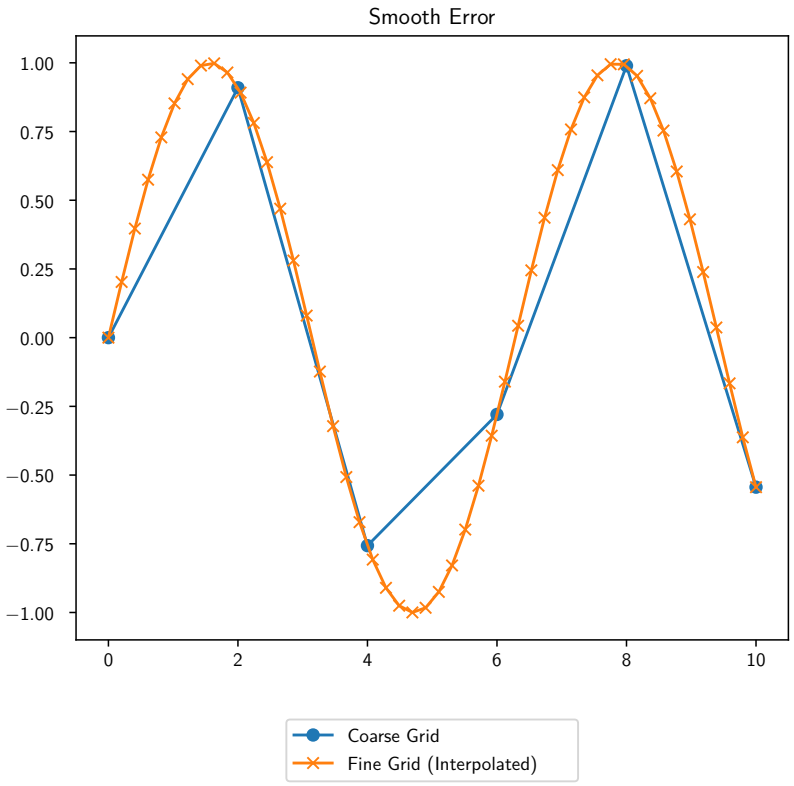


Figure 8: The figure shows what happens when we encounter a smooth error. As we can see, the coarse grid error gives a good representation of the exact error of the fine grid. As the error changes gradually, the application of the interpolation from the coarse grid to the fine grid guarantees the preservation of the smoothness, so that the interpolated values accurately represent the true error.

Oscillatory: an interpolation of the coarse grid error $\mathbf{e}_{2h}$ should give a **poor representation** of the exact error.

The oscillatory errors are errors that change rapidly and frequently over the grid. So if we interpolate an oscillatory error from a coarse grid to a fine grid (using the interpolation operator), the interpolation might not be accurate. This is caused because the rapid changes in the error are not captured well by simple averaging, leading to a less accurate representation. See the Figure 9 to see why this is a poor representation of the exact error.

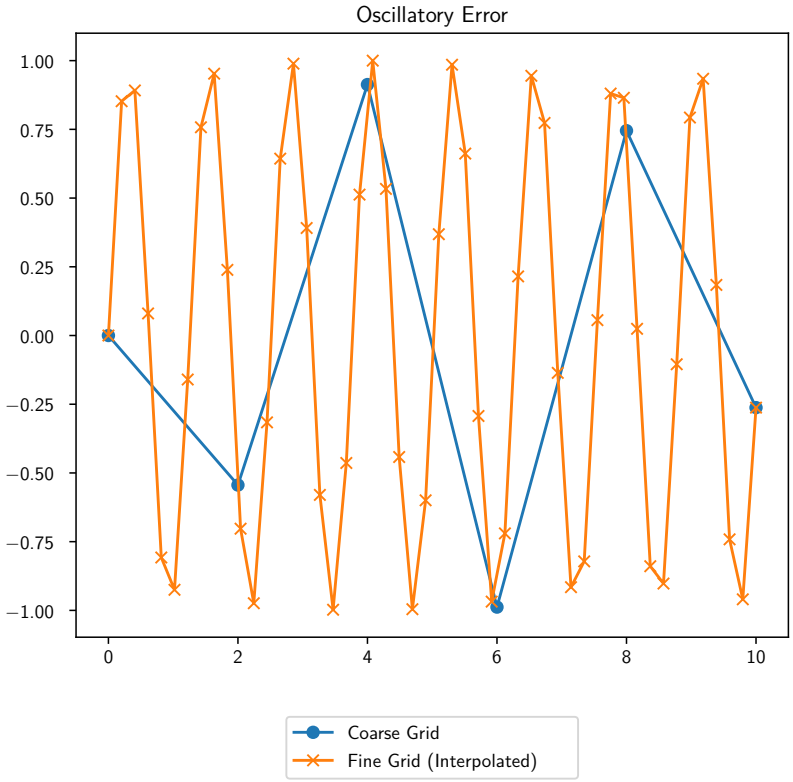


Figure 9: The figure shows what happens when we encounter an oscillatory error. As we can see, the coarse grid error is a very poor representation of the exact fine grid error. This is because the oscillatory errors are smoothed out when interpolating from the coarse grid. The averaging process inherent in the interpolation can't fully capture the high frequency changes, resulting in a loss of accuracy in representing the true error.

The Python code used to generate the plots. It requires numpy and matplotlib to be installed:

```
pip install numpy matplotlib
```

```python
import numpy as np
import matplotlib.pyplot as plt

# Coarse grid
coarse_x = np.linspace(0, 10, 6)
smooth_coarse_y = np.sin(coarse_x)
oscillatory_coarse_y = np.sin(5 * coarse_x)

# Fine grid
fine_x = np.linspace(0, 10, 50)
smooth_fine_y = np.sin(fine_x)
oscillatory_fine_y = np.sin(5 * fine_x)

plt.figure(figsize=(12, 6))

# Smooth error
plt.subplot(1, 2, 1)
plt.plot(coarse_x, smooth_coarse_y, 'o-', label='Coarse Grid')
plt.plot(fine_x, smooth_fine_y, 'x-', label='Fine Grid (
    Interpolated)')
plt.title('Smooth Error')
plt.legend(loc='lower center', bbox_to_anchor=(0.5, -0.25))

# Oscillatory error
plt.subplot(1, 2, 2)
plt.plot(coarse_x, oscillatory_coarse_y, 'o-', label='Coarse Grid')
plt.plot(fine_x, oscillatory_fine_y, 'x-', label='Fine Grid (
    Interpolated)')
plt.title('Oscillatory Error')
plt.legend(loc='lower center', bbox_to_anchor=(0.5, -0.25))

plt.tight_layout()
plt.show()
```

5.2.4 Restriction Operator

Purpose. The Restriction Operator **transfers data from a fine grid to a coarse grid**. It can be thought of as the opposite of the interpolation operator (page 70).

Mathematically, the restriction operator is a linear operator and it is **denoted** as a matrix I_h^{2h}:

$$\begin{aligned} I_h^{2h} : \mathcal{F}_h &\longrightarrow \mathcal{F}_{2h} \\ \mathbb{R}^n &\longrightarrow \mathbb{R}^m \end{aligned} \tag{61}$$

Unlike the interpolation operator, we have more problems here. Because to apply this tool and to guarantee that the coarse grid construction reflects the fine grid problem, we cannot use a simple system. There are **two ways to apply the restriction operator**:

- **Injection**. It is the simple form of restriction where **coarse grid values are directly taken from the corresponding fine grid values**. It transfers the value without any averaging. Mathematically it is expressed with the equation:

$$I_h^{2h}\mathbf{w}_h = \mathbf{w}_{2h} \tag{62}$$

  ✔ **Pros**

  - Very **simple**.
  - Computationally **inexpensive**.

  ✖ **Cons**

  - It can **miss fundamental details** of the solution.
  - **Less accurate** error correction.
  - Very **poor overall results**.

- **Galerkin condition**. It ensures that the coarse grid operator A_{2h} accurately represents the fine grid operator A_h. This means that the solution on the **coarse grid is a reasonable approximation of the solution on the fine grid**. The Galerkin condition is expressed by the following equation:

$$A_{2h} = I_h^{2h} A_h I_{2h}^h \tag{63}$$

  Where:

  - I_h^{2h} is the *interpolation operator* (page 70);
  - I_{2h}^h is the *restriction operator*;
  - A_h is the *fine grid*;
  - A_{2h} is the result *coarse grid* that we obtain.

  The Galerkin condition can be viewed as a scaled transpose of the interpolation operator:

$$I_h^{2h} = c\left(I_{2h}^h\right)^T \tag{64}$$

  Where c is a scalar factor in the real numbers $\mathbb{R}$, it adjusts the magnitude of the values when the restriction operator is applied.

  ✔ Pros

  - Ensures mathematical consistency.
  - Ensures accurate representation of the fine grid problem on the coarse grid.
  - Leads to effective error correction and greater accuracy in solutions.

  ✖ Cons

  - Slightly more complex and computationally intensive than injection.

5.2.5 Two-Grid Scheme

Purpose. The Two-Grid Scheme is a simple strategy that **uses only two levels** (a fine grid and a coarse grid) **to iteratively improve the solution**. The general idea is:

1. Given an initial guess $\mathbf{x}_h^{(0)}$;
2. While the stopping criteria is met:
   (a) Compute:
$$\mathbf{x}_h^{(k+1)} = \text{MG}\left(\mathbf{x}_h^{(k)}, \mathbf{b}_h, \nu_1, \nu_2\right) \tag{65}$$

The MG method is invoked and the algorithm is:

1. **Apply our favorite method for ν_1 times**. Do ν_1 iterations using a chosen method (e.g. Jacobi) on the system $A_h\mathbf{x}_h = \mathbf{b}_h$ starting with the initial guess $\mathbf{x}_h^{(k)}$. The solution after these iterations is $\mathbf{y}_h^{(\nu_1)}$.
$$\mathbf{y}_h^{(\nu_1)} \leftarrow \text{Relax } \nu_1 \text{ times on } A_h\mathbf{x}_h = \mathbf{b}_h$$
2. **Compute Fine Grid Residual**. Calculate the residual on the fine grid $\mathbf{r}_h^{(\nu_1)} = \mathbf{b}_h - A_h\mathbf{y}_h^{(\nu_1)}$:
$$\mathbf{r}_h^{(\nu_1)} \leftarrow \mathbf{b}_h - A_h\mathbf{y}_h^{(\nu_1)}$$
3. **Restriction to Coarse Grid**. Move the residual $\mathbf{r}_h^{(\nu_1)}$ from the fine grid to the coarse grid to obtain the residual $\mathbf{r}_{2h}^{(\nu_1)} = I_h^{2h}\mathbf{r}_h^{(\nu_1)}$:
$$\mathbf{r}_{2h}^{(\nu_1)} = I_h^{2h}\mathbf{r}_h^{(\nu_1)}$$
4. **Solve on Coarse Grid**. Solve the residual equations $A_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}^{(\nu_1)}$ on the coarse grid. Where $\mathbf{e}_{2h}$ is the error estimate. This can be helpful because lower frequency errors are harder to smooth on the fine grid.
$$A_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}^{(\nu_1)} \tag{66}$$
5. **Return to Fine Grid**. Move the error calculated previously $\mathbf{e_{2h}}$ from the coarse grid to the fine grid to obtain $\mathbf{e}_h$:
$$\mathbf{e}_h = I_{2h}^h\mathbf{e}_{2h}$$
6. **Update and apply correction**. Correct the approximation obtained on the fine grid with the error estimate obtained on the coarse grid. Applying this correction refines the solution on the fine grid, reducing the overall error:
$$\mathbf{y}_h^{(\nu_1+1)} \leftarrow \mathbf{y}_h^{(\nu_1)} + \mathbf{e}_h$$
7. **Apply ν_2 smoothing iterations**. Do ν_2 iterations using a chosen smoother (e.g. Jacobi) on the system $A_h\mathbf{x}_h = \mathbf{b}_h$ starting with the updated solution (initial guess) $\mathbf{y}_h^{(\nu_1+1)}$. The solution after these iterations is $\mathbf{y}_h^{(\nu_1+1+\nu_2)}$.

These additional smoothing iterations are essential to refine the solution and ensure that both high and low frequency errors are adequately addressed. It can also help stabilize the solution by ensuring that any residual errors are minimized.

$$\mathbf{y}_h^{(\nu_1+1+\nu_2)} \leftarrow \text{Relax } \nu_2 \text{ times on } A_h\mathbf{x}_h = \mathbf{b}_h$$

8. **Return the result**. The final updated solution $\mathbf{x}_h^{(k+1)}$ is set to $\mathbf{y}_h^{(\nu_1+1+\nu_2)}$.

$$\mathbf{x}_h^{(k+1)} \leftarrow \mathbf{y}_h^{(\nu_1+1+\nu_2)}$$

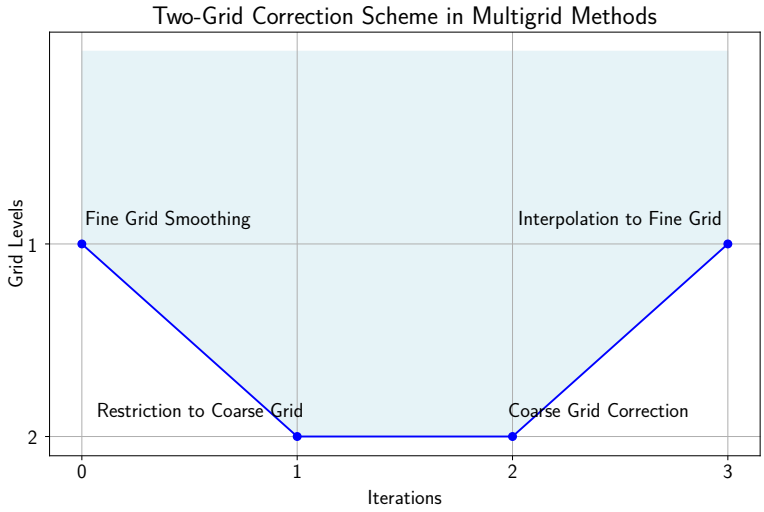


Figure 10: Graphical representation of the Two-Grid Scheme.

5.2.6 V-Cycle Scheme

Purpose. The V-Cycle Scheme has the powerful **ability to move between fine and coarse grids** in a *structured manner*, *efficiently* and *recursively* **reducing errors at all levels**. It is very similar to the Two-Grid scheme, but the V-Cycle version allows us to go as deep as we want (or can). The general idea is:

1. Given an initial guess $\mathbf{x}_h^{(0)}$;
2. While the stopping criteria is met:
   (a) Compute:
   $$\mathbf{x}_h^{(k+1)} = \text{MG}\left(\mathbf{x}_h^{(k)}, \mathbf{b}_h, \nu_1, \nu_2, J\right) \tag{67}$$

As in the Two-Grid scheme, the arguments are the same, but the difference is the parameter J, which indicates the maximum depth of the algorithm. However, when the MG method is invoked, the algorithm is executed:

1. **Fine Grid Smoothing (pre-smoothing)**. We start at the finest grid level (top of the V shape). We apply ν_1 iterations of a smoothing algorithm such as Jacobi, on the system $A_h\mathbf{x}_h = \mathbf{b}_h$ with the initial guess $\mathbf{x}_h^{(k)}$ to reduce high-frequency errors. The solution that we obtain is $\mathbf{y}_h^{(\nu_1)}$.
   $$\mathbf{y}_h^{(\nu_1)} \leftarrow \text{Relax } \nu_1 \text{ times on } A_h\mathbf{x}_h = \mathbf{b}_h$$
2. **Compute Fine Grid Residual**. Calculate the residual on the fine grid $\mathbf{r}_h^{(\nu_1)} = \mathbf{b}_h - A_h\mathbf{y}_h^{(\nu_1)}$:
   $$\mathbf{r}_h^{(\nu_1)} \leftarrow \mathbf{b}_h - A_h\mathbf{y}_h^{(\nu_1)}$$
3. **Restriction to Coarse Grid**. Move the residual $\mathbf{r}_h^{(\nu_1)}$ from the fine grid to the coarse grid to obtain the residual $\mathbf{r}_{2h}^{(\nu_1)} = I_h^{2h}\mathbf{r}_h^{(\nu_1)}$:
   $$\mathbf{r}_{2h}^{(\nu_1)} = I_h^{2h}\mathbf{r}_h^{(\nu_1)}$$
4. **Recursive V-Cycle on Coarser Grid**. Check if the coarsest level is the desired one, in other words, if we are at the coarsest level we requested when we first invoked the algorithm.

   If the level is the maximum depth requested (j = actual coarsest level), solve the problem or find an approximate solution using direct methods. If we are at this level, we are at the bottom of the V-shape. Otherwise, if the level is not the desired one, we apply the V-cycle process recursively on the coarsest grid, repeating steps 1 through 3 on progressively coarser grids until the coarsest grid is reached.
   $$\begin{cases} \text{Solve } A_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}^{(\nu_1)} & \text{if } J = \text{ actual coarsest level} \\ \mathbf{e}_{2h} = \text{MG}\left(\mathbf{0}, \mathbf{r}_{2h}^{(\nu_1)}, \nu_1, \nu_2, j-1\right) & \text{otherwise} \end{cases}$$

5. **Interpolate back to Fine Grid**. If we are here, the recursion has reached the maximum depth. Now we have to come back to the surface and follow the right side of the V-shape. Therefore, we transfer the correction $\mathbf{e}_{2h}$ calculated in the previous step from the coarse to the fine grid using an interpolation operator I_{2h}^{h} (page 70):

$$\mathbf{e}_h = I_{2h}^{h}\mathbf{e}_{2h}$$

6. **Update and apply correction**. Correct the approximation obtained on the fine grid with the error estimate obtained on the coarse grid. Applying this correction refines the solution on the fine grid, reducing the overall error:

$$\mathbf{y}_h^{(\nu_1+1)} \leftarrow \mathbf{y}_h^{(\nu_1)} + \mathbf{e}_h$$

7. **Fine Grid Smoothing (post-smoothing)**. Do ν_2 iterations using a chosen smoother (e.g. Jacobi) on the system $A_h\mathbf{x}_h = \mathbf{b}_h$ starting with the updated solution (initial guess) $\mathbf{y}_h^{(\nu_1+1)}$. The solution after these iterations is $\mathbf{y}_h^{(\nu_1+1+\nu_2)}$.

   These additional smoothing iterations are essential to refine the solution and ensure that both high and low frequency errors are adequately addressed. It can also help stabilize the solution by ensuring that any residual errors are minimized.

$$\mathbf{y}_h^{(\nu_1+1+\nu_2)} \leftarrow \text{Relax } \nu_2 \text{ times on } A_h\mathbf{x}_h = \mathbf{b}_h$$

8. **Recursively return to the surface**. We return the result obtained in the previous step $\mathbf{y}_h^{(\nu_1+1+\nu_2)}$ at the previous level of coarsest. Since we are in a recursive path, if the previous caller is the main, then the method stops, otherwise the previous caller recalculates its results from step 5 to 8, and so on.

$$\mathbf{x}_h^{(k+1)} \leftarrow (j-1 \text{ recursive steps}) \leftarrow \mathbf{y}_h^{\nu_1+1+\nu_2}$$

Note that the algorithm seems very similar to the Two-Grid Scheme because the V-Cycle Scheme is an extension applied j times! In Figure 11, we can see why the V-cycle scheme has a V-shape.

However, the V-cycle scheme is only one of several MG cycling schemes. Other types of schemes are W-cycle and F-cycle, and can be analyzed at the following MIT link.

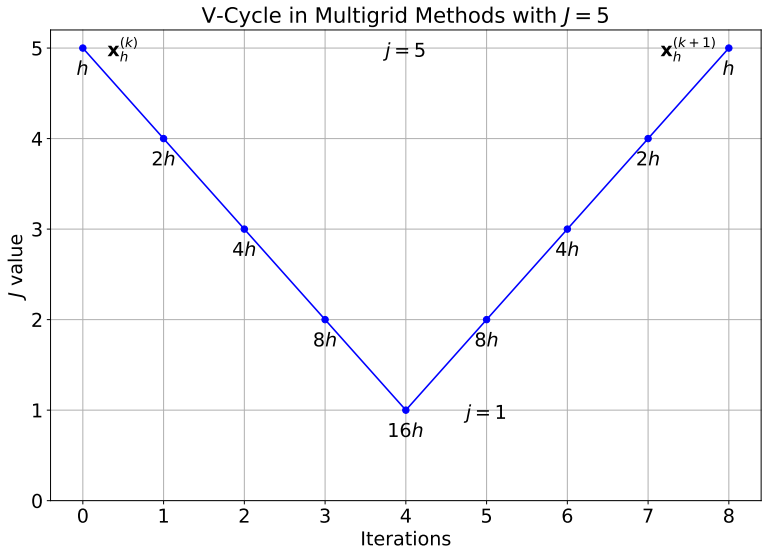


Figure 11: V-Cycle Scheme.

$ How much does it cost?

The V-Cycle Scheme has a **convergence less than one and independent** of h and **it costs only** $O\left(n^d \log\left(n\right)\right)$.

At each level j the values $\mathbf{x}_h^{(k)}$ and $\mathbf{b}_h$ must be stored. Also, each successively coarser grid requires progressively less memory because the number of grid points is reduced by a factor at each level. In d dimensions, the coarse grid has $\approx 2^{-d}$ the number of points as the finer grid. Therefore the memory requirement is:

$$\text{Storage cost} \approx \frac{2n^d}{1-2^{-d}}$$

Furthermore, for $d = 1$ the memory requirement is less than twice that of the fine-grid problem alone.

5.3 Classical Algebraic Multigrid (AMG)

Classical Algebraic Multigrid (AMG) is a numerical **method for solving large systems of equations**, especially those arising from the **discretization of partial differential equations**.

It is a **type of multigrid method that uses matrix coefficients to construct a hierarchy of grids** rather than relying on geometric information (such as the V-cycle scheme). It aims to speed up the convergence of iterative methods by combining smoothing operations with coarse grid corrections.

Why is AMG one of the best MG methods?

When we think about how the V-cycle works, we notice an interesting thing. Each MG tool presented in the previous pages requires an interpretation of the geometric properties of the problems. Unfortunately, especially in the real world, it is very difficult to understand the geometric relationship, and mainly it avoids the coding necessary for a true multigrid implementation (we mean an implementation of "how can we geometrically pass from a fine to a coarse grid without losing important details or conditions?).

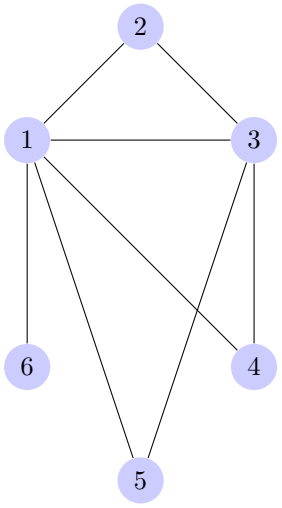
The **main goal of the AMG method is its *geometric independence***. Unlike geometric multigrid methods, which rely on the geometric structure of the problem (grid spacing, shape, etc.), AMG **constructs its grid hierarchies based purely on the algebraic structure of the system matrix**. This makes it highly versatile and applicable to a wide range of problems, including those with complex geometries or unstructured grids. This is one of the most important differences, but the AMG also has other good points (efficiency, applicability, etc.).

AMG Basis

The method is divided into several theoretical concepts:

1. Algebraic Multigrid (AMG) makes extensive use of **graph-based** concepts. The **system matrix** (representing the discretized problem) can be viewed as a **graph**. Each **node in the graph corresponds to a grid point**, and each **edge represents a connection** (or interaction) **between grid points**. For example, the following sparse matrix has the corresponding graph.

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



2. Classical AMG is based on the observation that the **algebraic smooth error varies slowly in the direction of the matrix's relatively large (negative) coefficients**. This gives us an algebraic way to track smooth errors. However, we still need to define large.

Definition 1: strong connection

Given a threshold $\theta \in (0, 1)$ we say that i is **strongly connected** with j if:

$$-a_{i,j} \geq \theta \max_{k \neq i} (-a_{i,k}) \tag{68}$$

Let us denote by S_i the set of vertices that i is strongly connected to by:

$$S_i = \{j \in N_i \ : \ i \text{ strongly connects to } j\} \tag{69}$$

Where:

$$N_i = \{j \neq i \ : \ a_{i,j} \neq 0\}$$

This gives us a strength matrix S, with S_i as its i-th row. AMG uses the concept of *strong connection* to **decide how strongly nodes (grid points) are connected**. This is based on the matrix coefficients. Strong connections are those **where the matrix coefficients are relatively large**, indicating significant interactions between grid points.

3. **Standard Coarsening**. Standard coarsening in AMG involves **reducing the number of variables** (or degrees of freedom) **in the problem**. This is achieved by **selecting a subset of nodes**, known as coarse nodes (**C-vertices**), while the **remaining nodes become** fine nodes (**F-vertices**). The **goal is to simplify the problem while preserving its essential characteristics**.

   - **C-vertices** (Coarse nodes): These are the selected nodes that will form the coarse grid.

- **F-vertices** (Fine nodes): These are the remaining nodes that are not selected as coarse nodes.

To put it simply, in AMG we deal with the original problem on a *fine grid*. However, solving large problems directly on this fine grid can be computationally expensive. To simplify, we **create several "coarser" versions of this grid**, in which the **problem size is progressively reduced**. This process is called *standard coarsening*.

How can we apply the Standard Coarsening?

It requires an observation before use. The oscillatory error should not be a problem, as this error is typically easier to reduce using standard relaxation methods on fine grids. However, the real dilemma is the smooth error, which can't be reduced by simple relaxation methods. **When applying standard coarsening, we need to focus on reducing smooth error while building each coarse grid group and preserving the most fundamental information**.

Implementation. To achieve this, we need to approach the problem from an algebraic point of view. The smooth error tends to vary slowly along strong connections (edges in the graph). Essentially, strong connections represent significant interactions or relationships between nodes (vertices). By **coarsening in the direction of** these **strong connections**, we **preserve the most critical aspects of the problem**, resulting in a more accurate and efficient MG method. In other words, we focus our coarsening efforts on the most "meaningful" parts of the graph, where the important information is.

What happens in practice. In practice, standard coarsening **divides the vertices into** Coarse (**C-vertices**) and Fine (**F-vertices that are strongly connected to the C-vertices**) sets. The main idea is to **ensure that each F-vertex has a strong connection to at least one C-vertex**. This allows us to approximate the values at the F-vertices by a linear combination of the values at the C-vertices, preserving the important relationships in the original problem.

What happens after Standard Coarsening. The values at the F-vertices can be expressed as a weighted combination of the values at their neighboring C-vertices. This ensures that the coarser problem is a good approximation of the finer problem.

General Coarsening Algorithm

Given a strength matrix S indicating the strong connections between nodes, the algorithm is:

(a) **Initialization**. We create an empty set C for Coarse vertices and an empty set F for Fine vertices.

(b) **Select an independent set of C-vertices**. Choose an independent set of C-vertices from the graph of S. An independent set means that no two selected C-vertices are directly connected by a strong connection.

The selection process is as follows:

i. **Choose a C-vertex**. Start with a node and mark it as a C-vertex. In general, we start at the node with the highest number of strong connections (or highest weight, if applicable).

ii. **Populate Fine vertices set**. All vertices strongly connected to the previously selected C-vertex become F-vertices.

iii. **Repeat**. We repeat the process by selecting another vertex from the undecided vertices as a C-vertex and making the vertices strongly connected to it as F-vertices.

iv. **Stop when all vertices are classified as either C-vertex or F-vertex**.

(c) **Select additional C-vertices**. Ensure that every F-vertex has a strong connection to at least one C-vertex. If any F-vertex is not strongly connected to a C-vertex, convert that F-vertex into a C-vertex to ensure the interpolation requirements are satisfied.

Interpolation

Interpolation is used to **estimate unknown values at fine nodes (F-nodes) using the known values at coarse nodes (C-nodes)**. It's crucial for maintaining accuracy and efficiency.

Let $\mathbf{e} = (e_1, e_2, \dots, e_i, \dots)$ be the error; a simple characterization of algebraic smooth error is $A\mathbf{e} \approx 0$. In other words:

$$a_{i,i}e_i + \sum_{j \in N_i} a_{i,j}e_j \approx 0 \qquad i \in F \tag{70}$$

The idea is that we want to choose proper weight $w_{i,j}$ such that for any algebraic smooth errors:

$$e_i \approx \sum_{j \in C} w_{i,j}e_j \qquad i \in F$$

But if we define for $i \in F$:

- C_i the C-points strongly connected to i:

$$C_i = C \cap N_i$$

- F_i^S the F-points strongly connected to i:

$$F_i = F \cap N_i$$

- $C_i^S = C \cap S_i$
- N_i^W all points weakly connected to i:

$$N_i^W = \frac{N_i}{\left(C_i^S \cup F_i^S\right)}$$

We can rewrite the characterization of algebraic smooth error as:

$$a_{i,i}e_i + \sum_{j \in N_i} a_{i,j}e_j = 0 \qquad \alpha = \frac{\sum_{j \in N_i} a_{i,j}}{\sum_{j \in C_i^S} a_{i,j}} \tag{71}$$

We conclude that the **formula of direct interpolation** is:

$$w_{i,j} = \alpha \frac{a_{i,j}}{a_{i,i}} \qquad i \in F,\ j \in C_i^S \qquad \alpha = \frac{\sum_{j \in N_i} a_{i,j}}{\sum_{j \in C_i^S} a_{i,j}} \tag{72}$$

The above direct interpolation can be applied as long as C_i^S.

$ How much does it cost?

The cost of each iteration in the Algebraic Multigrid (AMG) method primarily **depends on the operations involved**, such as smoothing, restriction, interpolation, and correction (the all tools that we have already discussed in the previous pages!). The cost is **generally linearly proportional to the problem size**. This means that as the problem size increases, the cost increases linearly, making AMG methods efficient for large-scale problems.

However, leaving aside the iteration cost for the moment, the **AMG method is the best of the multigrid methods because the construction of the MG hierarchy is done using only information from the matrix and not from the geometry of the problem**. This is the main and most important key. This is one of the most important reasons to choose AMG, especially in real practice problems.

Can it be parallelized?

AMG is not only the best because it is geometric independent, but also because it lends itself very well to parallelization! In general, **AMG methods are well suited for parallelization, especially for large problems**. The multi-level structure of AMG allows the workload to be distributed across multiple processors. However, the efficiency of parallelization depends on the specific implementation and the problem to be solved (of course). Optimizations and careful communication management can help achieve better parallel performance.

6 Domain Decomposition Methods

6.1 Introduction

Domain Decomposition Methods (DDM) are **numerical techniques used to solve large-scale computational problems by breaking them into smaller, more manageable subproblems**. These methods are essential in scientific computing, engineering simulations, and various other fields that require solving extensive linear systems or partial differential equations (PDEs).

What Are Domain Decomposition Methods?

DDM involves dividing a large **computational domain into smaller subdomains**. These subdomains are then **solved independently**, often **in parallel**, and their **solutions are combined to form the overall solution to the original problem**. This approach is particularly useful for problems that are too large to be solved as a single system due to computational limitations.

Importance of Domain Decomposition Methods

1. **Parallelism**: By solving subdomains in parallel, DDM significantly reduces the computation time, making it **feasible to tackle massive problems** that would otherwise be intractable.
2. **Scalability**: These methods can handle extremely large problems, ensuring that **computational resources are used efficiently**, and allowing for the **solution of problems on supercomputers or distributed computing systems**.
3. **Modularity**: Breaking down a complex problem into smaller subproblems makes it **easier to manage**, **understand**, and **solve**. This modularity also facilitates debugging and improving algorithms.
4. **Flexibility**: DDM can be **applied to various types of problems** across different disciplines, including fluid dynamics, structural mechanics, and electromagnetic simulations. This versatility makes them a powerful tool in the computational scientist's toolkit.
5. **Improved Convergence**: With appropriate preconditioners and iterative methods, DDM can enhance the convergence rates of solving systems, leading to **faster and more accurate solutions**.

6.2 Overlapping Subdomains

Overlapping Subdomains refer to the concept within Domain Decomposition Methods where the **computational domain is divided into smaller regions that share some common boundaries**. These shared boundaries are what make them *overlapping*. This method is widely used to **enhance the convergence and accuracy of solving large-scale computational problems**, particularly partial differential equations (PDEs).

Key Characteristics of Overlapping Subdomains

1. **Shared Boundaries**: The subdomains have regions where their boundaries overlap with adjacent subdomains. This overlapping region is critical for the exchange of information between subdomains.
2. **Independent Solutions**: Each subdomain is solved independently, often in parallel, using updated boundary conditions from adjacent subdomains.
3. **Iterative Process**: Solutions are updated iteratively by alternating between solving the subdomains and using the latest boundary conditions. This process continues until convergence is achieved.

Example 1

Suppose we have a domain Ω that we want to solve a PDE on. We can divide Ω into two overlapping subdomains Ω_1 and Ω_2:

- Subdomain Ω_1: Contains a portion of Ω and overlaps with Ω_2 at the boundary.
- Subdomain Ω_2: Contains another portion of Ω and overlaps with Ω_1 at the boundary.

Benefits of Overlapping Subdomains

1. **Improved Boundary Condition Handling**: The overlapping regions allow for better and more accurate boundary condition updates, leading to improved convergence rates.
2. **Enhanced Convergence**: The iterative updates and information exchange in the overlapping regions help accelerate the convergence of the solution.
3. **Parallel Processing**: Each subdomain can be solved independently, making it suitable for parallel computing, which significantly reduces computation time.

6.2.1 Alternating Schwarz Method

The **Alternating Schwarz Method** is a classical **iterative technique** used in numerical linear algebra **to solve partial differential equations (PDEs)**. It is a type of domain decomposition method that divides the computational domain into overlapping subdomains.

Main idea

- **Domain Decomposition**: The main idea is to **break down a large problem into smaller, overlapping subproblems**. By solving these smaller problems iteratively, the overall solution can be efficiently approximated.
- **Overlap**: **Subdomains overlap at their boundaries**, allowing for more accurate boundary condition updates and improving the convergence rate of the method.

Algorithm

1. **Initialization**: Start with an *initial guess* for the solution over the entire domain.
2. **Subdomain Solutions**: Alternately solve the PDE on each subdomain:
   - On the first subdomain, using boundary conditions updated from the initial guess or the latest solution.
   - On the second subdomain, using boundary conditions updated from the solution of the first subdomain.
3. **Iteration**: Continue alternating between subdomains, **updating the solution iteratively until convergence is achieved**.

Example 2: Elliptic Partial Differential Equation (PDE)

The problem involves solving an elliptic PDE of the form:

$$Lu = f \quad \text{in } \Omega = \Omega_1 \cup \Omega_2$$

Where:

- L is the **differential operator**.
- u is the **unknown function we are solving for**.
- f is a given function representing **the source term**.
- Ω is the **computational domain**, divided into two overlapping subdomains Ω_1 and Ω_2.

The boundary condition is:

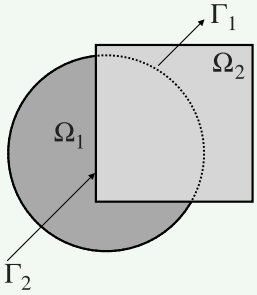
$$u = g \quad \text{on } \partial\Omega$$

Here, $\partial\Omega$ represents the **boundary of the entire domain**.

Domain Decomposition. The domain Ω is divided into two overlapping subdomains Ω_1 and Ω_2, each with its own boundary:

- Γ_1 is the boundary of Ω_1.
- Γ_2 is the boundary of Ω_2.

These subdomains overlap at certain regions, allowing for the exchange of boundary conditions and improving the overall convergence.



Iterative Solution Process. The Alternating Schwarz Method iteratively solves the PDE on each subdomain while updating the boundary conditions based on the solutions from neighboring subdomains. The process involves the following steps:

1. **Initialization**. Start with an initial guess $u^{(0)}$ for the solution.
2. **Subdomain Solutions**:
   - On Ω_1:
   $$\begin{cases} Lu_1^{\left(k+\frac{1}{2}\right)} = f & \text{in } \Omega_1 \\ u_1^{\left(k+\frac{1}{2}\right)} = g & \text{on } \partial\Omega_1 \setminus \Gamma_1 \\ u_1^{\left(k+\frac{1}{2}\right)} = u_2^{(k)} & \text{on } \Gamma_1 \end{cases}$$
   - On Ω_2:
   $$\begin{cases} Lu_2^{(k+1)} = f & \text{in } \Omega_2 \\ u_2^{(k+1)} = g & \text{on } \partial\Omega_2 \setminus \Gamma_2 \\ u_2^{(k+1)} = u_1^{(k+\frac{1}{2})} & \text{on } \Gamma_2 \end{cases}$$
3. **Update**. Combine solutions to form the updated solution $u^{(k+1)}$:
   $$u^{(k+1)} = \begin{cases} u_1^{\left(k+\frac{1}{2}\right)} & \text{in } \Omega_1 \setminus \Omega_2 \\ u_2^{(k+1)} & \text{in } \Omega_2 \end{cases}$$

Convergence. The iterative **process continues until the solutions on the overlapping subdomains converge to a stable solution for the entire domain** Ω. The convergence rate is often enhanced due to the overlap and the iterative updating of boundary conditions.

6.2.2 Discretized Schwarz Methods

The **Discretized Schwarz Methods** involve **applying the Schwarz decomposition principles to a discretized version of the computational domain**. This results in an algebraic system that can be efficiently solved using the same iterative techniques.

- **Discretization Process**. The **continuous domain is discretized into a grid or mesh**, resulting in a **finite set of points that represent the domain**. The PDE is then transformed into a system of linear equations:

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

  - A matrix representing the *discretized operator*;
  - $\mathbf{x}$ is the *vector of unknowns*;
  - $\mathbf{b}$ represents the *source terms* or *boundary conditions*.

- **Overlapping Subdomains**: Each subdomain Ω_i has a **set of grid points** (n) indexed by S_i (with $n_i = |S_i|$). The **overlap between subdomains ensures that the indices in** S_1 and S_2 are **not disjoint** ($S_1 \cap S_2 \neq \emptyset$ and $n_1 + n_2 > n$), facilitating the exchange of boundary information.

- **Boolean Restriction Matrices**: Boolean restriction matrices R_i are used to **extract the relevant components of the solution vector v for each subdomain**. These matrices play a crucial role in the iterative solution process by ensuring that each subdomain only processes its relevant data.

  Formally, for $i = 1, 2$, let R_i be an $n_i \times n$ boolean restriction matrix such that for any vector $\mathbf{v} \in \mathbb{R}^n$:

$$\mathbf{v}_i = R_i\mathbf{v} \in \mathbb{R}^{n_i}$$

  This means $\mathbf{v}_i$ contains the components of $\mathbf{v}$ corresponding to indices in S_i (i.e., those components associated with nodes in Ω_i).

- **Extension Matrices**: Extension matrices are used to **expand the solution from subdomains back to the global domain**. The extension matrix R_i^T maps the local solution $\mathbf{v}_i$ back to the global solution $\mathbf{v}$.

  Formally, for $i = 1, 2$:

$$\mathbf{v} = R_i^T\mathbf{v}_i$$

  This ensures that the components of $\mathbf{v}$ corresponding to indices in S_i are the same as those of $\mathbf{v}_i$, while the remaining components are zero.

- **Principal Submatrices**: Principal submatrices are **submatrices** of the global matrix **A that correspond to the subdomains**. For each subdomain Ω_i, the principal submatrix $\mathbf{A}_i$ is formed by restricting $\mathbf{A}$ to the indices in S_i.

  Formally, for subdomain Ω_i, the principal submatrix $A_i \in \mathbb{R}^{n_i \times n_i}$ is given by:

$$\mathbf{A}_i = R_i\mathbf{A}R_i^T \tag{73}$$

  These submatrices represent the **system of equations for the respective subdomains**, allowing them to be solved independently.

Linking the Concepts

- **Extension Matrices and Boolean Restriction Matrices**. The extension matrices R_i^T and the boolean restriction matrices R_i are **complementary**.
  - Extension Matrices R_i **extracts** the relevant **components of the solution** vector $\mathbf{v}$ **for each subdomain**;
  - Boolean Restriction Matrices R_i^T **extends** the **local solution** $\mathbf{v}_i$ **back to the global solution** $\mathbf{v}$.
- **Principal Submatrices and Overlapping Subdomains**. The principal submatrices $\mathbf{A}_i$ are directly related to the overlapping subdomains. **Each subdomain** Ω_i **has its own principal submatrix** $\mathbf{A}_i$**, which represents the local system of equations**. The overlap between subdomains ensures that the principal submatrices share common indices, facilitating the exchange of boundary information and ensuring consistency in the global solution.

The following pages introduce two types of Discretized Schwarz methods:

1. **Multiplicative Schwarz method**, where subdomains are updated sequentially. The solution from each subdomain is used to update the boundary conditions for the next subdomain in the sequence.
2. **Additive Schwarz method**, where subdomains are solved independently and their solutions are combined additively to form the global solution.

6.2.2.1 Multiplicative Schwarz method

The **Multiplicative Schwarz Method** is an **iterative technique used to solve discretized linear systems**, particularly in the context of domain decomposition methods. It is similar to the block Gauss-Seidel method but uses overlapping blocks. The method involves breaking down a large system into smaller subproblems that can be solved more easily.

1. **Principal Submatrices**. The matrix A is decomposed into principal submatrices A_1 and A_2, corresponding to two subdomains. These submatrices are given by (with eq. 73, page 91):

$$\begin{aligned} A_1 &= R_1 A R_1^T \\ A_2 &= R_2 A R_2^T \end{aligned}$$

   Where R_1 and R_2 are restriction operators that extract the relevant components for each subdomain.

2. **Alternating Schwarz Iteration**. For a discretized problem, the alternating Schwarz iteration takes the following form:

$$\begin{aligned} x^{\left(k+\frac{1}{2}\right)} &= x^{(k)} + R_1^T A_1^{-1} R_1 \left(b - A x^{(k)}\right) \\ x^{(k+1)} &= x^{\left(k+\frac{1}{2}\right)} + R_2^T A_2^{-1} R_2 \left(b - A x^{\left(k+\frac{1}{2}\right)}\right) \end{aligned}$$

   These equations update the solution vector x by solving the subproblems within each subdomain iteratively.

3. **Error Update**. The overall error $e^{(k)} = x - x^{(k)}$ is updated as:

$$e^{(k+1)} = B_{MS} e^{(k)}$$

   Where:

$$B_{MS} = \left(I - R_2^T A_2^{-1} R_2 A\right)\left(I - R_1^T A_1^{-1} R_1 A\right)$$

   The **error propagation matrix** B_{MS} shows how the **error evolves through each iteration**, ensuring convergence of the method.

Sequential Nature of the Multiplicative Schwarz Method

The Multiplicative Schwarz Method involves **sequential computation**, which can be both a strength and a limitation depending on the context of its application.

- **Subdomain Dependency**. In the Multiplicative Schwarz Method, **each subdomain's solution depends on the updated solution of the previous subdomain**. This dependency requires that subdomains be solved in a specific sequence. For example, subdomain Ω_2 uses the updated boundary conditions from the solution of subdomain Ω_1.

- **Iteration Steps**. The method alternates between solving subdomain problems, updating the boundary conditions iteratively. Here's the typical update process:

$$\begin{aligned}\mathbf{x}^{\left(k+\frac{1}{2}\right)} &= \mathbf{x}^{(k)} + R_1^T A_1^{-1} R_1 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right) \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{\left(k+\frac{1}{2}\right)} + R_2^T A_2^{-1} R_2 \left(\mathbf{b} - A\mathbf{x}^{\left(k+\frac{1}{2}\right)}\right)\end{aligned}$$

  Subdomain Ω_2 cannot be updated until subdomain Ω_1 has been solved and its results incorporated.

In fact, the method is called *multiplicative* because:

1. Each subdomain solution depends on the updated boundary conditions from the previous subdomain, effectively *multiplying* the effect of each update;
2. The error update formula for the Multiplicative Schwarz Method involves the sequential multiplication of error reduction operators for each subdomain.

However, there are limitations:

- **Lack of Parallelism**. Because of the sequential nature, **subdomains must be processed one after the other**, making it difficult to exploit parallel computing resources effectively. This limits the scalability of the method on modern high-performance computing systems, where parallelism is crucial for handling large-scale problems efficiently.
- **Longer Computation Times**. Sequential updates mean that each iteration can take longer compared to methods that allow for parallel processing. The overall computation time increases, especially for problems with many subdomains or very large domains.
- **Communication Overhead**. In distributed computing environments, the sequential nature can result in increased communication overhead between processors, as each processor must wait for the previous one to complete its computation before starting.

Gauss-Seidel Analogy

The Multiplicative Schwarz Method is analogous to the block Gauss-Seidel method. Just like the Gauss-Seidel method updates the solution iteratively using previously computed values, the Multiplicative Schwarz Method updates the solution sequentially by *multiplying* the effect of each subdomain's solution.

Pros

Can lead to **faster convergence for certain problems** due to the sequential dependency of updates.

Cons

Limited by its sequential nature, making it **less suitable for parallel processing**.

6.2.2.2 Additive Schwarz method

The **Additive Schwarz method** is a **domain decomposition method that allows subproblems to be solved in parallel**, unlike the Multiplicative Schwarz Method. This method is particularly suitable for parallel computing environments, making it **highly efficient for large-scale problems**.

Main features

- **Parallelism**. The Additive Schwarz Method achieves **parallelism by solving subproblems simultaneously**. This is in contrast to the sequential approach of the Multiplicative Schwarz Method, where subproblems are solved one after another.
- **Block Jacobi Approach**. The method uses a **block Jacobi approach** instead of a block Gauss-Seidel approach. In the block Jacobi approach, **each subdomain problem is solved independently**, allowing for parallel execution.
- **Equations and Iteration Steps**. The iterative process involves updating the solution vector $\mathbf{x}$ by combining the solutions from all subdomains additively:

$$\begin{aligned} \mathbf{x}^{\left(k+\frac{1}{2}\right)} &= \mathbf{x}^{(k)} + R_1^T A_1^{-1} R_1 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right) \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{\left(k+\frac{1}{2}\right)} + \underbrace{R_2^T A_2^{-1} R_2 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right)}_{\text{independently of } \mathbf{x}^{\left(k+\frac{1}{2}\right)}} \end{aligned}$$

  These equations indicate that the solutions for subdomains Ω_1 and Ω_2 can be **computed simultaneously**.
- **Error Update**. The overall error $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$ is updated as:

$$\mathbf{e}^{(k+1)} = B_{AS}\mathbf{e}^{(k)}$$

  where:

$$B_{AS} = (R_2^T A_2^{-1} R_2 + R_1^T A_1^{-1} R_1)A$$

  The error propagation matrix B_{AS} reflects the additive nature of the method, combining the effects of all subdomain solutions.

Additive Schwarz Preconditioner

The two separate equations for the Additive Schwarz Method can be **combined into a single update equation by introducing an Additive Schwarz Preconditioner**. This preconditioner combines the effects of all subdomain solutions into one equation, simplifying the iterative process. The combined update equation is:

$$x^{(k+1)} = x^{(k)} + P_{ad}^{-1}\mathbf{r}^{(k)} \qquad k \geq 0 \tag{74}$$

where P_{ad}^{-1} is the additive Schwarz preconditioner:

$$P_{ad}^{-1} = \left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right) \tag{75}$$

Proof. To simplify the formulation, we can eliminate the intermediate step $x^{\left(k+\frac{1}{2}\right)}$ to obtain a direct update equation:

$$x^{(k+1)} = x^{(k)} + R_1^T A_1^{-1} R_1 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right) + R_2^T A_2^{-1} R_2 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right)$$

This simplifies to:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right)\left(\mathbf{b} - A\mathbf{x}^{(k)}\right) \\ &= x^{(k)} + \left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right)\left(\mathbf{r}^{(k)}\right) \end{aligned}$$

Where:

- $\mathbf{r}^{(k)}$ is the residual vector $\mathbf{b} - A\mathbf{x}^{(k)}$;
- $\left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right)$ is the preconditioner P_{ad}^{-1}.

QED

The additive Schwarz preconditioner is **used to improve the convergence rate** of the iterative solver by combining the effects of all subdomain solutions.

Symmetry of Preconditioner (and PCG)

The preconditioner P_{ad}^{-1} retains the **symmetry** of the original matrix A. This is important because it **ensures compatibility with the Preconditioned Conjugate Gradient (PCG) method**, which requires the system to be symmetric and positive-definite. It is an iterative solver for symmetric positive-definite linear systems. However, the equation for **PCG with the additive Schwarz preconditioner** is:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k P_{ad}^{-1}\mathbf{p}^{(k)} \qquad k \geq 0 \tag{76}$$

Here, α_k is a step size, and $\mathbf{p}^{(k)}$ is the search direction at iteration k (see chapter 2.6, on page 32, for a refresher on Conjugate Gradient).

★ Symmetrized Multiplicative Schwarz Preconditioner

The **standard multiplicative Schwarz iteration matrix is not symmetric**, which can be a limitation when using methods like PCG that require symmetry. To address this, an additional step involving A_1^{-1} is introduced to make the preconditioner symmetric:

1. First Symmetric Step:

$$\mathbf{x}^{\left(k+\frac{1}{3}\right)} = \mathbf{x}^{(k)} + R_1^T A_1^{-1} R_1 \left(\mathbf{b} - A\mathbf{x}^{(k)}\right) \tag{77}$$

2. Second Symmetric Step:

$$\mathbf{x}^{\left(k+\frac{2}{3}\right)} = \mathbf{x}^{\left(k+\frac{1}{3}\right)} + R_2^T A_2^{-1} R_2 \left(\mathbf{b} - A\mathbf{x}^{\left(k+\frac{1}{3}\right)}\right) \tag{78}$$

3. Third Symmetric Step:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{\left(k+\frac{2}{3}\right)} + R_1^T A_1^{-1} R_1 \left(\mathbf{b} - A\mathbf{x}^{\left(k+\frac{2}{3}\right)}\right) \tag{79}$$

This process results in a symmetric preconditioner P_{mus}^{-1} that can be used effectively with the PCG method to accelerate convergence:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k P_{mus}^{-1} \mathbf{r}^{(k)} \qquad k \geq 0 \tag{80}$$

6.2.3 Many Overlapping Subdomains

When applying the Schwarz method, using many overlapping subdomains can increase parallelism. This approach can be more efficient than just using a two-domain algorithm, especially for large-scale problems.

To achieve a **higher degree of parallelism** with the Schwarz method, we can **apply the two-domain algorithm recursively or use many subdomains**. If there are N overlapping subdomains, we define matrices R_i and A_i for each subdomain i (where $i = 1, \ldots, N$). The **Additive Schwarz preconditioner** then takes the form:

$$P_{ad}^{-1} = \sum_{i=1,\ldots,N} R_i^T A_i^{-1} R_i \tag{81}$$

Generalization and Convergence Issues

The **generalization of the block-Jacobi iteration** to many subdomains is highly parallel but **not algorithmically scalable because the convergence rate degrades as N increases**. To restore the convergence rate, a **coarse grid correction is used to provide global coupling**. The updated **Additive Schwarz preconditioner**, including the coarse grid correction, takes the form:

$$P_{ad} = \sum_{i=0,\ldots,N} R_i^T A_i^{-1} R_i \tag{82}$$

Multiplicative Schwarz Iteration

The Multiplicative Schwarz iteration for p domains is defined similarly to the two-domain case but extended to p subdomains. Like the classical Gauss-Seidel method (compared to Jacobi), the **Multiplicative Schwarz method has a faster convergence rate than the corresponding Additive Schwarz method**. However, it **still requires a coarse grid correction to remain scalable**. A **limitation** of the Multiplicative Schwarz method **is the lack of parallelism since p subproblems must be solved sequentially per iteration**. Parallelism can be introduced by *coloring* subdomains to identify independent subproblems that can be solved simultaneously, similar to how parallelism is introduced in Gauss-Seidel.

6.2.4 Coloring Technique

The **Multiplicative Schwarz method** is inherently sequential because each subdomain's solution depends on the updated solution of the previous subdomain. **To introduce parallelism, we use a subdomain coloring mechanism**. This method **identifies independent subdomains that can be processed simultaneously by assigning different colors to different subdomains**.

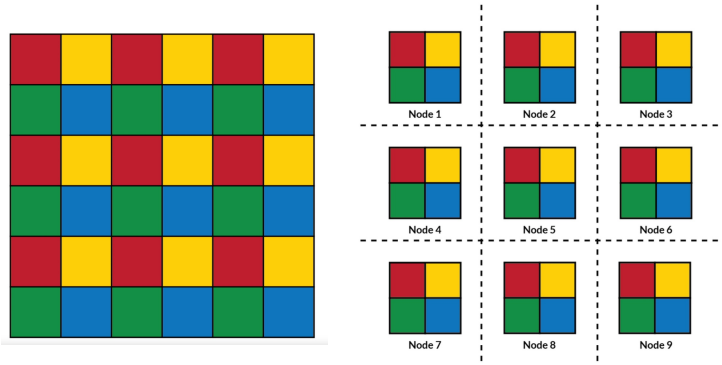
How does it work?

Sequential Nature of Multiplicative Schwarz

- **Sequential Computation**: The Multiplicative Schwarz preconditioner **operates sequentially**, meaning each subdomain is processed one after another.
- **Limitation**: This **serial nature limits the degree of parallelism, especially if there are few subdomains per color**. To overcome this, we use a coloring mechanism.

Subdomain Coloring

- **Coloring Mechanism**: Subdomain coloring identifies independent subdomains that can be solved concurrently. **Each color represents a group of subdomains that can be processed in parallel without dependency conflicts**.



A visual representation of a grid with different colored squares (red, green, blue, yellow) and smaller grids labeled as Node 1 to Node 9. This visual aid demonstrates how subdomains are divided and colored for parallel processing.

Advantages and Limitations

- **Degree of Parallelism**: The **degree of parallelism depends on the number of subdomains per color**. If there are few subdomains per color, the parallelism is limited.

- ✔ **Convergence Rate**: In general, the **Multiplicative Schwarz method converges faster than the Additive Schwarz method**. However, the **Additive Schwarz method can achieve better parallel speedup**.
- ✔ **Overall Comparison**: The coloring technique balances the need for parallelism with the inherently sequential nature of the Multiplicative Schwarz method, ensuring faster convergence while maintaining some level of parallel computation.

This section focuses on **enhancing parallelism in the Multiplicative Schwarz method by using a subdomain coloring mechanism**. This technique allows independent subdomains to be processed simultaneously, improving efficiency and convergence rates. However, the degree of parallelism is limited by the number of subdomains per color. Nevertheless, the Multiplicative Schwarz method typically converges faster than the Additive Schwarz method, although the latter may provide better parallel speedup

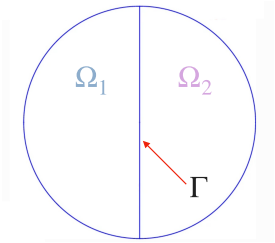
6.3 Non-Overlapping Subdomains

6.3.1 Introduction

When dealing with **Non Overlapping Subdomains**, the computational **domain is divided into adjacent subdomains that share a common boundary, but do not overlap**. This approach simplifies the handling of subdomain interfaces and ensures a clear distinction between different regions of the domain.

Key Characteristics of Non-Overlapping Subdomains

1. **Adjacent Subdomains**. The domain is divided into subdomains Ω_1 and Ω_2 that are *adjacent* to each other. These subdomains touch each other *only* along their common boundary Γ, but do *not overlap*.

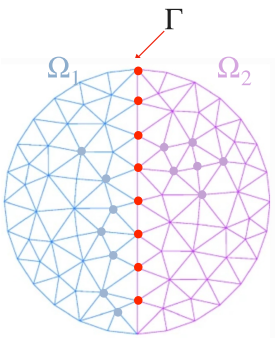


2. **Partitioning Indices**. Each subdomain has its own set of interior nodes. The indices of these nodes are divided as follows:

   - S_1: Indices of interior nodes in Ω_1.
   - S_2: Indices of interior nodes in Ω_2.

   The nodes that lie on the boundary Γ between the two subdomains are indexed by:

   - S_Γ: Indices corresponding to the interface nodes along Γ.



3. **Zero Blocks**. Nodes in Ω_1 are not directly connected to nodes in Ω_2; they are connected only through the interface nodes in Γ. This results in zero blocks in the off-diagonal positions of the matrix for direct connections between Ω_1 and Ω_2.

4. **Symmetric Block Linear System**. The matrix A and the solution vector x are partitioned into blocks corresponding to Ω_1, Ω_2 and Γ:

$$\begin{bmatrix} A_{11} & 0 & A_{1\Gamma} \\ 0 & A_{22} & A_{2\Gamma} \\ A_{1\Gamma}^T & A_{2\Gamma}^T & A_{\Gamma\Gamma} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_\Gamma \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_\Gamma \end{bmatrix}$$

   - A_{11}: Matrix block corresponding to the interior nodes in Ω_1.
   - A_{22}: Matrix block corresponding to the interior nodes in Ω_2.
   - $A_{1\Gamma}$ and $A_{2\Gamma}$: Matrix blocks corresponding to the connections between the interior nodes of Ω_1 and Ω_2 with the interface nodes in Γ.
   - $A_{\Gamma\Gamma}$: Matrix block corresponding to the interface nodes in Γ.

   The right-hand-side vector $\mathbf{b}$ is similarly partitioned into $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_\Gamma$, corresponding to the components in each subdomain and the interface.

5. **Algorithmic Approach**. When using non-overlapping subdomains, the **main goal is to solve the PDE in each subdomain independently while ensuring the interface conditions are satisfied**. Iterative solvers can be employed to update the solution at each step, ensuring that boundary conditions between subdomains are consistently respected.

6.3.2 The Schur Complement

The **Schur complement** is a powerful tool used in numerical linear algebra to **simplify the solution of linear systems involving block matrices**. When dealing with non-overlapping subdomains, the Schur complement allows us to **reduce the size of the system we need to solve directly by focusing on the interface unknowns**.

How does it work?

We introduce the Schur complement by the block LU factorization of a matrix A. The matrix A is partitioned into blocks corresponding to the subdomains Ω_1, Ω_2, and the interface Γ:

$$A = \begin{pmatrix} A_{11} & 0 & A_{1\Gamma} \\ 0 & A_{22} & A_{2\Gamma} \\ A_{1\Gamma}^T & A_{2\Gamma}^T & A_{\Gamma\Gamma} \end{pmatrix}$$

The factorization is expressed as:

$$A = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{1\Gamma}^T & A_{2\Gamma}^T & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 & A_{1\Gamma} \\ 0 & A_{22} & A_{2\Gamma} \\ 0 & 0 & S \end{pmatrix}$$

Where S is the **Schur complement**:

$$S = A_{\Gamma\Gamma} - A_{1\Gamma}^T A_{11}^{-1} A_{1\Gamma} - A_{2\Gamma}^T A_{22}^{-1} A_{2\Gamma}$$

To **solve for the interface unknowns** $\mathbf{x}_\Gamma$, we use the Schur complement system:

$$S\mathbf{x}_\Gamma = \widetilde{\mathbf{b}}_\Gamma$$

Where $\widetilde{\mathbf{b}}_\Gamma$ is defined as:

$$\widetilde{\mathbf{b}}_\Gamma = \mathbf{b}_\Gamma - A_{1\Gamma}^T A_{11}^{-1} \mathbf{b}_1 - A_{2\Gamma}^T A_{22}^{-1} \mathbf{b}_2$$

Once $\mathbf{x}_\Gamma$ is determined, the **remaining unknowns** in Ω_1 and Ω_2 can be computed as follows:

$$\begin{aligned} \mathbf{x}_1 &= A_{11}^{-1}(\mathbf{b}_1 - A_{1\Gamma}\mathbf{x}_\Gamma) \\ \mathbf{x}_2 &= A_{22}^{-1}(\mathbf{b}_2 - A_{2\Gamma}\mathbf{x}_\Gamma) \end{aligned}$$

Key Characteristics of Schur Complement

- ✗ **Computational Expense**. The Schur complement matrix S is expensive to compute and is generally dense, even if A is sparse.
- ✓ **Iterative Solution**. If the Schur complement **system** $S\mathbf{x}_\Gamma = \widetilde{\mathbf{b}}_\Gamma$ is **solved iteratively**, the Schur **complement** S **does not need to be formed explicitly**.
- ✓ **Matrix-Vector Multiplication**. Matrix-vector multiplication by S requires solving systems within each subdomain, implicitly involving A_{11}^{-1} and A_{22}^{-1}, which can be done in parallel.

1. **Condition Number**. The condition number[6] of S is generally **better than that of** A, typically $O\left(h^{-1}\right)$ instead of $O\left(h^{-2}\right)$ for a mesh size h.
2. **Preconditioning**. In practice, suitable interface preconditioners are needed to accelerate convergence when solving the Schur complement system.

The Schur complement allows us to **simplify the solution of linear systems involving block matrices by focusing on the interface unknowns**. This **reduces the size of the system** that needs to be solved directly. The process involves computing the Schur complement matrix S and solving the Schur complement system for the interface unknowns. Although computing S can be expensive, **iterative methods and parallel computation can help manage this complexity**.

[6] 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.

6.3.3 Many Non-Overlapping Subdomains

To improve parallelism in solving large linear systems, we use many non-overlapping subdomains. This section explains how to partition matrices and right-hand vectors and solve the resulting system using the Schur complement method. In other words, the goal is to **improve parallelism by using many non-overlapping subdomains with the Schur method**.

How does it work?

- **Definitions**.
  - Let I be the set of indices of **interior nodes of subdomains**.
  - Let Γ be the set of indices of **interface nodes**.
- **Discrete Linear System**. The system is represented in block form as:

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{II\Gamma}^T & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ b_\Gamma \end{pmatrix}$$

  Here, A_{II} is block diagonal with the structure:

$$A_{II} = \begin{pmatrix} A_{11} & 0 & \cdots & 0 \\ 0 & A_{22} & \cdots & 0 \\ 0 & \vdots & \ddots & 0 \\ 0 & \cdots & 0 & A_{NN} \end{pmatrix}$$

- **Block LU Factorization**.
  - The factorization of matrix A yields a system:

$$S\mathbf{x}_\Gamma = \widetilde{\mathbf{b}}_\Gamma$$

  - The Schur complement matrix S is given by:

$$S = A_{\Gamma\Gamma} - A_{I\Gamma}^T A_{II}^{-1} A_{I\Gamma}$$

    And:

$$\widetilde{\mathbf{b}}_\Gamma = \mathbf{b}_\Gamma - A_{I\Gamma}^T A_{II}^{-1} \mathbf{b}_I$$

- **Solution Method**.
  - This system can be **solved iteratively without forming S explicitly**.
  - Suitable interface **preconditioners can be used to accelerate convergence**.
  - **Interior unknowns** are then given by:

$$x_I = A_{II}^{-1}(\mathbf{b}_I - A_{I\Gamma}\mathbf{x}_\Gamma)$$

  - All occurrences of A_{II}^{-1} can be **performed on all subdomains in parallel** because A_{II} is block diagonal.

7 Direct Methods for Linear Systems

This section provides a brief introduction to direct methods for linear systems of equations.

7.1 LU Factorization

LU Factorization is a **method** used in numerical linear algebra **to decompose a square matrix A into two factors**:

- A lower unitary triangular matrix L;
- An upper triangular matrix U, such that $A = LU$.

How does it work?

1. **Principal Submatrices**. Let $A_{p,i} \in \mathbb{R}^{i,i}, i = 1, \dots, n$ be the principal submatrices of A obtained by considering the first i rows and columns. These submatrices are **crucial in determining the existence of an LU factorization**.
2. **Determinant Condition**. For LU factorization to be possible, the determinant of each principal submatrix must be non-zero:

$$\det(A_{p,i}) \neq 0 \qquad \forall i = 1, \dots, n-1$$

3. **LU Factorization Theorem**.

   Theorem 11. *If A is invertible, then it admits an LU factorization if and only if all its leading principal minors[7] are nonzero.*

[7] A **Leading Principal Minor** is the determinant of a top-left submatrix of A.
For a square matrix $A \in \mathbb{R}^{n \times n}$:

- The first leading principal minor is the determinant of the 1×1 submatrix that includes only the first element of A:
$$\Delta_1 = \det(A_{1,1})$$
- The second leading principal minor is the determinant of the 2×2 submatrix that includes the first two rows and columns of A:
$$\Delta_2 = \det\left(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\right)$$
- The k-th leading principal minor is the determinant of the $k \times k$ submatrix that includes the first k rows and columns of A:
$$\Delta_k = \det(A_{1:k,1:k})$$
This pattern continues until $k = n$, where n is the size of the matrix A.

7.2 Gaussian elimination

Gaussian elimination is a method used to solve systems of linear equations. It involves a **sequence of operations**:

- **Swapping rows**: reorder the rows to position the pivot elements (non-zero leading coefficients) appropriately.
- **Multiplying a row by a nonzero number**: scale rows to facilitate elimination.
- **Adding a multiple of one row to another row**: use row operations to eliminate variables systematically.

The **goal** is to **transform the matrix A into an upper triangular matrix** $A^{(n)} = U$. Besides transforming the matrix A, the right vector $\mathbf{b}$ undergoes similar operations to become:

$$\begin{aligned} A^{(1)} &= A \to A^{(2)} \to \cdots \to A^{(k)} \to A^{(k+1)} \to \cdots \to A^{(n)} = U \\ \mathbf{b}^{(1)} &= \mathbf{b} \to \mathbf{b}^{(2)} \to \cdots \to \mathbf{b}^{(k)} \to \mathbf{b}^{(k+1)} \to \cdots \to \mathbf{b}^{(n)} = \mathbf{y} \end{aligned}$$

Algorithm

The **main goal** is to transform the matrix A into an upper triangular form U while transforming the constant vector b.

1. For each **pivot position** k from 1 to $n-1$: $k = 1, \ldots, n-1$;
2. For each **row** i below pivot position k: $i = k+1, \ldots, n$;
3. **Compute Multiplier** l_{ik}:
$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$$
Ensure the pivot element a_{kk} is non-zero (swap rows if necessary).
4. For each **column** j to the right of the pivot element: $j = k+1, \ldots, n$;
5. **Update Matrix Elements**:
$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$$
6. **Update the Constants Vector**:
$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$$

After completing n steps, the matrix A is transformed into an upper triangular matrix U, and we get the transformed vector y:

$$A^{(n)} = U \qquad l_{ij} \to L \qquad b^{(n)} = y$$

```
1  For  k = 1, ..., n − 1
2
3      For  i = k + 1, ..., n
4
5          l_ik = a_ik^(k) / a_kk^(k)
6
7          For  j = k + 1, ..., n
8
9              a_ij^(k+1) = a_ij^(k) − l_ik a_kj^(k)
10
11         b_i^(k+1) = b_i^(k) − l_ik b_k^(k)
```

Listing 1: Gaussian Elimination Pseudo-Code

$ How much does it cost?

Gaussian elimination involves several steps, each with different computational requirements. We **quantify these steps in terms of floating point operations** (flops). For any step k from 1 to n:

Row 5. **Pivot Selection and Division**. This step involves selecting the pivot element and performing the necessary division to normalize the row. This process **requires** $n-k$ **flops**.

Row 9. **Row Operations**. **To eliminate the elements below the pivot**, we need $2\,(n-k)^2$ **flops** for the necessary multiplications and subtractions.

Row 11 **Update the Constants Vector**. The **total flops** for each step is $2\,(n-k)$.

Overall Computational Cost. Summing the costs over all steps (from $k=1$ to $n-1$):

$$\sum_{k=1}^{n-1} \left(2 \cdot (n-k)^2 + 3 \cdot (n-k)\right) = \sum_{p=1}^{n-1} \left(2 \cdot p^2 + 3 \cdot p\right)$$

This can be simplified to:

$$2 \cdot \frac{n \cdot (n-1) \cdot (2 \cdot n - 1)}{6} + 3 \cdot \frac{n \cdot (n-1)}{2} \approx \frac{2}{3} n^3$$

Sufficient conditions for Gaussian elimination

There are **two sufficient conditions** for Gaussian elimination to successfully reduce a matrix to row echelon form without encountering any issues like division by zero or numerical instability:

✔ **Strict Diagonal Dominance**.

- A matrix A is strictly diagonally dominant by rows if:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \qquad i = 1, \ldots, n$$

- Similarly, A is strictly diagonally dominant by columns if:

$$|a_{ii}| > \sum_{j \neq i} |a_{ji}|, \qquad i = 1, \ldots, n$$

This condition **ensures that the pivot elements are sufficiently large**, preventing division by small numbers and minimizing numerical errors.

✔ **Symmetric Positive Definite (SPD) Matrix**. A matrix A is symmetric positive definite if for all non-zero vectors $z \in \mathbb{R}^n$:

$$\forall z \in \mathbb{R}^n \qquad z \neq 0 \qquad z^T A z > 0$$

This condition **guarantees that the matrix has positive eigenvalues**, **ensuring stability during the elimination process** and **that the matrix is invertible**.

7.3 Cholesky Factorization

The **Cholesky factorization** is used to **decompose a symmetric positive definite (SPD) matrix A into a product of a lower triangular matrix and its transpose**.

Definition 1: Cholesky factorization

For an SPD matrix A of order n, there exists a unique upper triangular matrix R with real and positive diagonal entries such that:

$$A = R^T R$$

Algorithm

The algorithm to compute the Cholesky factorization involves the following steps:

1. Initialize r_{11}:
$$r_{11} = \sqrt{a_{11}}$$
2. For each column j from 2 to n:
   - Calculate the **off-diagonal elements** r_{ij} for $i = 1, \ldots, j-1$:
$$r_{ij} = \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj} \right)$$
   - Calculate the **diagonal element** r_{jj}:
$$r_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2}$$

How much does it cost?

The Cholesky factorization has a computational **cost of approximately**:

$$\frac{n^3}{3}$$

Floating-point operations. This factorization is particularly efficient for SPD matrices and is often used in numerical analysis and applications requiring matrix decompositions.

7.4 Pivoting

Problem

If, during the k-th step of Gaussian elimination, the **pivot element $a_{kk}^{(k)}$ is zero**, we cannot proceed with the standard elimination process because division by zero is undefined.

Solution

Pivoting is a **set of techniques used to prevent division by zero in Gaussian elimination**, but also to **provide numerical stability** and **minimize rounding errors**. There are several types of pivoting: *partial pivoting* (pivoting by rows), *complete pivoting*.

The pivoting strategy can be interpreted as pre-multiplying the original matrix A (and the constants vector b) by a permutation matrix P. The permutation matrix P reorders the rows of A and b to place a suitable pivot element in the k-th row.

$$A\mathbf{x} = \mathbf{b} \;\Rightarrow\; PA\mathbf{x} = P\mathbf{b} \;\Rightarrow\; LU\mathbf{x} = P\mathbf{b} \;\Rightarrow\; \begin{cases} L\mathbf{y} = P\mathbf{b} \\ U\mathbf{x} = \mathbf{y} \end{cases}$$

7.4.1 Pivoting by rows (partial pivoting)

Purpose and Goal

- **Prevent Division by Zero**. Ensure that the pivot element is non-zero.
- **Improve Numerical Stability**. Minimize rounding errors and improve the accuracy of the solution.

The **goal** is to ensure that the pivot element is the largest possible value in the current column.

Algorithm

For each column k (from the first to the last):

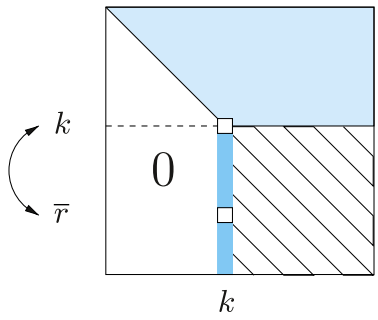
1. **Identify the Pivot Element**. **Find the row** i (where $i > k$) **with the largest absolute value in the pivot column**.

   This ensures that the selected pivot element a_{kk} is the largest possible, reducing the potential for numerical instability. From a mathematical point of view:

   $$\bar{i} \; : \; \left|a_{\bar{r}k}^{(k)}\right| = \max_{i>k} \left|a_{ik}^{(k)}\right|$$

2. **Swap Rows**. Swap row k with row i to move the largest pivot element to the diagonal position (a_{kk}). This is achieved by pre-multiplying the matrix A by a permutation matrix P, which swaps the rows in the identity matrix I.

3. **Update the Matrix and Vector**. Perform the standard row operations to eliminate the elements below the pivot element, resulting in an upper triangular matrix. Update the constants vector accordingly.



7.4.2 Complete Pivoting

Goal

Maximize numerical stability by selecting the largest possible pivot element in the entire submatrix. This ensures the most accurate and stable results during matrix factorization.

Algorithm

1. **Identify the Largest Element**. For each step k, identify the largest absolute value in the remaining submatrix. This element becomes the **pivot**.
2. **Swap Rows and Columns**.
   - Swap the current row k with the row containing the largest element.
   - Swap the current column k with the column containing the largest element.

Mathematical Representation

Complete pivoting involves two permutation matrices, P and Q:

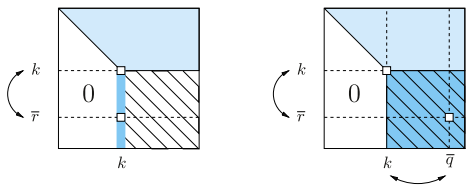
$$PAQ = LU$$

- P is the permutation matrix for row swaps.
- Q is the permutation matrix for column swaps.
- L is a lower triangular matrix.
- U is an upper triangular matrix.

Therefore, solving the system:

1. Transform the original system $A\mathbf{x} = \mathbf{b}$ into $PAQ\mathbf{x} = P\mathbf{b}$.
2. Substitute $Q\mathbf{x}$ with $\mathbf{x}^*$:

$$\begin{aligned} PAQ\mathbf{x} = P\mathbf{b} \quad &\Longrightarrow \quad L\left(U\mathbf{x}^*\right) = P\mathbf{b} \\ Ly = P\mathbf{b} \quad &\Longrightarrow \quad U\mathbf{x}^* = y \implies \mathbf{x} = Q\mathbf{x}^* \end{aligned}$$



7.5 Fill-In

What is the fill-in?

When performing LU decomposition on a sparse matrix A, the resulting matrices L (lower triangular) and U (upper triangular) may have non-zero elements in positions that were originally zero in A. This phenomenon is called fill-in.

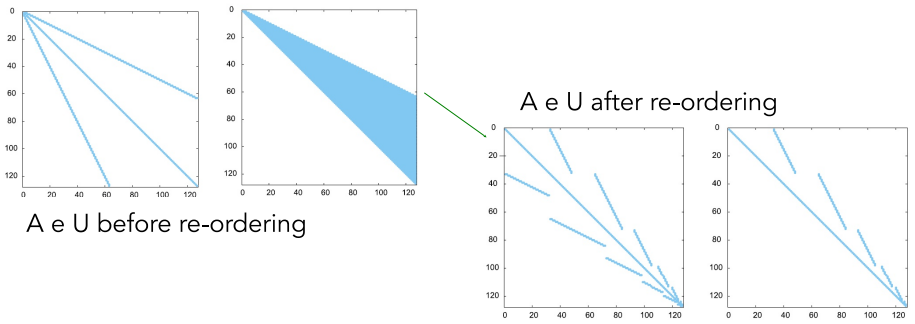
Impact

Fill-in **increases the storage requirements and computational complexity** of the decomposition.

Fill-In Reduction Strategies

To reduce fill-in, reordering techniques are employed. This involves renumbering the rows of A differently to minimize the number of non-zero elements in L and U.

- **Permutation Matrix P.** Reordering can be achieved by pre-multiplying A by a permutation matrix P, which swaps rows to achieve a more favorable structure.



8 Exams

This section contains the solutions to the exams for the Numerical Linear Algebra courses. I created this section to practice for the exam, so there may be errors even though I checked the official solutions.

8.1 Year 2024

8.1.1 July 03

Exercise 1 - Theory

1. **Consider the following problem: find $\mathbf{x} \in \mathbb{R}^n$, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$ are given. State under which conditions the mathematical problem is well posed.**

   Answer: To determine when the problem of finding $\mathbf{x} \in \mathbb{R}^n$ such that $A\mathbf{x} = \mathbf{b}$ (where $A \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$ are given) is well-posed, we need to consider the following conditions:

   (a) ***Existence***: There must be **at least one solution** to the equation $A\mathbf{x} = \mathbf{b}$. For this to happen, the vector $\mathbf{b}$ should be in the column space of A, which means A **must have full rank**, i.e., $\text{rank}\,(A) = n$.

   (b) ***Uniqueness***: There must be at most one solution. For the solution to be unique, the matrix A must be invertible. This is equivalent to saying that A has **full rank and its determinant is non-zero** ($\det(A) \neq 0$).

   (c) ***Stability***: The solution should change continuously with changes in $\mathbf{b}$. In other words, the problem should be well-conditioned. The **condition number of A should be reasonably small** to ensure that small changes in $\mathbf{b}$ do not cause large changes in the solution $\mathbf{x}$.

   Thus, the mathematical problem is well-posed if:

   - The matrix A is **square**;
   - The matrix A is **invertible**;
   - The matrix A has a **reasonably low condition number**, ensuring the existence, uniqueness, and stability of the solution.

   Tip

   To remember when a mathematical problem is well posed, it can be helpful to remember that:

   - A solution must exist, otherwise we have a problem not well posed, simple.
   - Ok, if the solution exists, how many? We mean, at least one solution should exist, right? So the existence of the solution has to be guaranteed. This can be achieved by taking advantage of the full rank of the matrix. But at this point the student's question should be: "*Why the full rank?*". Well, the answer is simple: invertible. If the matrix is invertible, we have the guarantee that there exists an inverse matrix such that $AA^{-1} = I$, where I is the identity matrix. This also tells us that the equation $A\mathbf{x} = \mathbf{b}$ has a unique solution given by $\mathbf{x} = A^{-1}\mathbf{b}$.
   - Ok, but a system with n solutions is not much information.

Yes, it is interesting to know that we can find almost one solution when we try to solve the system, but it should be very helpful to know that where we get the solution, it is unique. Again, "*why do we need to compute the determinant?*". A partial answer comes from the previous point, if the matrix is invertible, we have the mathematical guarantee that the determinant is not zero, therefore the inverse matrix exists and is uniqueness. Finally, an interesting implication can be observed:

$$\text{rank}\,(A) = n \;\Rightarrow\; \text{invertible} \;\Rightarrow \det(A) \neq 0$$

- Finally, the mathematical problem is well-posed if the condition number is small, so that the small changes in $\mathbf{b}$ do not cause large changes in the solution $\mathbf{x}$.

2. Describe the Cholesky factorization and how it is used to approximately solve the above linear system.

Answer: Cholesky factorization is a numerical method for decomposing a symmetric, positive-definite (SPD) matrix A into a product of a lower triangular matrix L and its transpose L^T. Specifically, if A is symmetric and positive-definite, then there exists a unique lower triangular matrix L such that:

$$A = LL^T$$

We also provide the algorithm, but it might not be necessary. However, the algorithm is:

- Start with an empty lower triangular matrix L.
- For $i = 1$ to n (where n is the dimension of the matrix A):
  - Compute the diagonal element:

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

  - For $j = i + 1$ to n:
    - Compute the off-diagonal elements:

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ji} - \sum_{k=1}^{i-1} L_{jk} L_{ik} \right)$$

About the previous and classical linear system $A\mathbf{x} = \mathbf{b}$, we can solve it using the Cholesky factorization:

(a) Decompose A as LL^T using Cholesky factorization.

(b) Since L is lower triangular, use **forward substitution**. Solve the triangular system $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$.

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij} y_j}{L_{ii}}$$

(c) Since L^T is upper triangular, use **backward substitution**. Solve the triangular system $L^T\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} L_{ji} x_j}{L_{ii}}$$

3. State under which conditions the Cholesky factorization can be use.

Answer: Cholesky factorization can be used under the following conditions:

(a) ***Symmetric Matrix***: The matrix A must be symmetric, meaning $A = A^T$. This ensures that the matrix is equal to its transpose and guarantees two important properties:

- Symmetric matrices are required because Cholesky factorization is designed to work with positive-definite matrices. A symmetric matrix that is also positive-definite ensures that all eigenvalues are positive, making the square root operation (used in the factorization) valid and yielding real numbers. This property is crucial for the numerical stability and accuracy of the factorization.
- The symmetry of the matrix guarantees that the Cholesky factorization, if it exists, is unique. This uniqueness means that for a given symmetric, positive-definite matrix A, there is exactly one lower triangular matrix L such that $A = LL^T$.

(b) ***Positive-Definite Matrix***: The matrix A must be positive definite. This means that for any non-zero vector $\mathbf{y} \in \mathbb{R}^n$, $\mathbf{y}^T A\mathbf{y} > 0$. Positive definiteness ensures that all eigenvalues of A are positive.

When these two conditions are met, the Cholesky factorization decomposes A into a product of a lower triangular matrix L and its transpose L^T (i.e., $A = LL^T$).

If these conditions are not met, Cholesky factorization is not applicable, and other factorization methods like LU decomposition might be used instead.

4. Report the main theoretical result and comment on the main differences with respect to the LU factorization.

Answer: The main differences between Cholesky factorization and LU factorization are

- ***Matrix Requirements***.
  - **Cholesky**: Requires the matrix A to be symmetric and positive-definite (SPD). This means all eigenvalues of A must be positive.
  - **LU**: Can be applied to any square matrix A (even not SPD). However, partial pivoting may be necessary for numerical stability, but it is not a requirement!
- ***Factorization Form***.
  - **Cholesky**: Decomposes A into LL^T, where L is a lower triangular matrix and A^T is an upper triangular matrix.
  - **LU**: Decomposes A into LU, where L is a lower triangular matrix and U is an upper triangular matrix.
- ***Complexity and Efficiency***.
  - **Cholesky**: More efficient for SPD matrices as it exploits symmetry, leading to approximately half the computational effort compared to LU factorization. Specifically, Cholesky factorization has a time complexity of $O\left(\frac{n^3}{3}\right)$.
  - **LU**: More general and can handle more matrix types, but typically requires more computational effort, with a time complexity of $O\left(2 \cdot \frac{n^3}{3}\right)$ when performed with partial pivoting.
- ***Stability***.
  - **Cholesky**: Numerically stable for SPD matrices. It avoids the potential pitfalls[8] of large pivot elements that can arise in LU factorization.
  - **LU**: May require partial or full pivoting to ensure numerical stability, particularly for matrices that are not well-conditioned.

In summary, Cholesky factorization is a specialized, efficient method for symmetric, positive-definite (SPD) matrices, while LU factorization is a more general approach applicable to a broader range of matrices, but potentially less efficient and stable without pivoting.

[8] In the context of numerical linear algebra, **pitfalls** refer to potential problems or challenges that can arise during the factorization process. These pitfalls can lead to numerical instability or inaccuracies in the computed solutions. Specifically, for LU factorization, pitfalls include issues like division by small numbers, round-off errors, ill-conditioned matrices, and zero or near-zero pivot elements. Pivoting strategies are often employed to mitigate these pitfalls and improve the robustness of the factorization.

Exercise 1 - Laboratory

The exam text provides the following matrix:

The solution code snippet is available here:

1. **Download the sparse matrix `Aex1.mtx` from the Exam folder and save it. Load the matrix in a new file `exer1.cpp`. Report on the sheet the matrix size and the Euclidean norm of `Aex1.mtx`. Is the matrix symmetric?**

   Answer:

```cpp
#include <Eigen/Sparse>
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Load the matrix
    Eigen::SparseMatrix<double> A;
    loadMarket(A, "Aex1.mtx");

    // Matrix size
    printf("Matrix size: %ldX%ld\n", A.rows(), A.cols());
    // The Euclidean norm is calculated by subtracting the
    // transpose of A from A;
    // This is done to check also if the matrix is symmetric
    // since A = A.t if it is symmetric;
    // However it can be calculated directly by A.norm()
    // if we want to check the magnitude of the matrix
    const Eigen::SparseMatrix<double> A_transpose = Eigen::
    SparseMatrix<double>(A.transpose());
    const Eigen::SparseMatrix<double> B = A_transpose - A;
    printf("The Euclidean norm of A-A.t: %g\n", B.norm());
    printf("The Euclidean norm of A: %g\n", A.norm());
}
```

   And the result is:

```
Matrix size: 256X256
The Euclidean norm of A-A.t: 5.24291e-15
The Euclidean norm of A: 117.201
```

2. Define an `Eigen` vector $\mathbf{x}^* = (1, 0, 1, 0, \ldots, 0)^T$ with size equal to the number of columns of `Aex1.mtx`. Compute $\mathbf{b} = Ax^*$ and report on the sheet $||\mathbf{b}||$.

Answer:

```
// Create a row vector x star: 1,0,1,0,...,0
// Use ColMajor to characterize the vector as a column vector
// (since the exercise asks for a transpose)
Eigen::SparseVector<double, Eigen::ColMajor> x_star(A.cols());
for (int i = 0; i < A.cols() / 2; i++) {
    // start with 0 and put 1.0 every 2 steps
    x_star.insert(2 * i) = 1.0;
}
// Compute b = Ax*
const Eigen::VectorXd b = A * x_star;
printf("Norm of b: %g\n", b.norm());
```

And the result is:

```
Norm of b: 51.2307
```

3. Solve the linear system $A\mathbf{x} = \mathbf{b}$ using the Cholesky decomposition method available in the Eigen library. Report on the sheet the norm of the absolute error.

Answer:

```
// Create the Cholesky decomposition solver
Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>>
    choleskySolver;
// Try to compute the Cholesky decomposition
choleskySolver.compute(A);
// Check if the decomposition was successful
if (choleskySolver.info() != Eigen::Success) {
    printf("Cannot factorize the matrix\n");
    return 0;
}
// The decomposition was successful, solve the system
Eigen::VectorXd x = choleskySolver.solve(b);
// Calculate the absolute error
printf("Solved the system Ax = b using Cholesky decomposition\
    n");
printf("The norm of the absolute error is: %g\n", (x - x_star)
    .norm());
```

And the result is:

```
Solved the system Ax = b using Cholesky decomposition
The norm of the absolute error is: 4.43266e-14
```

4. Solve the previous linear system using the Gradient method implemented in the `grad.hpp` template combined with the diagonal preconditioner provided by Eigen. Set a tolerance of 10^{-7} and take $\mathbf{x}_0 = (0, 0, \ldots, 0)^T$ as initial guess. Report on the sheet the iteration counts and the norm of the absolute error at the last iteration.

The `grad.hpp` file can be found here:

Answer:

```cpp
#include "headers/grad.hpp"

int main() {
    /// ...
    // Set the tolerance, the maximum iterations (avoid infinite loop)
    double tol = 1.e-7; // 10^-7
    int max_iter = 5000;
    // Create an initial guess of zeros
    Eigen::VectorXd x_gradient = Eigen::VectorXd::Zero(A.cols());
    // Create the diagonal preconditioner
    Eigen::DiagonalPreconditioner<double> diagonalPreconditioner(A);
    // Solve the system using the Gradient method
    int result = LinearAlgebra::GRAD(A, x_gradient, b, diagonalPreconditioner, max_iter, tol);
    // The number of iterations performed is overwritten
    // by the function on the maxIterations variable
    printf("The Gradient method has converged? %s", result ? "No\n" : "Yes\n");
    printf("The number of iterations performed: %d\n", max_iter);
    // The solution is stored in x_gradient
    printf("The absolute error is: %g\n", (x_gradient - x_star).norm());
    // Finally, just for curiosity, print the tolerance achieved
    printf("The tolerance achieved is: %g\n", tol);
}
```

And the result is:

```
The Gradient method has converged? Yes
The number of iterations performed: 4788
The absolute error is: 0.000235242
The tolerance achieved is: 9.96454e-08
```

5. Repeat the previous point using the Conjugate Gradient method implemented in the `cg.hpp` template combined with the diagonal preconditioner provided by Eigen. Set the solution obtained at the previous point as initial guess. Report the iteration counts (required to achieve a precision of 10^{-14}) and the norm of the absolute error at the last iteration.

The `cg.hpp` file can be found here:

Answer:

```cpp
#include "headers/cg.hpp"

int main() {
    // Set the tolerance,
    // the maximum iterations (avoid infinite loop)
    double tol_CG = 1.e-14; // 10^-14
    int max_iter_CG = 100;
    // Create the diagonal preconditioner
    Eigen::DiagonalPreconditioner<double>
    diagonalPreconditioner_CG(A);
    // Solve the system using the Conjugate Gradient method
    // We set the previous solution as the initial guess
    int result_CG = LinearAlgebra::CG(A, x_gradient, b,
    diagonalPreconditioner_CG, max_iter_CG, tol_CG);
    // The number of iterations performed is overwritten
    // by the function on the maxIterations variable
    printf("The Conjugate Gradient method has converged? %s",
    result_CG ? "No\n" : "Yes\n");
    printf("The number of iterations performed: %d\n",
    max_iter_CG);
    // The solution is stored in x_CG
    printf("The absolute error is: %g\n", (x_gradient - x_star
    ).norm());
    // Finally, just for curiosity, print the tolerance
    achieved
    printf("The tolerance achieved is: %g\n", tol_CG);
}
```

And the result is:

```
The Conjugate Gradient method has converged? Yes
The number of iterations performed: 7
The absolute error is: 2.8326e-13
The tolerance achieved is: 8.94393e-15
```

Exercise 2 - Theory

1. **Consider the rectangular linear system $Ax = b$, where A is an $m \times n$ matrix, $m \geq n$. Provide the definition of the solution in the least-square sense and state under which condition the problem is well-posed.**

   Answer: The solution method least-square finds an approximate solution of the overdetermined linear systems, by minimizing the sum of the squares of the residuals (difference between the left and the right sides of the equation):

   $$\Phi(\mathbf{x}^*) = \min_{\mathbf{y} \in \mathbb{R}^n} \Phi(\mathbf{y})$$

   Where:

   $$\Phi(\mathbf{y}) = \|A\mathbf{x} - \mathbf{b}\|_2^2$$

   An overdetermined linear system is a system of linear equations in which there are more equations than unknowns.

   The solution $\mathbf{x}^*$ can be found by imposing the condition that the gradient of the function $\Phi(\cdot)$ must be equal to zero at $\mathbf{x}^*$:

   $$\nabla\Phi(\mathbf{y}) = 2A^T A\mathbf{y} - 2A^T\mathbf{b}$$

   From which it follows that $\mathbf{x}^*$ must be the solution of the square system:

   $$A^T A\mathbf{x}^* = A^T\mathbf{b}$$

   Well-Posed Condition. The least-squares problem is well-posed if:

   - The matrix $A \in \mathbb{R}$ has full column rank $\text{rank}(A) = n$. This means that the matrix A has a solution and it is unique.

   The full rank is very important because it guarantees us that there is a least-square solution and it is unique.

   Tip

   It is not necessary to remember every equation. It may be enough to remember what the least squares method is (not necessary what an overdetermined linear system is), how to write mathematically and which is the system we have to solve to get the solution $\mathbf{x}^*$.

   About the gradient, it is enough to remember that starting from $\Phi(\mathbf{y})$ and applying the gradient operator (which points in the direction of the maximum rate of increase of the function), we can find a system where we can get the vector solution $\mathbf{x}^*$. We also want to use the gradient because we want the solution to converge to zero as the rate of the function increases.

   Finally, the well-posed condition is only the full rank. Because it is an essential property to guarantee that we have found the best (and unique) solution applicable.

2. Describe the QR factorization of the rectangular matrix A.

Answer: QR factorization is a decomposition of a matrix A into a product of an orthogonal (or unitary) matrix Q and an upper triangular matrix R.

Definition. Given an $m \times n$ matrix A (with $m \geq n$), the QR factorization of A is expressed as:

$$A = QR$$

Where:

- Q is an $m \times m$ orthogonal (or unitary) matrix ($Q^T Q = I$).
- R is an $m \times n$ upper triangular matrix.

Algorithm. The most basic QR decomposition method is the Classical Gram-Schmidt (CGS) process and the more stable version is the Modified Gram-Schmidt.

- Classical Gram-Schmidt (CGS) Process. Given the columns of A, $\mathbf{a}_1$, $\mathbf{a}_2, \ldots, \mathbf{a}_n$, the goal is to find orthogonal vectors $\mathbf{q}_j$.
  The CGS process involves subtracting projections of the current vector $\mathbf{a}_j$ onto all previously calculated orthogonal vectors:

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

  Normalizing to get $\mathbf{q}_j$:

$$\mathbf{q}_j = \frac{\mathbf{w}_j}{\|\mathbf{w}_j\|}$$

  We obtain QR factorization with:

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

- Modified Gram-Schmidt (MGS) Process. The CGS method can be sensitive to rounding errors, making it numerically unstable for some applications. The modification involves using a different projection step, making the process more stable:

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

When is it used? QR factorization is used when:

- Solving linear systems.
- Least-Squares Problems.

3. Discuss how the QR factorization can be employed to solve the above linear system in the least-square sense.

Answer: We need to solve the rectangular linear system $A\mathbf{x} = \mathbf{b}$ in the least-squares sense.

Given:

- A is an $m \times n$ matrix with $m \geq n$.
- $\mathbf{b}$ is an m-dimensional vector.

The goal (least-square goal) is to find the vector $\mathbf{x}$ that minimizes the residual $\|A\mathbf{x} - \mathbf{b}\|_2^2$, which is the square Euclidean norm of the difference between $A\mathbf{x}$ and $\mathbf{b}$.

(a) ***QR Factorization***. Perform QR factorization on matrix A:

$$A = QR$$

Where:

- Q is an $m \times m$ orthogonal (or unitary) matrix.
- R is an $m \times n$ upper triangular matrix.

(b) ***Simplify the problem***.

- Substitute A in the linear system with QR:

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

- Pre-multiply both sides by the transpose of Q $\left(Q^T\right)$, to simplify:

$$Q^T QR\mathbf{x} = Q^T\mathbf{b}$$

- Since $Q^T Q = I$ (because Q is invertible and orthogonal), this reduces to:

$$R\mathbf{x} = Q^T\mathbf{b}$$

(c) ***Solve the Triangular System***. The problem now is to solve the upper triangular system:

$$R\mathbf{x} = Q^T\mathbf{b}$$

Since R is an upper triangular matrix, we can use back-substitution to solve for $\mathbf{x}$.

In summary:

- *QR Factorization* decomposes A into Q (orthogonal) and R (upper triangular).
- *Simplification*: We transform the problem into solving $R\mathbf{x} = Q^T\mathbf{b}$.
- We use *back-substitution* to find $\mathbf{x}$.

Exercise 2 - Laboratory

The exam text provides the following matrix:

The solution code snippet is available here:

1. **Download the matrix `Aex2.mtx` from the Exam folder and save it. Load the matrix in a new file called `exer2.cpp` using the `unsupported/Eigen/SparseExtra` module. Report on the sheet $\|A^T A\|$.**

   Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Load matrix Aex2.mtx
    Eigen::SparseMatrix<double> A;
    loadMarket(A, "Aex2.mtx");

    // Check matrix properties (just to get an idea of the matrix)
    printf("Matrix size: %ldx%ld\n", A.rows(), A.cols());
    printf("Non zero entries: %ld\n", A.nonZeros());

    // Calculate A^T * A, that is called Gram matrix
    // Refs: https://en.wikipedia.org/wiki/Gram_matrix
    const Eigen::SparseMatrix<double> gram_matrix = A.transpose() * A;
    printf("Norm of A^T * A: %f\n", gram_matrix.norm());
    printf("Gram matrix size: %ldx%ld\n", gram_matrix.rows(), gram_matrix.cols());
    // Note, the Gram matrix is symmetric by definition
    printf("Gram Symmetry Proof: ");
    if (gram_matrix.isApprox(gram_matrix.transpose())) {
        printf("The Gram matrix is symmetric. QED\n");
    } else {
        printf("The matrix is not symmetric, so it is not a Gram matrix\n");
    }
}
```

   And the result is:

```
Matrix size: 800x400
Non zero entries: 8431
Norm of A^T * A: 335.009675
Gram matrix size: 400x400
Gram Symmetry Proof: The Gram matrix is symmetric. QED
```

2. Define an `Eigen` vector $\mathbf{b} = (A^T A)\,\mathbf{x}^*$, where $\mathbf{x}^* = (1, 1, \ldots, 1)^T$. Report on the sheet $\|\mathbf{b}\|$.

Answer:

```
// Create a vector e with all elements equal to 1;
// The size of the vector is the number of columns in the Gram matrix,
// since we are multiplying it by the Gram matrix
const Eigen::VectorXd x_star = Eigen::VectorXd::Ones(gram_matrix.cols());
// Calculate b = (A^T * A) * x_star
const Eigen::VectorXd b = gram_matrix * x_star;
printf("The norm of b: %f\n", b.norm());

```

And the result is:

```
The norm of b: 113.941047
```

3. Use the `SparseQR` solver available in the `Eigen` library to compute the approximate solution of the least-square problem associated to $A\mathbf{x} = A\mathbf{b}$. Report on the sheet the Euclidean norm of the residual $\|A\mathbf{x}_{\mathbf{SQR}} - A\mathbf{b}\|$, where $\mathbf{x}_{\mathbf{SQR}}$ is the obtained approximate solution.

Answer:

```
// Create the QR factorization solver
Eigen::SparseQR<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> qr_solver;
// Compute the QR factorization of the matrix A to solve the system of equations Ax = Ab
qr_solver.compute(A);
// Check if the factorization was successful
if (qr_solver.info() != Eigen::Success) {
    printf("Cannot factorize the matrix using QR factorization\n");
    return 0;
}
// Solve the system of equations using the QR factorization
const Eigen::VectorXd x_qr = qr_solver.solve(A * b);
// Print the norm of the difference between the solution obtained
// by the QR factorization and the expected solution
printf("\nSolution with Eigen QR:\n");
printf("Norm of the difference || Ax_SQR - Ab ||: %g\n", (x_qr - b).norm());
```

And the result is:

```
Norm of the difference || Ax_SQR - Ab ||: 3.89145e-13
```

4. Use the `LeastSquareConjugateGradient` solver available in the `Eigen` library to compute the approximate solution of the previous least-square problem up to a tolerance of 10^{10}. Report on the sheet the iteration counts and the norm of the residual $\|A\mathbf{x}_{\mathbf{LSCQ}} - A\mathbf{b}\|$, where $\mathbf{x}_{\mathbf{LSCQ}}$ is the obtained approximate solution.

Answer:

```
// We use the Least Squares Conjugate Gradient (LSCG) solver to solve the previous problem
// Define the convergence tolerance and the maximum number of iterations
double tol = 1e-10;
int max_iter = 1000;
// Create the LSCG solver
Eigen::LeastSquaresConjugateGradient<Eigen::SparseMatrix<double>> lscg_solver;
// Set the maximum number of iterations and the convergence tolerance
lscg_solver.setMaxIterations(max_iter);
lscg_solver.setTolerance(tol);
// Compute the LSCG solver
lscg_solver.compute(A);
// Check if the computation was successful (not necessary at all)
if (lscg_solver.info() != Eigen::Success) {
    printf("Cannot compute the LSCG solver\n");
    return 0;
}
// Solve the system of equations using the LSCG solver
const Eigen::VectorXd x_lscg = lscg_solver.solve(A * b);
// Print the number of iterations, the relative residual, and the norm of the difference
// between the solution obtained by the LSCG solver and the expected solution
printf("\nSolution with Eigen LSCG:\n");
printf("Number of iterations: %ld\n", lscg_solver.iterations());
printf("Relative residual: %g\n", lscg_solver.error());
printf("Norm of the difference || Ax_SQR - Ab ||: %g\n", (x_lscg - b).norm());

```

And the result is:

```
Solution with Eigen LSCG:
Number of iterations: 87
Relative residual: 9.88679e-11
Norm of the difference || Ax_SQR - Ab ||: 2.14369e-07
```

5. Export the matrix A^TA in the matrix market format (save it as `AtA.mtx`) and move it to the lis-2.0.34/test folder. Using the Conjugate Gradient iterative solver available in LIS compute the approximate solution of the linear systm $A^TAx = b$ up to a tolerance of 10^{-10}. Report the iteration counts and the relative residual at the last iteration.

Answer:

```
// Save the Gram matrix to a file named AtA.mtx
saveMarket(gram_matrix, "AtA.mtx");
```

We download and unzip into the lis folder. Then we move the matrix to the test folder. After have compiled `test1.c`, we run the command:

```
./test1 AtA.mtx 2 sol.mtx hist.txt -i cg -tol 1.e-10
```

- `AtA.mtx` is the input matrix.
- `2` uses $b = A \times (1, 1, \ldots, 1)^T$.
- `sol.mtx` and `hist.txt` are the results.
- `-i cg` is the type of solver (Conjugate Gradient).
- `-tol 1.e-10` is the tolerance.

And the final result is:

```
number of processes = 1
matrix size = 400 x 400 (19590 nonzero entries)

initial vector x      : all components set to 0
precision             : double
linear solver         : CG
preconditioner        : none
convergence condition : ||b-Ax||_2 <= 1.0e-10 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status  : normal end

CG: number of iterations = 114
CG:   double             = 114
CG:   quad               = 0
CG: elapsed time         = 2.149641e-03 sec.
CG:   preconditioner     = 2.702900e-05 sec.
CG:     matrix creation  = 1.730000e-07 sec.
CG:   linear solver      = 2.122612e-03 sec.
CG: relative residual    = 8.993059e-11
```

8.1.2 June 17

Exercise 1 - Theory

1. **Consider the following problem: find $\mathbf{x} \in \mathbb{R}^n$, $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$ are given. State under which conditions the mathematical problem is well posed.**

   Answer: page 116.

2. **Describe the general form of a linear iterative method for the approximate solution of $A\mathbf{x} = \mathbf{b}$ and describe the stopping criteria.**

   Answer: Linear iterative methods are used to find approximate solutions to the linear system $A\mathbf{x} = \mathbf{b}$. These methods generate a sequence of approximations that ideally converge to the exact solution.

   Definition. A linear iterative method updates the approximation $\mathbf{x}^{(k)}$ at each step k according to a specific rule. The general form of an iterative method can be written as:

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

   Where:

   - $\mathbf{x}^{(k)}$ is the *approximation of the solution* at iteration k.
   - B is the *iteration matrix*.
   - $\mathbf{f}$ is a component that identifies the *selected method*.

   Stopping Criteria. The iterative process is stopped when one of the following criteria is met:

   - ***Maximum iterations***. The simpler way. It is used to avoid an infinite loop in case the chosen method doesn't converge.
   - ***Residual Norm***. Since the residual gets smaller as the solution gets closer to the exact answer, we stop the iteration method when it is small enough. This works because the residual essentially tracks the behavior of the error (usually small residual, small error).

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

     Where $K(A)$ is the condition number and $\mathbf{r}^{(k)}$ is the residual.
   - ***Distance between consecutive iterates criteria***. It is a criterion that looks at how much the current iterate (solution) changes from the previous one. When this difference becomes small enough, it's a signal that the method is converging and can be stopped.

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

3. State the necessary and sufficient condition for convergence.

Answer: The necessary and sufficient condition for convergence is given by the following theorem.

A consistent iterative method with iteration matrix B converges if and only if its spectral radius is less than 1: $\rho(B) < 1$.

The spectral radius of a matrix is the largest absolute value of its eigenvalues:

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

Where $\lambda_j(B)$ are the eigenvalues of B. Also, when the matrix B is Singular Positive-Definite (SPD), the spectral radius is equal to the Euclidean norm:

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

This result is very helpful because it indicates that the influence of the matrix is well distributed.

4. Describe the Jacobi iterative method. Recall the derivation of the scheme and the main theorical results.

Answer: Given a system of linear equations $A\mathbf{x} = \mathbf{b}$, where A is a square matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the result vector, the Jacobi method iterates to approximate the solution.

Algorithm

(a) Start with an initial guess $\mathbf{x}^{(0)}$, also zero.

(b) Update each component:

$$\mathbf{x}_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}} \qquad \forall i = 1, \ldots, n$$

where a_{ii} are the diagonal elements of A, and a_{ij} are the off-diagonal elements.

(c) Repeat until we meet the stop criteria.

Theoretical Results

- ***Convergence Condition***. The Jacobi method converges if and only if the spectral radius $\rho(B)$ of the iteration matrix is less than 1: $\rho(B) < 1$.
- ***Diagonal Dominance***. A sufficient condition for the convergence of the Jacobi method is that the matrix A is diagonally dominant. This means that for each row i:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

- ***Convergence Rate***. The convergence rate of the Jacobi method depends on the spectral radius of the iteration matrix. A smaller spectral radius results in faster convergence.

Exercise 1 - Laboratory

The exam text provides the following matrix:

The solution code snippet is available here:

1. **Download the matrix `Aex1.mtx` from the Exam folder and save it. Load the matrix in a new file called `exer1.cpp` using the `unsupported/Eigen/SparseExtra` module and check if the matrix is symmetric. Report on the sheet $\|A\|$ where $\|\cdot\|$ denotes the Euclidean norm.**

   Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Load the matrix
    Eigen::SparseMatrix<double> A;
    Eigen::loadMarket(A, "Aex1.mtx");
    // Matrix properties
    printf("Matrix size: %ldx%ld\n", A.rows(), A.cols());
    // Calculate the norm of the matrix
    const Eigen::SparseMatrix<double> A_t = A.transpose();
    const Eigen::SparseMatrix<double> B = A_t - A;
    printf("Euclidean norm of A: %g\n", A.norm());
    printf("Euclidean norm of A.t - A: %g\n", B.norm());
    // The matrix is symmetric if the norm of the skew-
    symmetric part
    // is approximately zero; we use a tolerance of 1.e-10;
    // This is because if the matrix is symmetric,
    // then A.t - A = 0 and the norm of the zero matrix is
    zero
    printf("The matrix is %s symmetric.\n", B.norm() > 1.e-10
    ? "not" : "");
}
```

   And the result is:

```
Matrix size: 500x500
Euclidean norm of A: 105.193
Euclidean norm of A.t - A: 47.9613
The matrix is not symmetric.
```

2. Define the `Eigen` vector $\mathbf{b} = (1, 1, \ldots, 1)^T$ and find the approximate solution $\mathbf{x}_j$ of $A\mathbf{x} = \mathbf{b}$ using the Jacobi method (implemented in the `jacobi.hpp` template). Fix a maximum number of iterations which is sufficient to reduce the (relative) residual below than 10^{-5} and take $\mathbf{x}_0 = \mathbf{b}$ as initial guess. Report on the sheet the iteration counts and $\|\mathbf{x}_j\|$.

Answer:

```cpp
#include "headers/jacobi.hpp"

int main() {
    // ...
    // Define the vector b
    const Eigen::VectorXd b = Eigen::VectorXd::Ones(A.rows());
    // Calculate the approximate solution using the Jacobi
    method
    // Define the tolerance and maximum number of iterations
    double tol = 1.e-5;
    int max_iter = 5000;
    // Create the Diagonal Preconditioner needed for the
    Jacobi method
    Eigen::DiagonalPreconditioner<double> D(A);
    // Create the vector x to store the approximate solution
    Eigen::VectorXd x = Eigen::VectorXd::Zero(A.rows());
    // Solve the system using the Jacobi method
    LinearAlgebra::Jacobi(A, x, b, D, max_iter, tol);
    // Print the results
    printf("- Jacobi method -\n");
    printf("Iterations count: %d\n", max_iter);
    printf("Norm of the solution: %g\n", x.norm());
}
```

And the result is:

```
- Jacobi method -
Iterations count: 13
Norm of the solution: 7.07625
```

3. Compute the approximate solution $\mathbf{x}_g$ of $A\mathbf{x} = \mathbf{b}$ obtained using the BICGSTAB method available in `Eigen`. Fix a maximum number of iterations which is sufficient to reduce the residual below than 10^{-10} considering $\mathbf{x}_j$ (computed in the previous point) as initial guess. Use the `Eigen` diagonal preconditioner. Report the iteration counts and $\|\mathbf{x}_j - \mathbf{x}_g\|$.

Answer:

```
// Calculate the approximate solution using the BiCGSTAB
    method
// Define the tolerance and maximum number of iterations
tol = 1.e-10;
max_iter = 500;
// Create the vector x to store the approximate solution
Eigen::VectorXd x_bicgstab = Eigen::VectorXd::Zero(A.rows());
// Solve the system using the BiCGSTAB method and set the
    preconditioner to the Diagonal Preconditioner
Eigen::BiCGSTAB<Eigen::SparseMatrix<double>, Eigen::
    DiagonalPreconditioner<double>> bi_cgstab(A);
bi_cgstab.setMaxIterations(max_iter);
bi_cgstab.setTolerance(tol);
x_bicgstab = bi_cgstab.solveWithGuess(b, x);
// Print the results
printf("- BiCGSTAB method -\n");
printf("Iterations count: %ld\n", bi_cgstab.iterations());
printf("Estimated error: %g\n", bi_cgstab.error());
```

And the result is:

```
- BiCGSTAB method -
Iterations count: 4
Estimated error: 3.62252e-12
```

4. Repeat the previous point using the iterative solvers available in the LIS library. First, compute the approximate solution obtained with the Jacobi method prescribing a tolerance of 10^{-5}. Then, using the BICGSTAB solver compute the approximate solution up to a tolerance of 10^{-10}. Find a preconditioning strategy that yield a decrease in the number of required iterations with respect to the BICGSTAB method without preconditioning. Report on the sheet the iteration counts and the relative residual at the last iteration.

Answer: We download and unzip into the lis folder. Then we move the matrix to the test folder. After have compiled `test1.c`, we run the command:

```
$ ./test1 Aex1.mtx 1 sol.mtx hist.txt -i jacobi -tol 1.e-5
number of processes = 1
matrix size = 500 x 500 (7708 nonzero entries)

initial vector x      : all components set to 0
precision             : double
linear solver         : Jacobi
preconditioner        : none
convergence condition : ||b-Ax||_2 <= 1.0e-05 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status  : normal end

Jacobi: number of iterations = 14
Jacobi:   double             = 14
Jacobi:   quad               = 0
Jacobi: elapsed time         = 1.407420e-04 sec.
Jacobi:   preconditioner     = 4.766000e-06 sec.
Jacobi:     matrix creation  = 2.720000e-07 sec.
Jacobi:   linear solver      = 1.359760e-04 sec.
Jacobi: relative residual    = 9.603771e-06
```

This is the Jacobi method with a tolerance of 10^{-5}. Then we use the BICGSTAB solver:

```
$ ./test1 Aex1.mtx 1 sol.mtx hist.txt -i bicgstab -tol 1.e-10
number of processes = 1
matrix size = 500 x 500 (7708 nonzero entries)

initial vector x      : all components set to 0
precision             : double
linear solver         : BiCGSTAB
preconditioner        : none
convergence condition : ||b-Ax||_2 <= 1.0e-10 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status  : normal end

BiCGSTAB: number of iterations = 10
BiCGSTAB:   double             = 10
BiCGSTAB:   quad               = 0
BiCGSTAB: elapsed time         = 2.180690e-04 sec.
BiCGSTAB:   preconditioner     = 6.565000e-06 sec.
BiCGSTAB:     matrix creation  = 3.380000e-07 sec.
BiCGSTAB:   linear solver      = 2.115040e-04 sec.
BiCGSTAB: relative residual    = 5.109306e-11
```

The ILU preconditioner is one of the best, so we can try it right away:

```
$ ./test1 Aex1.mtx 1 sol.mtx hist.txt -i bicgstab -tol 1.e-10
    -p ilu
number of processes = 1
matrix size = 500 x 500 (7708 nonzero entries)

initial vector x      : all components set to 0
precision             : double
linear solver         : BiCGSTAB
preconditioner        : ILU(0)
convergence condition : ||b-Ax||_2 <= 1.0e-10 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status  : normal end

BiCGSTAB: number of iterations = 4
BiCGSTAB:   double             = 4
BiCGSTAB:   quad               = 0
BiCGSTAB: elapsed time         = 2.838500e-04 sec.
BiCGSTAB:   preconditioner     = 1.208690e-04 sec.
BiCGSTAB:     matrix creation  = 9.980000e-07 sec.
BiCGSTAB:   linear solver      = 1.629810e-04 sec.
BiCGSTAB: relative residual    = 8.914148e-11
```

It is also interesting notice that if we use the fill-in technique with $k = 4$ (`ilu_fill` parameter) we obtain a single iteration. Fill-in refers to the number of non-zero elements that are added to the lower and upper triangular matrices during the ILU factorization process.

```
$ ./test1 Aex1.mtx 1 sol.mtx hist.txt -i bicgstab -tol 1.e-10
    -p ilu -ilu_fill 4
number of processes = 1
matrix size = 500 x 500 (7708 nonzero entries)

initial vector x      : all components set to 0
precision             : double
linear solver         : BiCGSTAB
preconditioner        : ILU(4)
convergence condition : ||b-Ax||_2 <= 1.0e-10 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status  : normal end

BiCGSTAB: number of iterations = 1
BiCGSTAB:   double             = 1
BiCGSTAB:   quad               = 0
BiCGSTAB: elapsed time         = 9.994100e-05 sec.
BiCGSTAB:   preconditioner     = 5.042200e-05 sec.
BiCGSTAB:     matrix creation  = 1.720000e-07 sec.
BiCGSTAB:   linear solver      = 4.951900e-05 sec.
BiCGSTAB: relative residual    = 1.978281e-11
```

Exercise 2 - Theory

1. **Consider the following eigenvalue problem: $Ax = \lambda x$, where $A \in \mathbb{R}^{n \times n}$ is given. Describe the power method for the numerical approximation of the largest in modulus eigenvalue of A. Introduce the notation, the algorithm, ad the applicability conditions.**

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

Notation

- A: An $n \times n$ real matrix.
- $\mathbf{x}$: An eigenvector corresponding to the eigenvalue λ.
- λ: The eigenvalue associated with the eigenvector $\mathbf{x}$.
- $\mathbf{x}^{(k)}$: The approximation of the eigenvector at iteration k.
- $\lambda^{(k)}$: The approximation of the eigenvalue at iteration k.

Algorithm

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

(b) For $k \geq 0$:

i. Multiply the current vector by the matrix:

$$\mathbf{y}^{(k+1)} = A\mathbf{x}^{(k)}$$

ii. After each multiplication, normalize the vector to prevent it from becoming too large:

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

iii. Approximate the eigenvalue (an estimate of the eigenvalue associated with the current eigenvector approximation):

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

(c) Repeat until we meet a specific stopping criteria.

Applicability conditions

- ***Distinct Dominant Eigenvalue***: Convergence is guaranteed if the dominant eigenvalue λ_1 is unique and the largest in absolute value compared to other eigenvalues. The dominant eigenvalue must be well-separated from the other eigenvalues for rapid convergence.
- ***Starting Vector***: The initial vector $\mathbf{x}^{(0)}$ must have a non-zero component in the direction of the dominant eigenvector $\mathbf{x}_1$. Typically, a random initial vector suffices because it will almost always have a component in the direction of $\mathbf{x}_1$.

2. State the main theoretical results.

Answer:

- Dominant Eigenvalue. The power method converges to the eigenvalue λ_1 with the largest absolute value (dominant eigenvalue) of the matrix A. The associated eigenvector $\mathbf{x}_1$ will also be obtained.
- Convergence Rate. The convergence rate of the power method depends on the ratio of the largest absolute eigenvalue $|\lambda_1|$ to the second largest absolute eigenvalue $|\lambda_2|$:
  - $\frac{|\lambda_2|}{|\lambda_1|} \ll 1$, convergence rate high, the method converges quickly.
  - $\frac{|\lambda_2|}{|\lambda_1|} \approx 1$, convergence rate low, the method converges slowly.

3. Comment of the computational costs.

Answer: The computational cost of the power method is determined by several factors, including the size of the matrix A, the number of iterations required to achieve convergence, and the operations performed during each iteration.

At each iteration we need to perform a matrix-vector multiplication $A\mathbf{x}^{(k)}$ and this involves in:

- Dense matrix: $O\left(n^2\right)$.
- Sparse matrix: $O\left(m\right)$, where m is the number of nonzero entries.

Also, normalization is required at each iteration. Therefore, the normalization and the division take $O\left(n\right)$ operations (both dense and sparse matrices). Finally, the number of iterations of the algorithm is K, and its value depends on how much accuracy we want and on the spectral gap (absolute value of the second largest eigenvalue divided by the absolute value of the largest eigenvalues).

We can conclude that the complexity is:

- Dense matrix: $O\left(K \cdot \left(n^2 + n\right)\right)$
- Sparse matrix: $O\left(K \cdot \left(m + n\right)\right)$

Exercise 2 - Laboratory

The solution code snippet is available here:

1. **Let A be a 100×100 pentadiagonal matrix defined such that**

$$A = \begin{pmatrix} -8 & 3 & 1 & 0 & 0 & \cdots & 0 \\ 3 & -8 & 3 & 1 & 0 & \cdots & 0 \\ 1 & 3 & -8 & 3 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \cdots & 0 \\ \vdots & \ddots & 1 & 3 & -8 & 3 & 1 \\ \vdots & \vdots & 0 & 1 & 3 & -8 & 3 \\ 0 & 0 & \cdots & 0 & 1 & 3 & -8 \end{pmatrix}$$

In a new file called `exer2.cpp`, define the matrix A in the sparse format. Report on the sheet $\mathbf{v}^T A \mathbf{v}$, where $\mathbf{v}$ is such that $v_i = -1$ for all $0 \leq i < 100$.

Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Create a 100x100 sparse matrix
    Eigen::SparseMatrix<double> A(100, 100);
    const int n = static_cast<int>(A.rows());
    // Create a pentadiaonal matrix
    for (int row = 0; row < n; ++row) {
        A.insert(row, row) = -8;
        if(row > 0) A.coeffRef(row, row-1) = 3;
        if(row < n-1) A.coeffRef(row, row+1) = 3;
        if(row > 1) A.coeffRef(row, row-2) = 1;
        if(row < n-2) A.coeffRef(row, row+2) = 1;
    }
    // Create vector v composed of -1 from 1 to 100
    Eigen::VectorXd v = -Eigen::VectorXd::Ones(n);
    // Report the norm of v.t A v
    printf("Norm of v.t A v: %g\n", (v.transpose() * A * v).
    value());
}
```

And the result is:

```
Norm of v.t A v: -10
```

2. Solve the eigenvalue problem $Ax = \lambda x$ using the proper solver provided by `Eigen`. Report on the sheet the smallest and largest $\lambda_{\min} < \lambda_{\max}$ computed eigenvalues of A.

Answer:

```cpp
#include <Eigen/Dense>

int main() {
    // ...
    // Create a copy of A
    const Eigen::SparseMatrix<double> A_copy = Eigen::
    SparseMatrix(A);
    // Use selfadjoint eigen solver to compute the eigenvalues
     of A
    const Eigen::SelfAdjointEigenSolver<Eigen::SparseMatrix<
    double>> self_adjoint_eigen_solver(A_copy);
    // Check if the computation was successful
    if (self_adjoint_eigen_solver.info() != Eigen::Success) {
        printf("Error: Computation failed.");
        return 1;
    }
    // Print the eigenvalues of A
    const double lambda_min = self_adjoint_eigen_solver.
    eigenvalues()[0];
    const double lambda_max = self_adjoint_eigen_solver.
    eigenvalues()[n-1];
    printf("The minimum eigenvalue of A is: %g\n", lambda_min)
    ;
    printf("The maximum eigenvalue of A is: %g\n", lambda_max)
    ;
}
```

And the result is:

```
The minimum eigenvalue of A is: -12.2484
The maximum eigenvalue of A is: -0.00672505
```

3. Using the `unsupported/Eigen/SparseExtra` module, export matrix A in the matrix market format (save as `Aex2.mtx`) and move it to the folder `lis-2.0.34/test`. Using the proper iterative solver available in the LIS library compute the largest eigenvalue $\lambda_{\max}$ of A up to a tolerance of 10^{-7}. Report the computed eigenvalue and the number of iterations required to achieve the prescribed tolerance.

Answer: We export the matrix:

```
// Export matrix
saveMarket(A, "./Aex2.mtx");
```

We download and unzip into the lis folder. Then we move the matrix to the test folder. After have compiled `eigen1.c`, we run the command:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e ii -emaxiter 25000
    -etol 1.e-7

number of processes = 1
matrix size = 100 x 100 (494 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Inverse
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : 0.000000e+00
linear solver         : BiCG
preconditioner        : none
eigensolver status    : normal end

Inverse: mode number          = 0
Inverse: eigenvalue           = -6.725050e-03
Inverse: number of iterations = 8
Inverse: elapsed time         = 4.748510e-04 sec.
Inverse:   preconditioner     = 3.319400e-05 sec.
Inverse:     matrix creation  = 2.200000e-07 sec.
Inverse:   linear solver      = 3.652310e-04 sec.
Inverse: relative residual    = 6.262051e-08
```

Since the eigenvalues are negative, we must use the inverse power method to find the largest eigenvalue (`-0.00672505`). Normally we use the inverse power method to find the minimum, but here the eigenvalues are negative!

4. Using the proper iterative solver available in the LIS library compute the smallest eigenvalue $\lambda_{\min}$ of A up to a tolerance of 10^{-7}. Report the computed eigenvalue and the number of iterations required to achieve the prescribed tolerance.

Answer: Since the eigenvalues are negative, we must use the power method to find the smallest eigenvalue:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e pi -emaxiter 25000
    -etol 1.e-7

number of processes = 1
matrix size = 100 x 100 (494 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Power
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : 0.000000e+00
eigensolver status    : normal end

Power: mode number          = 0
Power: eigenvalue           = -1.224829e+01
Power: number of iterations = 23732
Power: elapsed time         = 1.925408e-02 sec.
Power:   preconditioner     = 0.000000e+00 sec.
Power:     matrix creation  = 0.000000e+00 sec.
Power:   linear solver      = 0.000000e+00 sec.
Power: relative residual    = 9.996329e-08
```

5. Find a shift $\mu \in \mathbb{R}$ yielding an acceleration of the previous eigensolver. Report μ and the number of iterations required to achieve a tolerance of 10^{-7}.

Answer:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e pi -emaxiter 50000
    -etol 1.e-7 -shift -6.0

number of processes = 1
matrix size = 100 x 100 (494 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Power
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : -6.000000e+00
eigensolver status    : normal end

Power: mode number          = 0
Power: eigenvalue           = -1.224829e+01
Power: number of iterations = 12997
Power: elapsed time         = 1.075117e-02 sec.
Power:   preconditioner     = 0.000000e+00 sec.
Power:     matrix creation  = 0.000000e+00 sec.
Power:   linear solver      = 0.000000e+00 sec.
Power: relative residual    = 9.997123e-08
```

8.1.3 January 23

Exercise 1 - Theory

1. **Consider the following problem: find $\mathbf{x} \in \mathbb{R}^n$, such that $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$ are given. State under which conditions the mathematical problem is well posed.**

   Answer: page 116.

2. **Describe the LU factorization and its use to approximately solve the above linear system.**

   Answer: LU factorization is a method of decomposing a given square matrix A into the product of two matrices: a lower triangular matrix L with ones on the diagonal ($L_{ii} = 1$) and an upper triangular matrix U.

   $$A = LU$$

   Steps to Solve $A\mathbf{x} = \mathbf{b}$ using LU Factorization:

   (a) Compute the LU factorization of A:

   $$A = LU$$

   (b) Solve the intermediate system $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$: since L is a lower triangular matrix, this step is performed using ***forward substitution***. Start with the first equation and solve for y_1, then proceed to the next equation, incorporating previously computed values.

   $$\begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

   (c) Solve the upper triangular system $U\mathbf{y} = \mathbf{x}$: using the solution $\mathbf{y}$ obtained from the forward substitution step, solve for $\mathbf{x}$ using ***backward substitution***. Start with the last equation and solve for x_n, then proceed upwards, incorporating previously computed values.

   $$\begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

3. State the necessary and sufficient condition that guarantees existence and uniqueness of the LU factorization. For what classes of matrices the LU factorization exists and is unique?

Answer: The ***necessary and sufficient condition*** for the existence and uniqueness of the LU factorization of a matrix A is that all leading principal minors[9] of A are non-zero.

Formally, let A be an $n \times n$ matrix. The LU factorization $A = LU$ exists and is unique if and only if the leading principal minors of A, i.e., the determinants of the submatrices $A_{1:k,1:k}$ for $k = 1, 2, \ddots, n$, are non-zero:

$$\exists\,(A = LU) \iff \det(A_{1:k,1:k}) \neq 0 \qquad \forall k = 1, 2, \ldots, n$$

Classes of Matrices

(a) ***Non-Singular (Invertible) Matrices***. If A is a non-singular (invertible) matrix and all leading principal minors are non-zero, LU factorization exists and is unique.

(b) ***Diagonally Dominant Matrices***. For any matrix A that is strictly diagonally dominant (where the magnitude of each diagonal element is greater than the sum of the magnitudes of the other elements in the same row), the LU factorization exists and is unique without the need for pivoting.

(c) ***Positive Definite Matrices***. If A is a symmetric positive definite matrix, LU factorization exists and is unique. This case often leads to Cholesky decomposition, where $L = U^T$.

Pivoting. For matrices that do not meet these conditions (i.e., if some leading principal minors are zero or close to zero), LU factorization can still be performed by using partial pivoting or complete pivoting.

[9] A **Leading Principal Minor** is the determinant of a top-left submatrix of A.

4. Describe the main pivoting techniques and comment on the computational costs.

Answer: The ***Main pivoting techniques***

(a) Pivoting by rows (Partial Pivoting):

- It involves swapping the current row with another row below it such that the largest absolute value element in the current column is moved to the diagonal position.
- The procedure is: for each column k (from the first to the last)
  i. Find the row i (where $i > k$) with the largest absolute value in the pivot column.
  ii. Swap row k with row i to move the largest pivot element to the diagonal position a_{kk}.
  iii. Continue with the Gaussian elimination process.
- Its goal is to avoid division by zero (pivot element $\neq 0$) and to improve numerical stability by avoiding division by small numbers.

(b) Complete Pivoting:

- It involves both row and column swaps to ensure the largest absolute value element in the remaining submatrix is moved to the diagonal position.
- The procedure is: for each step k
  i. Identify the largest absolute value element in the remaining submatrix.
  ii. Swap the current row k with the row containing the largest element.
  iii. Swap the current column k with the column containing the largest element.
- Its goal is to guarantee the most accurate and stable results during matrix factorization. Because of the double swap (row and column), it introduces more overhead despite the partial pivoting.

Computational Costs The computational cost of pivoting techniques includes the overhead associated with identifying and performing row and column swaps.

- Pivoting by rows (partial pivoting). The maximum element search takes $O\left(n^2\right)$, the row swapping is $O\left(n^2\right)$, therefore the ***total cost*** is $O\left(n^2\right)$, additional to the $O\left(n^3\right)$ cost of LU factorization.
- Complete Pivoting. The maximum element search takes $O\left(n^3\right)$, the row/column swapping is $O\left(n^2\right)$, therefore the ***total cost*** is $O\left(n^3\right)$, comparable to the cost of LU factorization.

Complete pivoting provides greater numerical stability at the cost of increased computational effort, whereas partial pivoting offers a good balance between stability and efficiency.

Exercise 1 - Laboratory

The exam text provides the following matrix:

A: B: C:

The solution code snippet is available here:

1. **Download the sparse matrices `A.mtx`, `B.mtx`, and `C.mtx` and save it. Load the three matrices in a new file `exer1.cpp`. Define the block matrix $M = \begin{pmatrix} A & B^T \\ B & C \end{pmatrix}$. Report on the `.txt` file the matrix size and the Euclidean norm of M. Is the matrix symmetric?**

   Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Load the 3 matrices
    Eigen::SparseMatrix<double> A, B, C;
    loadMarket(A, "A.mtx");
    loadMarket(B, "B.mtx");
    loadMarket(C, "C.mtx");
    const auto B_t = Eigen::SparseMatrix<double>(B.transpose());
    // Create the matrix M
    Eigen::MatrixXd M(A.rows()+B.rows(), A.cols()+B_t.cols());
    // Create the matrix M = [A B.t ; B C]
    M.topLeftCorner(A.rows(), A.cols()) = A;
    M.topRightCorner(A.rows(), B_t.cols()) = B_t;
    M.bottomLeftCorner(B.rows(), A.cols()) = B;
    M.bottomRightCorner(B.rows(), C.cols()) = C;
    // Print size of M
    printf("Size of M: %ldx%ld\n", M.rows(), M.cols());
    // The norm (magnitude) of the matrix M and the M-M.t norm
    printf("Norm of M: %g\n", M.norm());
    printf("Norm of M-M.t is %g ", (M.transpose() - M).norm());
    // To see if the matrix is symmetric
    // we can check if the difference between M and M.t is zero.
    // Instead of checking if the difference is zero,
    // we can check if the norm of the difference is zero.
    printf("and the matrix M is indeed %s symmetric.\n",
            (M.transpose() - M).norm() < 1e-10 ? "" : "not");
}
```

   And the result is:

```
Size of M: 278x278
Norm of M: 459.79
Norm of M-M.t is 132.073 and the matrix M is indeed not
    symmetric.
```

2. Define an `Eigen` vector b $= (1, 1, \ldots, 1)^T$ with size equal to the number of rows of M. Solve the linear system Mx $=$ b using the LU decomposition method available in the `Eigen` library. Report on the `.txt` file the norm of the obtained absolute residual.

Answer:

```
// Define vector b
const Eigen::VectorXd b = Eigen::VectorXd::Ones(M.rows());
// Solve the system Mx = b using the LU decomposition
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
solver.compute(M.sparseView());
if (solver.info() != Eigen::Success) {
    printf("Factorization failed\n");
    return 1;
}
const Eigen::VectorXd x = solver.solve(b);
// Print the norm of the absolute residual
// We compute the absolute residual by subtracting the product
     of M and x from b.
// In fact, the residual is the difference between the right-
    hand side
// and the left-hand side of the equation.
// If the residual is (approximately) zero, the solution is
    correct.
printf("Norm of the absolute residual: %g\n", (M*x - b).norm()
    );
```

And the result is:

```
Norm of the absolute residual: 1.62043e-13
```

3. Using again the LU decomposition provided by `Eigen`, compute an approximation of the Schur complement of M with respect to the A block defined as the matrix $S = C - BA^{-1}B^T$. Report on the `.txt` file the matrix size and the Euclidean norm of S.

Answer:

```
/**
* The Schur complement is a concept in linear algebra that
* describes the result of eliminating a set of variables from
    a system of linear equations.
*
* However, from the definition of the Schur complement,
* we can see that it is not necessary to compute the inverse
    of A.
* We can use the LU decomposition of A to calculate its
    inverse.
* Therefore, we can directly compute the Schur complement
    using its definition.
*
* This is the same thing we do in the real world on the paper.
*/
// Compute the inverse of A
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_A;
solver_A.compute(A);
if (solver_A.info() != Eigen::Success) {
    printf("Factorization failed\n");
    return 1;
}
const Eigen::SparseMatrix<double> A_inv = solver_A.solve(
    Eigen::MatrixXd::Identity(A.rows(), A.cols()).sparseView()
);
// Compute the Schur complement
const Eigen::SparseMatrix<double> S = C - B * A_inv * B_t;
// Print the norm of the Schur complement
printf("Norm of the Schur complement: %g\n", S.norm());
```

And the result is:

```
Norm of the Schur complement: 9.7299
```

4. Solve the linear system:

$$M\mathbf{x} = \begin{pmatrix} A & B^T \\ B & C \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} = \mathbf{b}$$

by exploiting the Schur complement S. First use the LU factorization method to approximate the solution of $S\mathbf{x}_2 = \mathbf{b}_2 - BA^{-1}\mathbf{b}_1$. Then compute the approximate solution of $A\mathbf{x}_1 = \mathbf{b}_1 - B^T\mathbf{x}_2$. Report the norm of the absolute residual $\mathbf{r} = \mathbf{b} - M * [\overline{\mathbf{x}}_1, \overline{\mathbf{x}}_2]^T$, where $\overline{\mathbf{x}}_1$ and $\overline{\mathbf{x}}_2$ are the computed approximations of $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively.

Answer:

```
// Solve in two steps exploiting the Schur complement
Eigen::SparseLU<Eigen::SparseMatrix<double>> solve_lu;
// Compute the Schur complement
solve_lu.compute(S);
if (solve_lu.info() != Eigen::Success) {
    printf("Factorization failed\n");
    return 1;
}
const Eigen::VectorXd x2 = solve_lu.solve(b.tail(S.rows()) - B
    *solver_A.solve(b.head(A.rows())));
const Eigen::VectorXd x1 = solver_A.solve(b.head(A.rows()) -
    B_t*x2);
Eigen::VectorXd x12(M.rows());
x12 << x1, x2;
// Print the norm of the absolute residual
printf("Solution with Schur complement\n");
printf("Norm of the absolute residual: %g\n", (b - M*x12).norm
    ());
printf("Comparison solutions: %g\n", (x - x12).norm());
```

And the result is:

```
Solution with Schur complement
Norm of the absolute residual: 1.29778e-13
Comparison solutions: 1.13896e-13
```

Exercise 2 - Theory

1. **Consider the following eigenvalue problem: $Ax = \lambda x$, where $A \in \mathbb{R}^{n \times n}$ is given. Describe the inverse power method for the numerical approximation of the smallest in modulus eigenvalue of A. Introduce the notation and discuss the applicability conditions.**

   Answer: The inverse power method is an iterative technique used to find the smallest eigenvalues of a matrix.

   The inverse power method leverages the fact that applying the power method to the inverse of a matrix A will converge to the eigenvalue of A^{-1} with the largest magnitude. Since the eigenvalues of A^{-1} are the reciprocals of the eigenvalues of A, the inverse power method converges to the eigenvalue of A with the smallest absolute value.

   Notation

   - A: An $n \times n$ real matrix.
   - $\mathbf{x}$: An eigenvector corresponding to the eigenvalue λ.
   - λ: The eigenvalue associated with the eigenvector $\mathbf{x}$.
   - $\mathbf{x}^{(k)}$: The approximation of the eigenvector at iteration k.
   - $\lambda^{(k)}$: The approximation of the eigenvalue at iteration k.

   Applicability Conditions

   (a) ***Invertibility***: The matrix A must be invertible, i.e., it should have a non-zero determinant.

   (b) ***Distinct Smallest Eigenvalue***: The smallest eigenvalue in magnitude should be distinct and well-separated from the other eigenvalues. This ensures rapid convergence.

   (c) ***Initial Vector***: The initial vector $\mathbf{x}^{(0)}$ should have a non-zero component in the direction of the eigenvector corresponding to the smallest eigenvalue. Random vectors typically satisfy this condition. Also, its Euclidean norm should be one.

   (d) ***Numerical Stability***: Solving the linear system $A\mathbf{y}^{(k+1)} = \mathbf{x}^{(k)}$ at each iteration can introduce numerical instability, especially if A is ill-conditioned. Preconditioning or regularization techniques might be needed.

   Tip

   How can we remember the "Applicability Conditions"?

   (a) ***Invertibility***. If the matrix A is not invertible, then the matrix A^{-1} doesn't exist. This means that the step in the algorithm $A\mathbf{y}^{(k+1)} = \mathbf{x}^{(k)}$ cannot be solved if the matrix is not invertible!

   (b) ***Distinct Smallest Eigenvalue***. Having a distinct smallest eigenvalue ensures that the inverse power method can

effectively and accurately converge to the correct smallest eigenvalue. It guarantees that the iterative process is dominated by the single smallest eigenvalue, leading to reliable and stable convergence.

(c) ***Initial Vector***. In each iteration of the inverse power method, the vector is multiplied by A^{-1}. This process amplifies the components of the vector in the direction of the eigenvector corresponding to the smallest eigenvalue. If the initial vector $\mathbf{x}^0$ has a zero component in this direction, the iterative process will not capture or amplify this component, leading to incorrect convergence.

(d) ***Numerical Stability***. The first operation of the algorithm at each step can introduce some instability, so preconditioning and pivoting techniques are used to massage the matrices during factorization.

2. Write the (pseudo) algorithm.

Answer:

(a) Choose an initial non-zero vector $\mathbf{x}^{(0)}$, such that its norm is one $\left\|\mathbf{x}^{(0)}\right\| = 1$. Often, a random vector or a vector of all ones is chosen.

(b) For $k \geq 0$ until convergence:

i. Solve the linear system:

$$A\mathbf{y}^{(k+1)} = \mathbf{x}^{(k)}$$

ii. Normalize the resulting vector to prevent it from growing too large:

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

iii. Approximate the eigenvalue:

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

(c) Repeat until we meet a specific stopping criteria, for example:

$$\left|\lambda^{(k+1)} - \lambda^{(k)}\right| < \varepsilon$$

3. State the main theoretical result.

Answer:

- ***Convergence***. The inverse power method converges to the eigenvalue of A with the smallest absolute value. This is because applying the power method to A^{-1} targets the largest eigenvalue of A^{-1}, which corresponds to the smallest eigenvalue of A.
- ***Rate of Convergence***. The rate of convergence of the inverse power method depends on the ratio of the second smallest eigenvalue to the smallest eigenvalue.

  If λ_1 is the smallest eigenvalue and λ_2 is the second smallest eigenvalue in magnitude, the error in the k-th iteration decreases proportionally to:

  $$\left(\frac{\lambda_1}{\lambda_2}\right)^k$$

  A larger gap between the smallest and the second smallest eigenvalues results in faster convergence.

  Faster convergence when:

  $$\frac{|\lambda_1|}{|\lambda_2|} \ll 1$$

- ***Computational Cost***. For the dense matrices it takes n^3 flops, while for the sparse matrices it takes only $n \cdot m$ flops (where n is the number of rows in the square matrix, and m is the number of non-zero elements).

Exercise 2 - Laboratory

The solution code snippet is available here:

1. **Let A be a 100×100 tetradiagonal matrix defined such that:**

$$\begin{pmatrix} 8 & -4 & -1 & 0 & 0 & \cdots & 0 \\ -2 & 8 & -4 & -1 & 0 & \cdots & 0 \\ 0 & -2 & 8 & -4 & -1 & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & -1 \\ \vdots & \vdots & \vdots & 0 & -2 & 8 & -4 \\ 0 & 0 & \cdots & 0 & 0 & -2 & 8 \end{pmatrix}$$

In a new file called `exer2.cpp`, define the matrix A in the sparse format. Report $\|A\|$ on the `.txt` file, where $\|\cdot\|$ denotes the Euclidean norm.

Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>

int main() {
    // Create matrix A 100x100
    constexpr int rows = 100;
    constexpr int cols = 100;
    Eigen::SparseMatrix<double> A(rows, cols);
    // Insert on the matrix A:
    // * 8 on the main diagonal
    // * -2 on the first subdiagonal
    // * -4 on the first superdiagonal
    // * -1 on the second superdiagonal
    //  8 -4 -1
    // -2  8 -4 -1
    //    -2  8 -4 -1
    //       -2  8 -4 -1
    for (int row = 0; row < rows; ++row) {
        A.insert(row, row) = 8;
        if (row > 0) A.insert(row, row-1) = -2;
        if (row < rows-1) A.insert(row, row+1) = -4;
        if (row < rows-2) A.insert(row, row+2) = -1;
    }
    // Make matrix A compressed
    A.makeCompressed();
    printf("Euclidean norm of A: %g\n", A.norm());
}
```

And the result is:

```
Euclidean norm of A: 92.0761
```

2. Solve the eigenvalue problem $Ax = \lambda x$ using the proper solver provided by `Eigen`. Report on the `.txt` file the computed smallest and largest (in modulus) eigenvalues of A.

Answer:

```cpp
#include <unsupported/Eigen/SparseExtra>
#include <Eigen/Dense>

int main() {
    // ...
    // Check if the matrix is symmetric to use the correct eigenvalue solver
    const Eigen::SparseMatrix<double> A_transpose = A.transpose();
    const Eigen::SparseMatrix<double> B = A_transpose - A;
    const bool is_symmetric = B.norm() < 1e-10;
    printf("Norm of A.t - A: %g\n", B.norm());
    printf("The matrix A is %s\n", is_symmetric ? "symmetric" : "not symmetric");
    // Copy the original matrix
    const auto A_sparse_copy = Eigen::SparseMatrix(A);
    // Found the largest/smallest eigenvalue of A
    if (is_symmetric) {
        const Eigen::SelfAdjointEigenSolver<Eigen::SparseMatrix<double>> eigen_solver(A_sparse_copy);
        if (eigen_solver.info() != Eigen::Success) {
            printf("Eigen solver failed\n");
            return 1;
        }
        printf("Largest eigenvalue of A: %g\n", eigen_solver.eigenvalues()[rows-1]);
        printf("Smallest eigenvalue of A: %g\n", eigen_solver.eigenvalues()[0]);
    } else {
        const Eigen::EigenSolver<Eigen::MatrixXd> eigen_solver(A_sparse_copy.toDense());
        if (eigen_solver.info() != Eigen::Success) {
            printf("Eigen solver failed\n");
            return 1;
        }
        // Dense matrix, 1x100, of complex numbers
        const Eigen::Matrix<std::complex<double>, -1, 1>& eigenvalues = eigen_solver.eigenvalues();
        printf("Largest eigenvalue of A: %g\n", eigenvalues[rows-1].real());
        printf("Smallest eigenvalue of A: %g\n", eigenvalues[0].real());
    }
}
```

And the result is:

```
Norm of A.t - A: 31.4325
The matrix A is not symmetric
Largest eigenvalue of A: 11.1048
Smallest eigenvalue of A: 1.91332
```

3. Using the `unsupported/Eigen/SparseExtra` module, export matrix A in the matrix market format (save as `Aex2.mtx`) and move it to the folder `lis-2.0.34/test`. Using the proper iterative solver available in the LIS library compute the largest eigenvalue $\lambda_{\max}$ of A up to a tolerance of 10^{-7}. Report on the `.txt` file the computed eigenvalue and the number of iterations required to achieve the prescribed tolerance.

Answer: We export the matrix:

```
// Save the matrix A to a file
saveMarket(A, "Aex2.mtx");
printf("Matrix A saved to Aex2.mtx\n");
```

We download and unzip into the lis folder. Then we move the matrix to the test folder. After have compiled `eigen1.c`, we run the command:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e pi -etol 1.e-7 -
    emaxiter 50000

number of processes = 1
matrix size = 100 x 100 (396 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Power
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : 0.000000e+00
eigensolver status    : normal end

Power: mode number          = 0
Power: eigenvalue           = 1.299901e+01
Power: number of iterations = 39800
Power: elapsed time         = 3.054283e-02 sec.
Power:   preconditioner     = 0.000000e+00 sec.
Power:     matrix creation  = 0.000000e+00 sec.
Power:   linear solver      = 0.000000e+00 sec.
Power: relative residual    = 9.997934e-08
```

4. **Find a shift $\mu \in \mathbb{R}$ yielding an acceleration of the previous eigensolver. Report μ and the number of iterations required to achieve a tolerance of 10^{-7}.**

   Answer:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e pi -etol 1.e-7 -
    emaxiter 50000 -shift 7.0

number of processes = 1
matrix size = 100 x 100 (396 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Power
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : 7.000000e+00
eigensolver status    : normal end

Power: mode number          = 0
Power: eigenvalue           = 1.299901e+01
Power: number of iterations = 19920
Power: elapsed time         = 1.569868e-02 sec.
Power:   preconditioner     = 0.000000e+00 sec.
Power:     matrix creation  = 0.000000e+00 sec.
Power:   linear solver      = 0.000000e+00 sec.
Power: relative residual    = 9.997483e-08
```

5. **Using the proper iterative solver available in the LIS library compute the smallest eigenvalue $\lambda_{\min}$ of A up to a tolerance of 10^{-7}. Report the computed eigenvalue and the number of iterations required to achieve the prescribed tolerance.**

   Answer:

```
$ ./eigen1 Aex2.mtx eigvec.txt hist.txt -e ii -etol 1.e-7 -
    emaxiter 50000

number of processes = 1
matrix size = 100 x 100 (396 nonzero entries)

initial vector x      : all components set to 1
precision             : double
eigensolver           : Inverse
convergence condition : ||lx-(B^-1)Ax||_2 <= 1.0e-07 * ||lx||
    _2
matrix storage format : CSR
shift                 : 0.000000e+00
linear solver         : BiCG
preconditioner        : none
eigensolver status    : normal end

Inverse: mode number          = 0
Inverse: eigenvalue           = 1.913297e+00
Inverse: number of iterations = 1348
Inverse: elapsed time         = 2.169116e-02 sec.
Inverse:   preconditioner     = 1.313231e-03 sec.
Inverse:     matrix creation  = 2.978600e-05 sec.
Inverse:   linear solver      = 1.501240e-02 sec.
Inverse: relative residual    = 9.946932e-08
```

References

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

Index