Numerical Linear Algebra - Notes - v0.6.0

260236

November 2024

Preface

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

- Course slides. [1]

About:

GitHub repository

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

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

Contents

1 Preliminaries **5**
- 1.1 Notation . . . 5
- 1.2 Matrix Operations . . . 6
- 1.3 Basic matrix decomposition . . . 8
- 1.4 Determinants . . . 10
- 1.5 Sparse matrices . . . 11
  - 1.5.1 Storage schemes . . . 11

2 Iterative methods for linear systems of equations **15**
- 2.1 Why not use the direct methods? . . . 15
- 2.2 Linear iterative methods . . . 17
  - 2.2.1 Definition . . . 17
  - 2.2.2 Jacobi method . . . 20
  - 2.2.3 Gauss-Seidel method . . . 21
  - 2.2.4 Convergence of Jacobi and Gauss-Seidel methods . . . 22
  - 2.2.5 Stationary Richardson method . . . 24
- 2.3 Stopping Criteria . . . 27
- 2.4 Preconditioning techniques . . . 29
  - 2.4.1 Preconditioned Richardson method . . . 30
- 2.5 Gradient method . . . 31
- 2.6 Conjugate Gradient method . . . 32
- 2.7 Krylov-space . . . 35
  - 2.7.1 BiConjugate Gradient (BiCG) and BiCGSTAB method . . . 37
  - 2.7.2 Generalized Minimum Residual (GMRES) method . . . 39

3 Solving large scale eigenvalue problems **41**
- 3.1 Eigenvalue problems . . . 41
- 3.2 Power method . . . 43
  - 3.2.1 Deflation method . . . 45
- 3.3 Inverse power method . . . 46
  - 3.3.1 Inverse power method with shift . . . 47
- 3.4 QR Factorization . . . 48
  - 3.4.1 Schur decomposition applied to QR algorithm . . . 51
  - 3.4.2 Hessenberg applied to QR algorithm . . . 54
- 3.5 Lanczos method . . . 56

4 Numerical methods for overdetermined linear systems and SVD **59**
- 4.1 Overdetermined systems and Least Squares . . . 59
- 4.2 Singular Value Decomposition (SVD) . . . 61

5 Multigrid methods **64**
- 5.1 Idea of MG methods . . . 64
- 5.2 How it works . . . 65
  - 5.2.1 Coarse Grids . . . 66
  - 5.2.2 Correction . . . 69
  - 5.2.3 Interpolation Operator . . . 70
  - 5.2.4 Restriction Operator . . . 74
  - 5.2.5 Two-Grid Scheme . . . 76

5.2.6 V-Cycle Scheme 78
5.3 Classical Algebraic Multigrid (AMG) 81

Index **87**

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**:

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

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

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

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

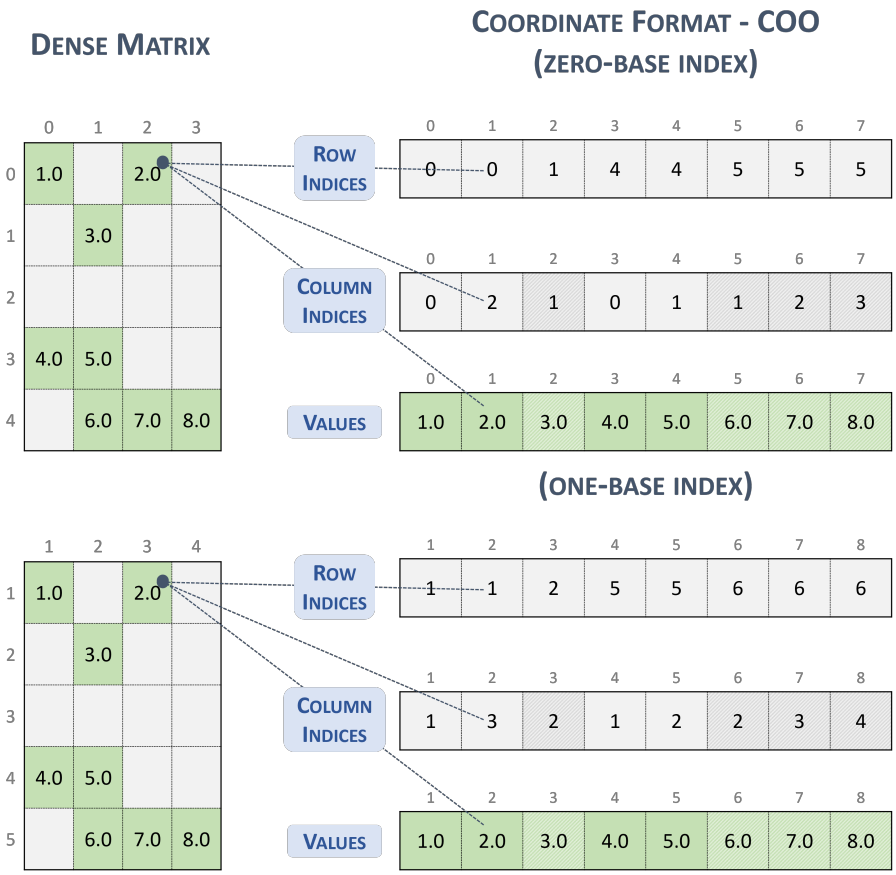


Figure 1: Graphical representation of the coordinate format (COO) technique. From the figure we can see the representation of the `AA` array, called *values*, the `JR`, called *row indices*, and finally the `JC`, called *column indices*. The algorithm is very simple. The figures are taken from the NVIDIA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

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

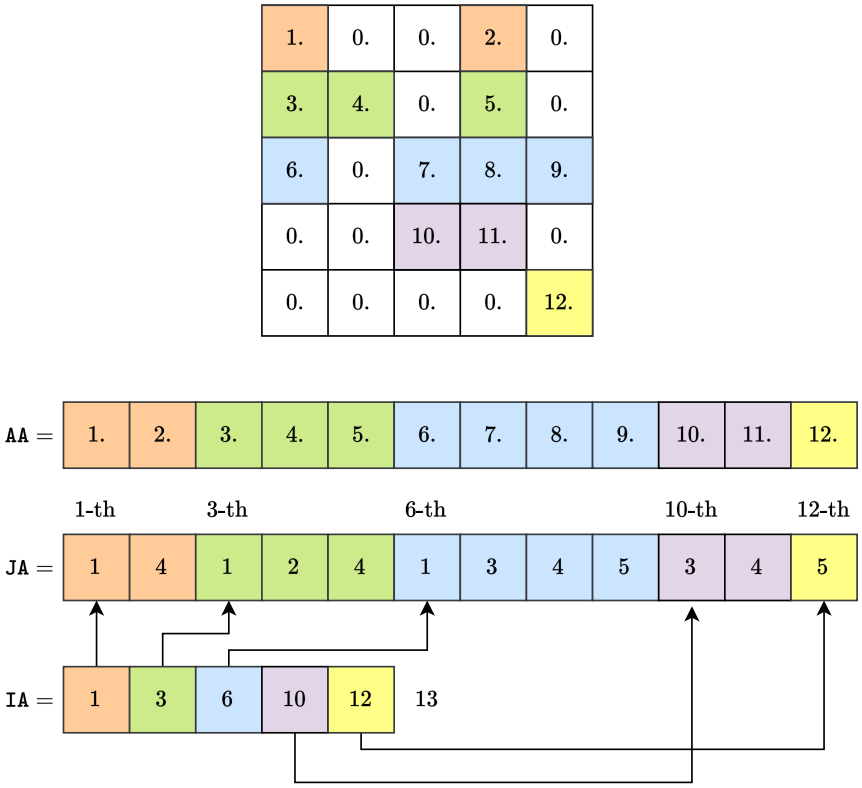


Figure 2: View an illustration of the CRS technique using colors to improve readability.

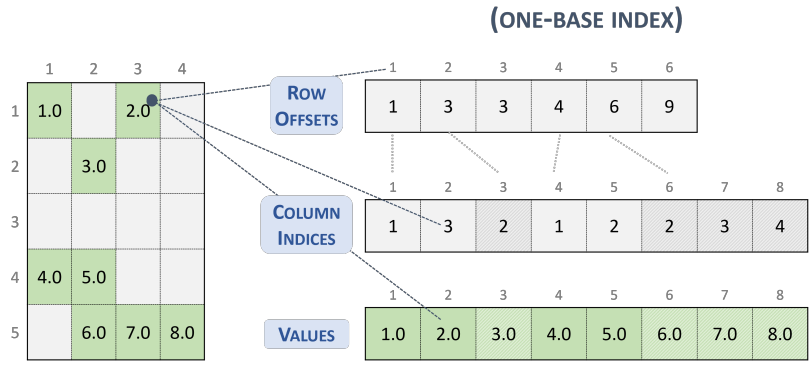


Figure 3: Graphical representation of the coordinate compressed sparse row (CSR) technique. From the figure we can see the representation of the `AA` array, called *values*, the `IA`, called *row offset*, and finally the `JA`, called *column indices*. It's interesting to see how the empty line case is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

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

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:

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

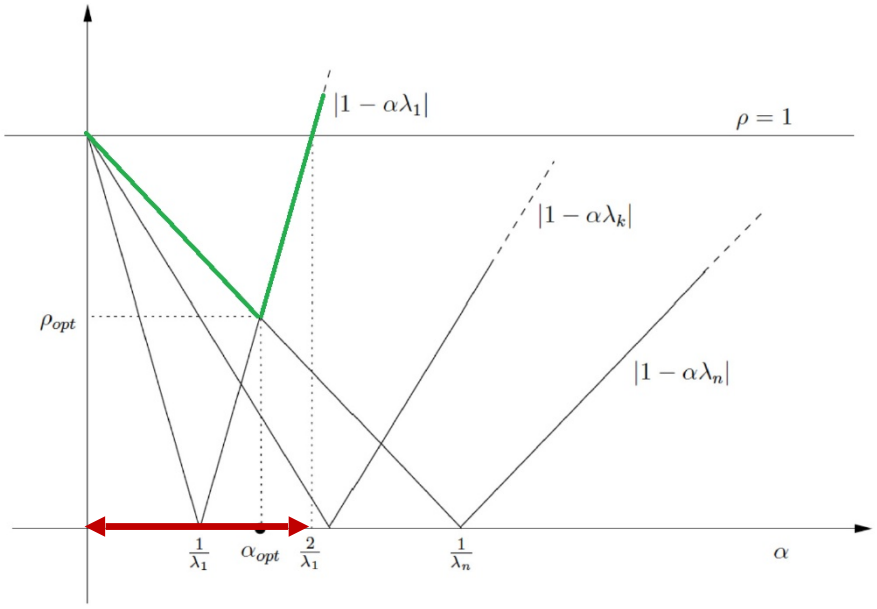


Figure 4: Graphical representation of the optimal alpha to choose in the stationary Richardson method.

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:

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

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:

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

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

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

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

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

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

  Mathematically, define:

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

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

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

Indeed:

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

2.4 Preconditioning techniques

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

The optimal spectral radius 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 $|\lambda_2|$.

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

$ How much does it cost?

It depends on the matrix used:

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

Can it be parallelized?

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

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

3.2.1 Deflation method

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

Mathematical point of view

Suppose we have computed an eigenvalue 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

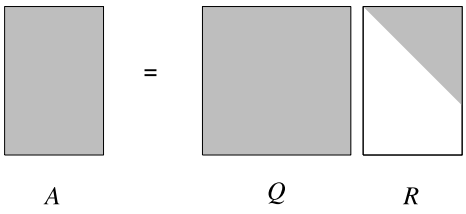


Figure 5: Full QR Factorization.

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

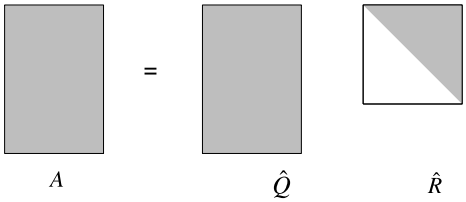


Figure 6: Reduced QR Factorization.

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 is called **Classical Gram-Schmidt (CGS, or simply GS)**, and the following one is called **Classical Gram-Schmidt (CGS, or simply GS)**:

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

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

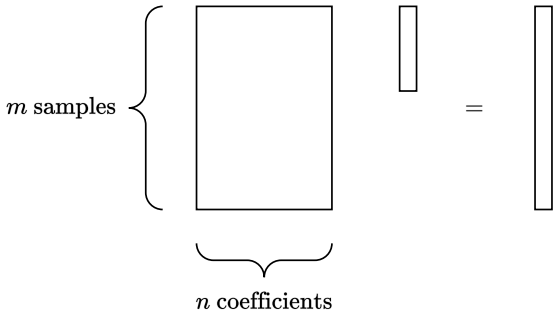
4.1 Overdetermined systems and Least Squares

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:

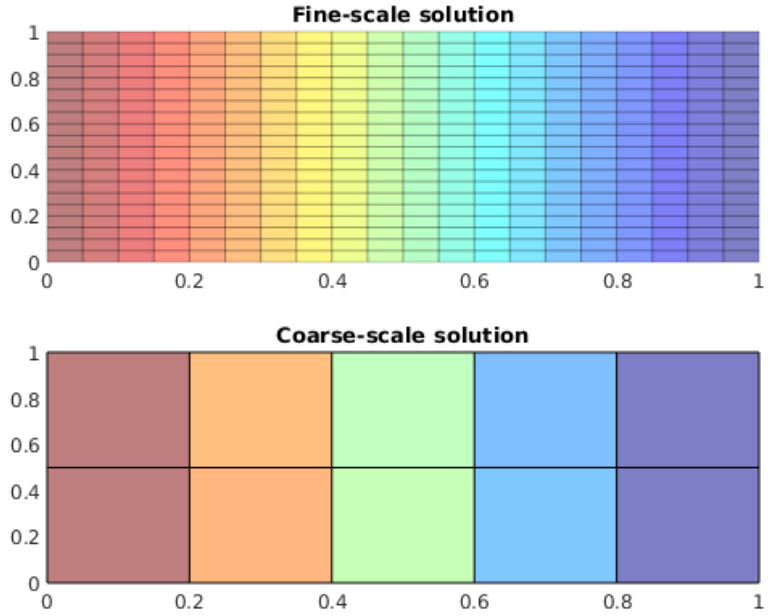
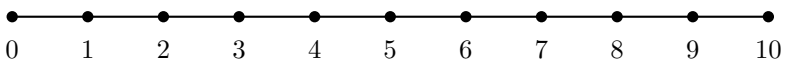


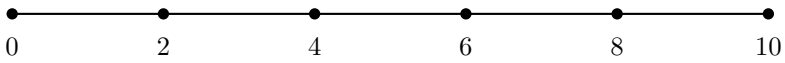
Figure 7: Difference between Coarse and Fine Grid.

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.

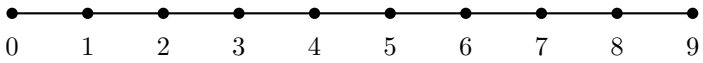


- **Coarse Grid** has a lower resolution, then fewer points spaced farther apart.

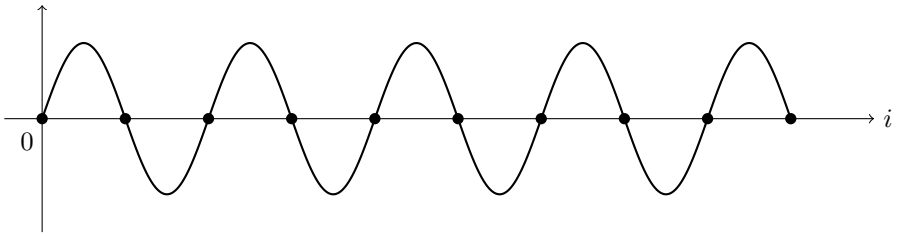


As we **move from the fine grid to the coarse grid, the mode becomes more oscillatory because the same error pattern spans fewer points, increasing its apparent frequency**. The term "mode" refers to the different error patterns or components in the solution. See the following illustration for a 100%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

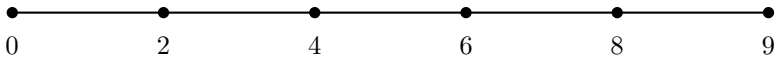
Consider a wave function on the fine grid $w_j = \sin\left(\frac{j\pi}{n+1}i\right)$ (where j determines the frequency, n is the number of points, and i is the index of the grid point), its 1D representation, and the signal:



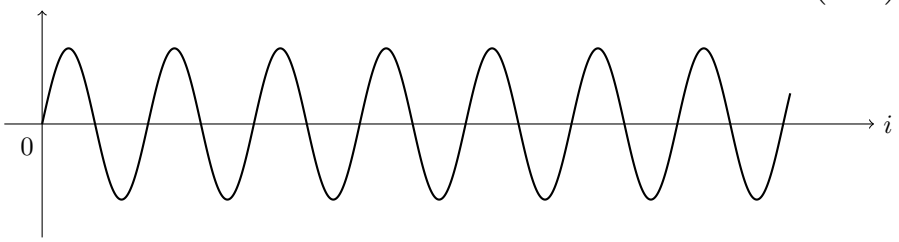
$j = 10 \qquad n = 10 \qquad w_{10} = \sin\left(\frac{10\pi}{11}i\right)$

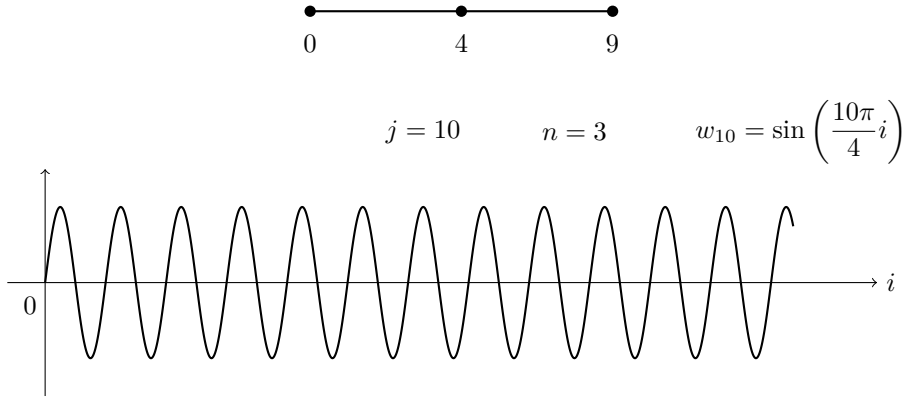


If we pass from the Fine Grid to Coarse Grid reducing the number of points, for example from 10 to 6, we obtain an increase of the oscillatory and also the same error pattern is repeated several times:



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





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:

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

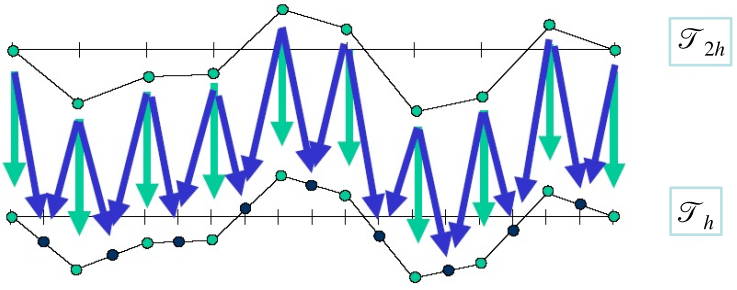
And it is multiplied by the coarse grid $\mathbf{v}_{2h}$ to get the fine grid with the interpolated values $\mathbf{v}_h$:

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

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.

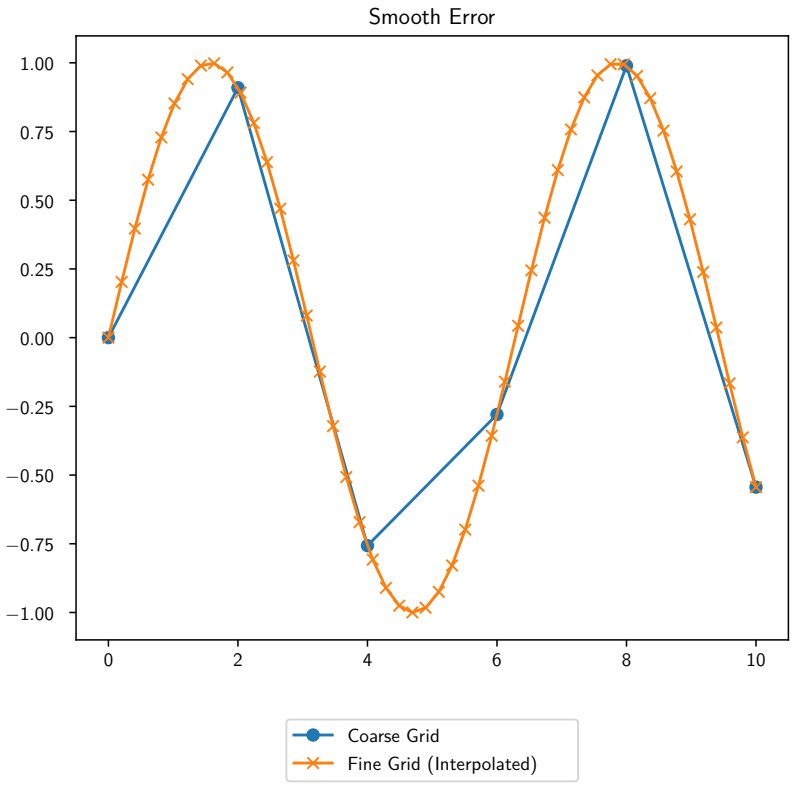


Figure 8: The figure shows what happens when we encounter a smooth error. As we can see, the coarse grid error gives a good representation of the exact error of the fine grid. As the error changes gradually, the application of the interpolation from the coarse grid to the fine grid guarantees the preservation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

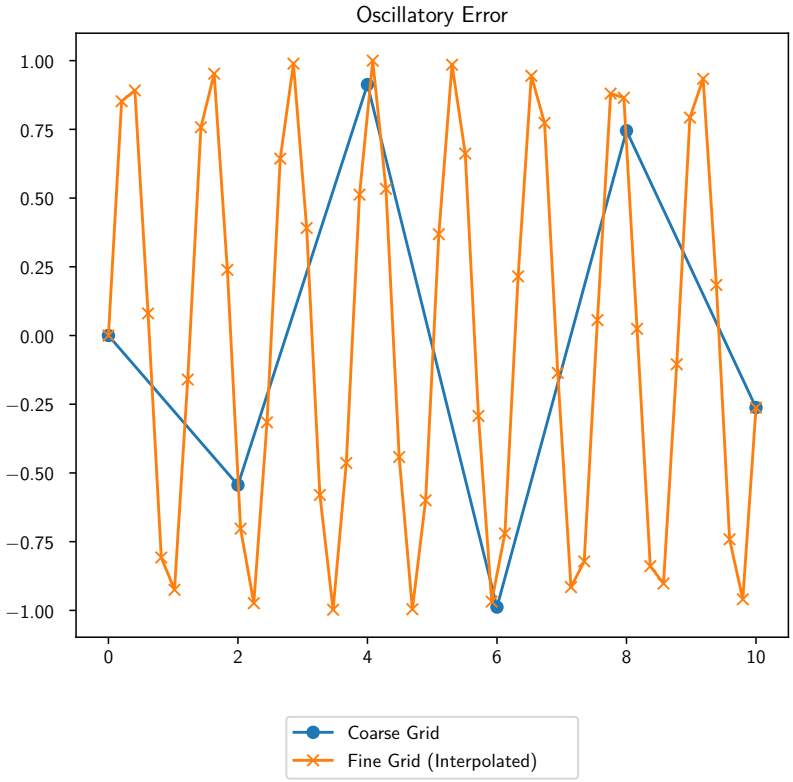


Figure 9: The figure shows what happens when we encounter an oscillatory error. As we can see, the coarse grid error is a very poor representation of the exact fine grid error. This is because the oscillatory errors are smoothed out when interpolating from the coarse grid. The averaging process inherent 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 $\mathbf{y}_h^{(\nu_1+1+\nu_2)}$.

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

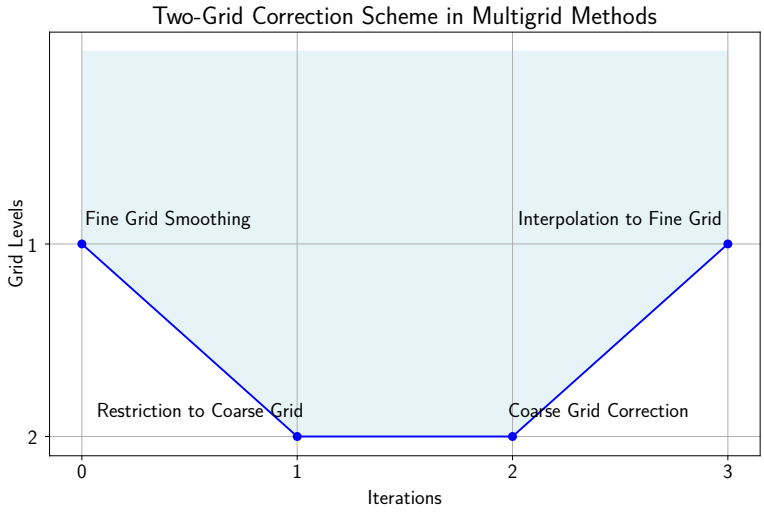


Figure 10: Graphical representation of the Two-Grid Scheme.

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.

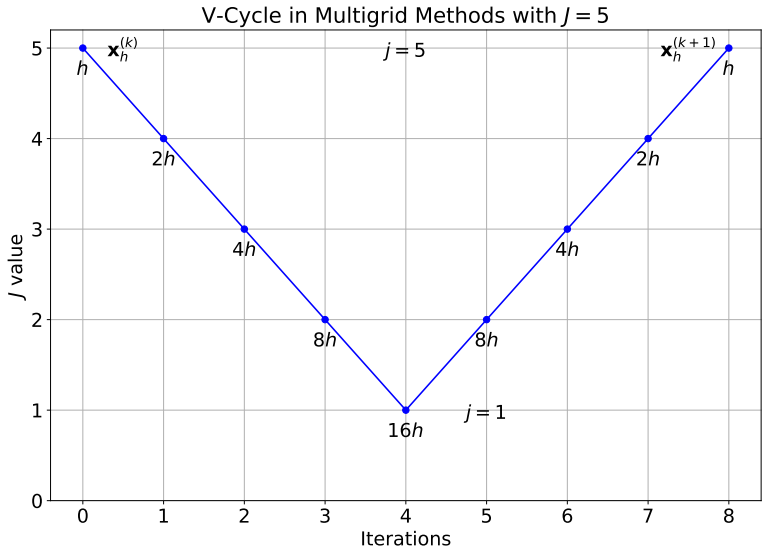


Figure 11: V-Cycle Scheme.

$ 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?).

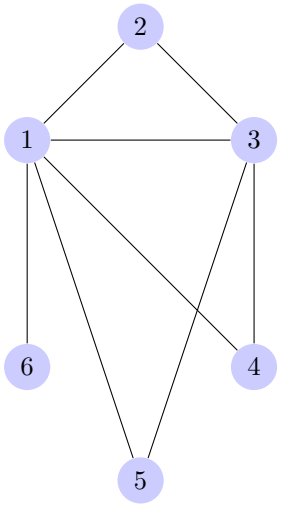
The **main goal of the AMG method is its *geometric independence***. Unlike geometric multigrid methods, which rely on the geometric structure of the problem (grid spacing, shape, etc.), AMG **constructs its grid hierarchies based purely on the algebraic structure of the system matrix**. This makes it 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.

References

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

Index

B

BiConjugate Gradient (BiCG) method 37
BiConjugate Gradient Stabilized (BiCGSTAB) method 38

C

Classical Algebraic Multigrid (AMG) 81
Classical Gram-Schmidt (CGS, or simply GS) 50
Conjugate Gradient method (GC) 32
Convergence property 16
Coordinate Compressed Sparse Row format (CSR) 12
Coordinate format (COO) 11

D

Deflation method 45
Diagonalizable matrix 42
Distance between consecutive iterates criteria 27

G

Generalized Inverse of a matrix 62
Generalized Minimum Residual (GMRES) method 39
Gram-Schmidt orthogonalization 50

H

Hessenberg matrix 54
Householder reflection 63

I

Idempotent Matrices 6
Inverse Power method 46
Inverse Power method with shift 47
Invertible Matrices 6
Iterative Method 15

L

Lanczos algorithm 56
Least Squares method 59
Lower triangular matrix 7

M

Matrices Multiplication 6
Matrix Associativity Property 6
Matrix Distributive Property 6
Moore-Penrose pseudo-inverse matrix 62
Multigrid (MG) methods 64

N

Nilpotent Matrices 6
Non-singular Matrices 6

O
Orthogonal Matrices 7
Orthogonal Vectors 6
Overdetermined linear system 59

P
Power method 43

Q
QR Factorization 48

R
Residual-based stopping criteria 27

S
Schur decomposition applied to QR algorithm 51
Singular Matrices 6
Singular Value Decomposition (SVD) method 61
Sparse Matrix 11, 15
SPD (Symmetric Positive Definite) 18
Spectral radius of a matrix 41
Spectrum of a matrix 41

T
Transpose product between matrices 6
Tridiagonal matrix 23

U
Unitary lower triangular matrix 7
Unitary upper triangular matrix 7
Upper triangular matrix 7