Iterative methods for large and sparse linear systems

Mahmoud Elsawy and Jean Luc Bouchot

Centre Inria d'Université Côte d'Azur

January 7, 2026

Outline

What this course is

Overall, this course will talk about

- Mathematics
- (Numerical) Linear algebra
- Linear systems
- Empirical evaluations through implementations
- Basic programming

This is a week long project-based course: progress on your own, as we slowly detail things.

What this course is not

Remember this course is not

- A CS / programming course – Even though implementation is an integral part of the deliverables expected, the emphasis is not on clean code or good IT practices
- A proof-heavy course
- An exhaustive presentation of all methods for solving linear systems
- targetting a precise application

Course setting

Plan:

- M./Tu./F. 8am-12-noon; 1:30pm-5:30pm
- W. 1:30pm-5:30pm
- Th. 8am-12 noon
- Group evaluation on Friday afternoon

Evaluation:

- Groups of 3 – to be chosen among yourselves and notified to us by Monday afternoon
- Work presentation : 10 minutes / group + 5 minutes of questions
- Deliverables sent to us no later than Friday 9/01/25, 12 noon:
  - All codes as an archived file (LastName1_LastName2_SLS.tar or zip)
  - A brief report describing your results
  - A small presentation used as a support for the oral presentation

More on the evaluation

What is the absolute minimum (= passing grade) the report should contain:

- Solutions to in-slides question "Now you try"
- Comments on convergence

What is the absolute minimum your code should contain:

- All routines/codes asked for in the slides
- All code used for generating your report and presentation's contents

To get over the top, be creative in analysing methods. You may include

- Study of convergence (with respect to time, number of iterations, type of structure CSR vs COO)
- Cases of convergence of the various methods (in terms of types of matrices, spectral radius)
- Memory limitations
- and much more....
- Why not invent your own method?!

Linear System Definition

- Given a square matrix A of size $n \times n$ and a vector b of size $n \times 1$, find a vector x of size $n \times 1$ that satisfies:

$$Ax = b$$

- Applications:
    - Engineering: Structural analysis, fluid dynamics, heat transfer
    - Physics: Electromagnetism, quantum mechanics
    - Computer Graphics: Image processing, computer vision
    - Machine Learning: Solving linear regression problems

Direct approach cost

- The direct way is to solve the system $Ax = b$ as

$$x = A^{-1}b$$

- This involves computing the inverse of matrix A, A^{-1}.
- Computing the inverse of a general matrix A has a computational cost of $O(n^3)$ using standard algorithms.
- For large systems (large values of n), this cubic complexity can lead to prohibitively long computation times.
- Memory requirements: storing and manipulating the entire matrix can be memory-intensive, especially for sparse matrices
- Numerical issues (like round-off errors) can accumulate during the inversion process, potentially leading to inaccurate results.

Why iterative methods?

Iterative methods as an alternative

- Avoid the explicit computation of the matrix inverse
- Can provide approximate solutions within a desired tolerance, which may be sufficient for many practical applications
- Often exploit the structure of the matrix (e.g., sparsity) to reduce computational cost and memory usage

Fixed Point Problem: A Simple Analogy

- **Imagine a mirror:** You stand in front of a mirror. Your image in the mirror is a "reflection" of you.
- **A fixed point** In this analogy, the fixed point is the position where you and your image perfectly overlap. You are "fixed" in that position relative to your reflection.
- **Finding the fixed point**
  - You might initially stand slightly off-center.
  - You then adjust your position slightly to try and align with your image.
  - You continue making small adjustments until you find the position where you and your image perfectly coincide.

Fixed Point Problem: A Simple Analogy

- A fixed point of a function $G(x)$ is a value x^* such that:
$$G(x^*) = x^*$$
- Choose an initial guess $x^{(0)}$.
- for $k = 0, 1, 2, ...$
$$x^{(k+1)} = G(x^{(k)})$$
- Stop the iteration when:
$$\lim_{k \to \infty} ||x^{(k+1)} - x^{(k)}|| = 0$$
where $|| \cdot ||$ denotes a suitable norm (e.g., Euclidean norm).
- In the context of linear systems, iterative methods work similarly.
- Reformulate the linear system $Ax = b$ as a fixed-point problem: $x = G(x)$.
- We start with an initial guess for the solution.
- We refine this guess by applying a specific rule "G" until we converge to the "fixed point"

General Formula for Iterative Methods

Let $A \in \mathbb{R}^{n\times n}$ be invertible and $b \in \mathbb{R}^n$. We denote by $\bar{x} = A^{-1}b \in \mathbb{R}^n$ **the** solution to the linear system $Ax = b$.
Iterative methods work by constructing a sequence of estimates (or approximations) $x^{(k)}$, $k \in \mathbb{N}$ as follows

- Initialisation: Pick an initial value $x^{(0)} \in \mathbb{R}^n$.
- Iterate: For any $k \in \mathbb{N}, k > 0$, compute

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

where $G : \mathbb{R}^n \to \mathbb{R}^n$ maps one estimate to the next one.

Jacobi Method: Introduction

- Consider the system of linear equations:

$$Ax = b$$

  - $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$, $b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$ and $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

- Find an iterative method to approximate the solution vector x.

Decomposition of Matrix A= D-L-U

Diagonal matrix $$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Strictly Lower Triangular (L) $$-L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

Strictly Upper Triangular (U) $$-U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Jacobi Iteration

$$(D - L - U)x = b$$

$$Dx = b + (L + U)x$$

$$x = D^{-1}b + D^{-1}(L + U)x$$

- Jacobi Iteration Formula

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b$$

  where:
  - $x^{(k)}$ is the current approximation of the solution vector at iteration k.
  - $x^{(k+1)}$ is the updated approximation at iteration $k + 1$.
- Component-wise Form:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \quad \text{for } i = 1, 2, ..., n$$

Convergence of the Jacobi Method

- Sufficient condition for convergence
  - The Jacobi method is guaranteed to converge if the matrix A is strictly diagonally dominant
  - A matrix A is strictly diagonally dominant if:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all } i = 1, 2, ..., n$$

  - This means the absolute value of each diagonal element must be greater than the sum of the absolute values of the other elements in its row.
- Necessary and sufficient condition:
  - A more general condition for convergence is that the "spectral radius" of the iteration matrix $T = D^{-1}(L + U)$ must be less than 1:

$$\rho(T) < 1$$

where $\rho(T)$ is the spectral radius of T, which is defined as the maximum eigenvalue of T in absolute value.

Theoretical convergence of Jacobi: Now you try

Compute the first 4 iterates through Jacobi's method starting at $x_0 = 0$ of the following systems and conclude on the convergence. (assuming the true solution x^* is the vector filled with 1's, i.e. $b_i = A_i x^*$)

$$A_0 = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \quad A_1 = \begin{bmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{bmatrix} \quad A_2 = \begin{bmatrix} -3 & 3 & -6 \\ -4 & 7 & -8 \\ 5 & 7 & -9 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 4 & 1 & 1 \\ 2 & -9 & 0 \\ 0 & -8 & -6 \end{bmatrix} A_4 = \begin{bmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{bmatrix}$$

Jacobi Method Convergence Investigation: Now you try

Consider the linear system $Ax = b$, where matrix A has the following structure
$a_{ii} = 5(i+1)$ for $i = 1, 2, ..., n$ and $a_{ij} = -1$ for $i \neq j$
The right-hand side vector b is a random vector of dimension n.

- Investigate the convergence behavior of the Jacobi iterative method for this specific system for various values of n. Does strict diagonal dominance hold in this case? Will the Jacobi method always converge for this type of matrix structure?

Jacobi Method Convergence Investigation: (Minimum) Expected outcome in file jacobi_dense.py

- A function named *jacobi_method* with 5 inputs
  - a numpy array A, the matrix,
  - a numpy array b, the right hand side,
  - a numpy array $x0$, the first estimate,
  - a float *tol* with default value $1E^{-5}$ which will stop the iterations once the error gets below, and
  - a float *max_iter* to represent a maximum number of iterations

  and returns three outputs
  - a numpy array x, the approximate solution,
  - the number of iterations done (an int), and
  - the list of errors in the estimates at each iteration (list of float)
- a function *generate_linear_system* taking a single input
  - an integer n representing the dimension of the system

  and returning two outputs
  - the matrix A generated (a numpy array) and
  - the right hand side generated (a numpy array)

Sparse Matrices: Introduction

- A sparse matrix is a matrix with a high proportion of zero elements.
- Most of the entries in the matrix are zero.
- Examples:
  - Network adjacency matrices (social networks, transportation networks)
  - Finite element method (FEM) matrices
  - Discretization of partial differential equations
- Memory Efficiency (Only non-zero elements are stored)
- Many matrix operations can be performed more efficiently by exploiting the sparsity.
- Only non-zero elements need to be considered in calculations.

Importance of Sparse Matrices

- Large-scale simulations:
  - Essential for solving large systems of equations arising in scientific and engineering applications.
- Machine learning:
  - Used in various machine learning algorithms, such as support vector machines and graph neural networks.
- Data analysis:
  - Efficiently handling and analyzing large datasets with sparse representations.

Sparse Matrices: An Example

Tridiagonal Matrix Example

A tridiagonal matrix is a sparse matrix with non-zero elements only on the main diagonal, the diagonal above it (superdiagonal), and the diagonal below it (subdiagonal).

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

This structure arises frequently in numerical methods for solving differential equations, such as those encountered in finite difference methods.

Sparse Matrix Storage Formats

- Compressed Row Storage (CSR):
  - Store non-zero elements in a single array.
  - Store an array of cumulative numbers of non zero entries per row.
  - Store column indices of the non-zero elements in a third array.
- Compressed Column Storage (CSC):
  - Similar to CSR, but stores non-zero elements and indices column-wise.
- Coordinate List (COO):
  - Stores the row and column indices of each non-zero element along with its value.

Storing a Tridiagonal Matrix in COO Format

- Tridiagonal Matrix

$$\begin{bmatrix} 5 & 1 & 0 & 0 & 0 \\ 1 & 5 & 1 & 0 & 0 \\ 0 & 1 & 5 & 1 & 0 \\ 0 & 0 & 1 & 5 & 1 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$$

  - Stores non-zero values [5, 5, 5, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1]
  - row indices of non-zero values: [0, 1, 2, 3, 4, 0, 1, 2, 3, 1, 2, 3, 4]
  - column indices of non-zero values: [0, 1, 2, 3, 4, 1, 2, 3, 4, 0, 1, 2, 3]

Storing a Tridiagonal Matrix in COO Format

- Sparse matrix

Coords	Values
(0, 0)	5
(0, 1)	1
(1, 0)	1
(1, 1)	5
(1, 2)	1
(2, 1)	1
(2, 2)	5
(2, 3)	1
(3, 2)	1
(3, 3)	5
(3, 4)	1
(4, 3)	1
(4, 4)	5

Storing a Tridiadonal Matrix in CSR and CSC formats

The previous matrix

$$\begin{bmatrix} 5 & 1 & 0 & 0 & 0 \\ 1 & 5 & 1 & 0 & 0 \\ 0 & 1 & 5 & 1 & 0 \\ 0 & 0 & 1 & 5 & 1 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$$

can be written as

- CSR:
  - Data: $[5, 1, 1, 5, 1, 1, 5, 1, 1, 5, 1, 1, 5]$
  - Row indices: $[0, 2, 5, 8, 11, 13]$
  - Col indices: $[0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4]$
- CSC:
  - Data: $[5, 1, 1, 5, 1, 1, 5, 1, 1, 5, 1, 1, 5]$
  - Row indices: $[0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4]$
  - Col indices: $[0, 2, 5, 8, 11, 13]$

Storing matrices in sparse format: Now you try

Convert the following three matrices in all three of the format introduced

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 0 & 5 & 0 & 0 \\ 0 & 6 & 0 & 7 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 0 & 5 & 0 \\ 0 & 6 & 0 & 7 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Comment on the memory usage of each approaches. When is one advantageous over another? Over a plain dense matrix? Generalise your results in the case of triangular matrices and tridiagonal matrices of size n.

Preconditioning

- Preconditioning is a technique used to accelerate the convergence of iterative methods for solving linear systems of the form $Ax = b$.
- The core idea is to transform the original system into an equivalent system that is better conditioned, meaning it converges more rapidly under iterative methods.
- We introduce a "preconditioner matrix" C (invertible) and transform the original system as follows:

$$Ax = b$$
$$C^{-1}Ax = C^{-1}b$$

Examples for preconditioning

- Jacobi: $C = D$, where D is the diagonal of matrix A

$$\begin{aligned} D^{-1}(D-(L+U))x &= D^{-1}b \\ \left(I - D^{-1}(L+U)\right)x &= D^{-1}b \\ x &= D^{-1}b + D^{-1}(L+U)x \end{aligned}$$

Iterative form

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right) \quad \text{for } i = 1, 2, ..., n$$

Sparse and dense Jacobi: Now you try, file dense_sparse_jacobi.py

- Two functions named *jacobi_dense* (similar to the previous one) and *jacobi_sparse* with 5 inputs as before (except the A matrix should be a csr matrix in the case of the sparse implementation) and returning three outputs
    - a numpy array x, the approximate solution,
    - the number of iterations done (an int), and
    - the time taken for the solution
- a function *generate_corrected_sparse_tridiagonal_matrix* generating a tridiagonal matrix, taking two inputs
    - an integer n representing the dimension of the system
    - two floats, the first one for the diagonal entries, and the second for the upper and lower diagonal

  and returning three outputs
    - the sparse matrix A generated as a csr_matrix and
    - the equivalent dense matrix (a numpy array, implemented via a for loop, not with *to_array*)
    - a random right hand side b

Gauss-Seidel Iteration Derivation

- Gauss-Seidel: $C = D - L$ or $C = D - U$
- Applying the preconditioner $C = D - L$ to the linear system $Ax = b$:

$$\begin{aligned} (D-L)^{-1}(D-U-L)x &= (D-L)^{-1}b \\ (I-(D-L)^{-1}U)x &= (D-L)^{-1}b \\ x &= (D-L)^{-1}b + (D-L)^{-1}Ux \end{aligned}$$

Iterative form:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \quad \text{for } i = 1, 2, \ldots, n$$

Now you try: file GS_jacobi_sparse.py

We now compare Jacobi with Gauss-Seidel's method in terms of convergence. Your goal for this exercise is to implement

- 2 functions for generating matrices:
  - generate_simple_sparse_tridiagonal_matrix with three inputs (dimension, diag value, off diag value) and three outputs (csr matrix, full matrix, random b vector). The returned matrix is tridiagonal.
  - generate_sparse_tridiagonal_matrix with one single input (n), the dimension of the output matrix and the same three outputs as before. The output matrix represents the 2nd order finite difference Laplacian operator in dimension 1 (n+1 subintervals of the [0,1] interval with boundary conditions 0 at 0 and 1)
- 2 functions implementing solvers:
  - *jacobi_sparse_with_error* implementing Jacobi's iterations for csr matrices. It has an extra (6 in total) parameter, *x_exact*, the true solution, used to compute $\|x^{(k)} - x_{\text{exact}}\|, \forall k$. It returns three outputs: the found solution, the number of iterations, the list of $\|x^{(k)} - x_{\text{exact}}\|$.
  - *gauss_seidel_sparse_with_error* implementing the Gauss Seidel iterations with the same inputs and outputs as the Jacobi function above
- All useful plotting functions showing the rate of convergence.

Successive Over-Relaxation (SOR) Method

Preconditioning Matrix for SOR:
The SOR method can be viewed as a preconditioned iterative method with the preconditioner with relaxiation parameter ω $(0 < \omega < 2)$

$$C = \frac{1}{\omega}(D - \omega L)$$

Starting with the preconditioned system $C^{-1}Ax = C^{-1}b$:
SOR iterative formula:

$$s_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

$$x_i^{(k+1)} = \omega s_i^{(k)} + (1-\omega)x_i^{(k)}$$

Symmetric Successive Over-Relaxation (SSOR)

Preconditioning Matrix for SSOR:

$$C = \frac{1}{\omega}(D - \omega L)D^{-1}(D - \omega U)$$

The precondidioing system written as

$$C^{-1}Ax = C^{-1}b$$

$$\frac{1}{\omega}(D - \omega U)^{-1}D(D - \omega L)^{-1}(D - L - U)x = \frac{1}{\omega}(D - \omega U)^{-1}D(D - \omega L)^{-1}b$$

Symmetric Successive Over-Relaxation (SSOR)

- Introduce intermediate variable:

$$z = \frac{1}{\omega}(D - \omega L)^{-1} b$$

- One can obtain

$$(D - \omega U)x = Dz$$

- where

$$(D - \omega L)z = b$$

Symmetric Successive Over-Relaxation (SSOR)

Solving the preconditioning system

- Solve for an intermediate variable (forward)

$$(D - \omega L)z = b$$

$$z_i^{(k)} = \frac{1}{a_{ii}} \left(\omega \sum_{j=1}^{i-1} a_{ij} z_j^{(k)} + b_i \right) \quad \text{for } i = 1, 2, ..., n$$

- Solve for the solution (backward)

$$(D - \omega U)x = Dz$$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(\omega \sum_{j=i+1}^{n} a_{ij} x_j^{(k+1)} + z_i^{(k)} \right) \quad \text{for } i = n, n-1, ..., 1$$

Ideas for project experiments

- Compare the convergence speed between J and GS for tridiagonal matrices (and compute the spectral radii of the iteration matrices)
- Compare convergence speeds for matrices A_3 and A_4 from pen and paper exercises
- Compare convergence speeds for with respect to spectral radius (e.g. create a matrix with a parameter on the diagonal which controls it)
- Compare the speed of convergence (in time / iterations) of the methods using 3 for loops (iteration/row index / column index) vs 2 loops (iterations / row index + dot product) vs 1 loop (iterations + matrix handling)
- Implement proper index handling for CSR matrices
- Analyze the impact of the spectral ratidus on the convergence: what is the slope of the lines in the semilog graph? Use, for instance, *numpy polyfit*.
- Can these methods be used in image processing tasks for (de)blurring? Apply your approaches to the discretization of the heat equation. Compare to pen and paper solutions.