Numerical Optimization for Large Scale Problems
Lecture Notes

Francesco Della Santa [*]

Politecnico di Torino, A.Y. 2024/2025

Abstract

Notes of the laboratories held by Francesco Della Santa for the course of Numerical Optimization for Large Scale Problems and Stochastic Optimization.

Contents

[*]Dipartimento di Scienze Matematiche, Politecnico di Torino, Turin, Italy

Introduction

In this document, we report the text of the laboratories for the course of *Numerical Optimization for Large Scale Problems and Stochastic Optimization*, A.Y. 2024/2025. Each section corresponds to one laboratory and it will consist of a brief theoretical recap of the arguments, followed by the exercises.

The solutions of the exercises of each laboratory are published (on the course web page) periodically, before the start of the next one. Solutions will be published only as MATLAB code (except for the first laboratory, published also as Python code).

Remark 0.1 (Material for refreshing knowledge: MATLAB and basic numerical methods for Linear Algebra)**.** In Appendix A, we report the content of the message of Prof. Pieraccini on the course web page and related to the available material about "*MATLAB pills*" and "*Background*". Look at that material to refresh your knowledge about MATLAB and basic numerical methods for Linear Algebra, since they are strongly suggested for the laboratories (and the course in general).

1 Direct Methods for Linear Systems (Lab. 1)

Let us consider a linear system, with coefficients in $\mathbb{R}$, of m equations and n unknowns

$$A\boldsymbol{x} = \boldsymbol{b}\,, \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$ denotes the matrix of the *coefficients*, $\boldsymbol{x} \in \mathbb{R}^n$ denotes the vectors of the *unknowns*, and $\boldsymbol{b} \in \mathbb{R}^m$ is the vector of *known terms*.

Then, we recall that:

1. **Rouché-Capelli's Theorem:** system (1) admits solutions if and only if

   $$\operatorname{rank}(A) = \operatorname{rank}(A|\boldsymbol{b})\,. \tag{2}$$

   In particular, if (2) holds, (1) admits $\infty^{n-\operatorname{rank}(A)}$ solutions (one solution if $\operatorname{rank}(A) = n$).

2. If $\boldsymbol{b} = \boldsymbol{0}$, the system (1) is defined as *homogeneous system*. Due to item 1, a homogeneous system always admits solutions (trivial solution: $\boldsymbol{x}^* = \boldsymbol{0}$).

3. If $\boldsymbol{b} \neq \boldsymbol{0}$ The set of all and only the solutions of system (1) is

   $$\mathcal{S}_{\boldsymbol{b}} = \{\,\bar{\boldsymbol{x}} + \boldsymbol{x}^* \mid \boldsymbol{x}^* \in \mathcal{S}_{\boldsymbol{0}}\}\,, \tag{3}$$

   where $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ is a particular solution of (1) (i.e., $A\bar{\boldsymbol{x}} = \boldsymbol{b}$) and $\mathcal{S}_{\boldsymbol{0}} \subseteq \mathbb{R}^n$ is the set of all and only the solutions of the homogeneous system $A\boldsymbol{x} = \boldsymbol{0}$.

1.1 Square Matrices

Here, we focus on the case of linear systems like (1) but where the number of equations is equal to the number of unknowns. Therefore, the matrix of the coefficients is a square matrix $A \in \mathbb{R}^{n\times n}$ and the vector of known terms is $\boldsymbol{b} \in \mathbb{R}^n$ (i.e., $m = n$).

We recall the following types of square matrices:

- **Diagonal:** A is diagonal if $a_{ij} = 0$, for each $i \neq j$.

  $$A = \begin{bmatrix} \bullet & & \\ & \ddots & \\ & & \bullet \end{bmatrix}.$$

- **Lower/Upper Triangular:** A is lower/upper triangular if $a_{ij} = 0$, for each $i \lessgtr j$.

  $$A_{lt} = \begin{bmatrix} \bullet & & \\ \vdots & \ddots & \\ \bullet & \cdots & \bullet \end{bmatrix}, \quad A_{ut} = \begin{bmatrix} \bullet & \cdots & \bullet \\ & \ddots & \vdots \\ & & \bullet \end{bmatrix}.$$

- **Symmetric:** A is symmetric if it is equal to its transpose (i.e., $A = A^T$).

- **Positive/Negative (Semi-)Definite:** A is positive/negative definite if $\boldsymbol{x}^T A\boldsymbol{x} \gtrless 0$, for each $\boldsymbol{x} \in \mathbb{R}^n \backslash \{\boldsymbol{0}\}$. A is positive/negative semi-definite if $\boldsymbol{x}^T A\boldsymbol{x} \gtreqless 0$, for each $\boldsymbol{x} \in \mathbb{R}^n \backslash \{\boldsymbol{0}\}$.

- **Invertible and Inverse Matrix:** A is *invertible* if exists a matrix $B \in \mathbb{R}^{n\times n}$ such that $AB = BA = \mathbb{I}_n$. Then, B is defined as *inverse matrix* of A and is denoted by A^{-1}.

  A matrix that is not invertible is also called *singular*; then, saying that a matrix is *non-singular* is equivalent to saying that it is *invertible*.

- **Orthogonal:** A is *orthogonal* if its inverse is equal to its transpose (i.e., $A^{-1} = A^T$ and $AA^T = A^T A = \mathbb{I}_n$).

1.2 Gaussian Elimination

Here, we briefly recall the *Gaussian Elimination* properties for matrices and linear systems. First of all, we recall that the gaussian elimination operations are:

- **Switch:** switch row i and row j of a matrix;
- **Multiplication by Scalar:** multiply the i-th row of a matrix by a scalar value $\lambda \in \mathbb{R}$;
- **Add Row to Another:** add row i, multiplied by a scalar $\lambda \in \mathbb{R}$, to row j of a matrix.

We recall that the gaussian elimination operations have the following properties:

- They are invertible linear applications; i.e., their action can be described left-multiplying A by a non-singular matrix $E \in \mathbb{R}^{m \times m}$.
- Let $E_1, \dots, E_k$ be the ordered sequence of $k \in \mathbb{N}$ gaussian elimination operations applied to A and let $E \in \mathbb{R}^{m \times m}$ be $E := E_k \cdots E_1$. Then:
  - E is non-singular;
  - The linear system $A'\boldsymbol{x} = \boldsymbol{b}'$, with $A' = EA$ and $\boldsymbol{b}' = E\boldsymbol{b}$, has the same solutions of (1).

1.3 Square Linear Systems

Concerning square linear systems, we recall that:

1. A **invertible (i.e., non-singular):** due to Rouché-Capelli's Theorem (see item 1 above), if $\det(A) \neq 0$, then (1) admits one unique solution. Moreover, since A is invertible if and only if $\det(A) \neq 0$, the unique solution of (1) is $\boldsymbol{x}^* = A^{-1}\boldsymbol{b}$.
2. A **not invertible (i.e., singular):** if A is singular, the solutions of the linear system depends on the Rouché-Capelli's Theorem.
3. **Forward/Backward Substitution:** if A is lower/upper triangular and non-singular, the unique solution $\boldsymbol{x}^*$ can be easily computed using *forward/backward substitution*. We recall that all the diagonal elements of a non-singular triangular matrix are non-null (i.e., $a_{ii} \neq 0$, for each $i = 1, \dots, n$).
   - **Forward Substitution:**
   $$x_1^* = \frac{b_1}{a_{11}} \rightarrow x_2^* = \frac{b_2 - a_{21}x_1^*}{a_{22}} \rightarrow \cdots \rightarrow x_n^* = \frac{b_n - a_{n\,(n-1)}x_{(n-1)}^* - \cdots - a_{n1}x_1^*}{a_{nn}}.$$
   - **Backward Substitution:**
   $$x_n^* = \frac{b_n}{a_{nn}} \rightarrow x_{(n-1)}^* = \frac{b_{(n-1)} - a_{(n-1)\,n}x_n^*}{a_{(n-1)\,(n-1)}} \rightarrow \cdots \rightarrow x_1^* = \frac{b_1 - a_{12}x_2^* - \cdots - a_{1n}x_n^*}{a_{11}}.$$
4. **Substitution (Diagonal Special Case):** if A is non-singular and diagonal, the unique solution of (1) is $\boldsymbol{x}^* = (b_1/a_{11}, \dots, b_n/a_nn)$.
5. **Gaussian Elimination:** if A is non-singular we can perform a gaussian elimination $E \in \mathbb{R}^{n \times n}$ such that EA is a triangular matrix (usually, upper triangular); then, we solve the triangular linear system $(EA)\boldsymbol{x} = E\boldsymbol{b}$ by substitution.
6. **LU Factorization:** let A be non-singular and E the matrix of gaussian elimination such that $EA = U$ is upper triangular and E^{-1} is lower triangular; then, $LU = E^{-1}EA = A$ and we can solve the linear system (1) by two substitutions: $\boldsymbol{y}^* = L^{-1}\boldsymbol{b}$, $\boldsymbol{x}^* = U^{-1}\boldsymbol{y}^*$.

7. **Cholesky Factorization:** if A is symmetric and positive definite, exist a unique upper triangular matrix R, with positive diagonal elements, such that the LU factorization of A is $A = R^T R$.
8. **QR Factorization:** with proper operations (other than gaussian elimination), A can be decomposed as $A = QR$, where Q is orthogonal and R is upper triangular.

Observation: factorizations typically are more convenient than the gaussian elimination if you have to solve many linear systems with the same matrix (e.g., $A\boldsymbol{x} = \boldsymbol{b}_1, \dots, A\boldsymbol{x} = \boldsymbol{b}_N$).

Observation: the advantages of the QR factorization are:

- methods for obtaining the QR factorization are usually *more stable* (see Section 1.4 below), but more expensive, than the ones used for obtaining the LU factorization;
- QR factorization can be easily extended to *rectangular* linear systems.

1.4 Condition Number

The reliability of a solution $\boldsymbol{x}^*$ of system (1) depends on the condition number of the matrix A. We recall that the condition number of A is

$$\mathrm{k}(A) = \| A \| \| A^{-1} \| , \tag{4}$$

where $\| \cdot \|$ denotes a matrix norm (typically, it is used $\| \cdot \|_2$).

Specifically, let $A \in \mathbb{R}^{n \times n}$ be a non-singular matrix and let $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$ be two vectors such that $A\boldsymbol{x} = \boldsymbol{b}$. Moreover, let $\widehat{\boldsymbol{x}} := \boldsymbol{x} + \delta\boldsymbol{x}$ and $\widehat{\boldsymbol{b}} := \boldsymbol{b} + \delta\boldsymbol{b}$, with $\delta\boldsymbol{x} \neq \boldsymbol{0}$ and $\delta\boldsymbol{b}$ such that $A\widehat{\boldsymbol{x}} = \widehat{\boldsymbol{b}}$.

Then

$$\frac{\| \delta\boldsymbol{x} \|}{\| \boldsymbol{x} \|} \leqslant \mathrm{k}(A) \frac{\| \delta\boldsymbol{b} \|}{\| \boldsymbol{b} \|} , \tag{5}$$

where here $\| \cdot \|$ denotes a vector norm (typically, it is used $\| \cdot \|_2$).

Observation: $\mathrm{k}(A) \geqslant \| AA^{-1} \| = \| \mathbb{I}_n \| = 1$; i.e., $\mathrm{k}(A) \in [1, +\infty)$.

Observation: the smaller $\mathrm{k}(A)$, the more reliable the solution $\boldsymbol{x}^*$ of system (1). On the contrary, the larger $\mathrm{k}(A)$, the larger the upper bound for the noise of the solution and, by consequence, the lower the reliability of $\boldsymbol{x}^*$.

If a matrix A is characterized by a large condition number (i.e., $\mathrm{k}(A) \gg 1$), the matrix is *ill-conditioned*.

1.5 Direct Methods in Matlab

In Figures 1 and 2, the schemes of the Matlab procedure for solving linear systems are reported. In particular, we have two different schemes depending on the method used to store matrices: *dense* matrices and *sparse* matrices. In general, for solving a linear system in Matlab using a direct method, the `mldivide` function is used (equivalent to the operator \). Here, the online documentation.

1.6 Direct Methods in Python

For solving linear systems in Python, the `numpy` and `scipy` modules are strongly suggested; indeed, they have many built-in linear algebra tools. In general, the sub-modules `numpy.linalg` and `scipy.linalg` are almost equivalent; on the other hand, for working with sparse matrices, the sub-modules `scipy.sparse` and `scipy.sparse.linalg` are necessary. In particular, for solving linear systems you can use `numpy.linalg.solve` or `scipy.linalg.solve` for dense matrices and `scipy.sparse.linalg.spsolve` for sparse matrices. The schemes adopted by these procedures are similar to the ones of `mldivide` in Matlab.

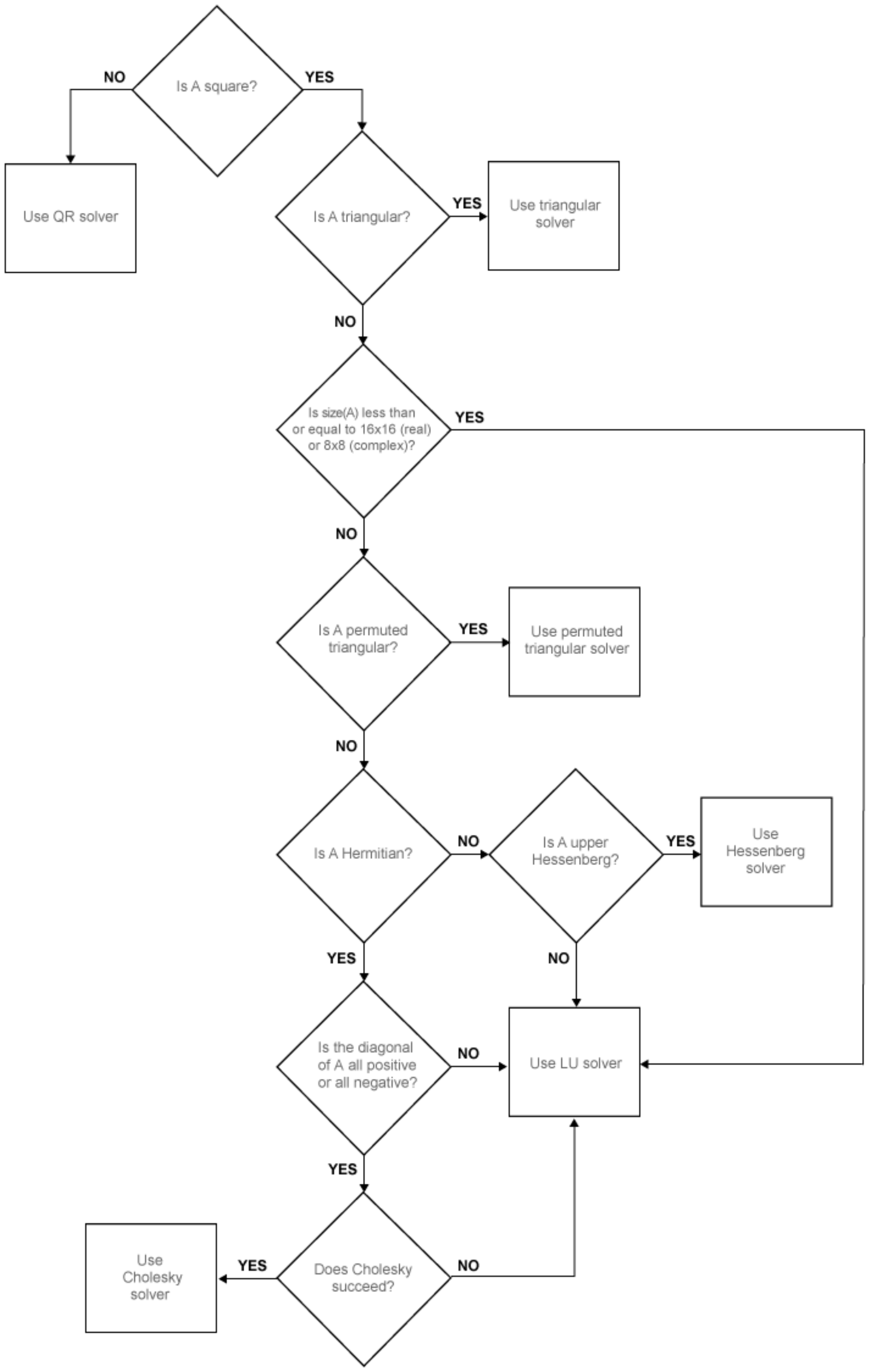

Figure 1: Scheme for the `mldivide` (\) function in Matlab (dense matrices case).

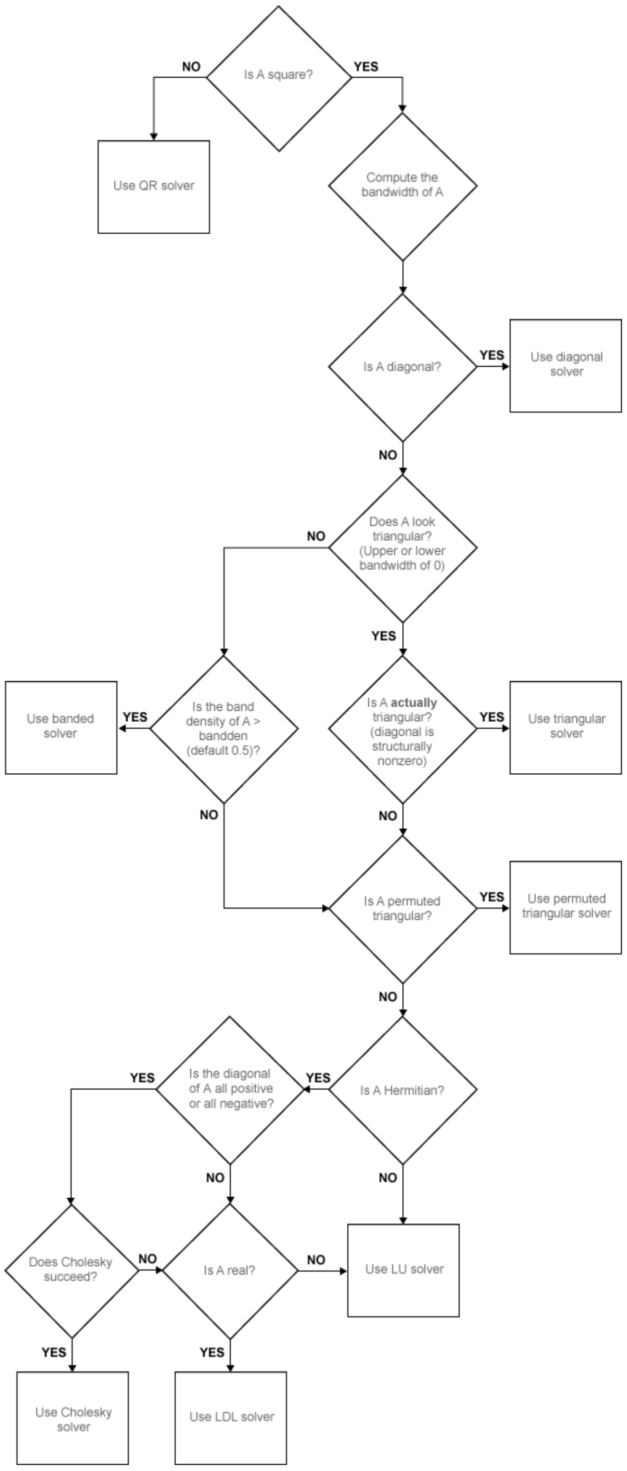
Is A square?
NO
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
Use diagonal solver
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO
YES
Is the band density of A > bandden (default 0.5)?
YES
Use banded solver
NO
Is A **actually** triangular? (diagonal is structurally nonzero)
YES
Use triangular solver
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
Is A Hermitian?
YES
Is the diagonal of A all positive or all negative?
YES
NO
NO
Use LU solver
Does Cholesky succeed?
NO
YES
Use Cholesky solver
Is A real?
NO
YES
Use LDL solver

Figure 2: Scheme for the `mldivide` (\) function in Matlab (sparse matrices case).

1.7 Fill-In Phenomenon

We recall that a matrix is *sparse* if most of its elements are equal to zero. Sparse matrices have specific storage methods for reducing memory storage without losing efficiency in matrix operations.

Nonetheless, direct methods are not suitable for very large sparse matrices, due to the *fill-in* phenomenon. In particular, we have that a factorization of a sparse matrix A can return two *dense* matrices. This phenomenon can lead to memory storage problems and/or expensive computations of the factors.

1.8 Exercises

Exercise 1.1 (Basic Linear Systems)**.** *Write a script file where:*

1. *The random seed is 23.*
2. *A matrix $A \in \mathbb{R}^{n\times n}$ is initialized with elements randomly sampled with uniform distribution in $[0,1]$. The value of n can be initialized as $n = 5$ for simplicity.*
3. *A vector $\boldsymbol{b} \in \mathbb{R}^n$ is initialized such that the exact solution of $A\boldsymbol{x} = \boldsymbol{b}$ is the vector $\boldsymbol{x}^* = \boldsymbol{e} = (1,\dots,1) \in \mathbb{R}^n$.*
4. *A set of N vectors $\boldsymbol{b}_1,\dots,\boldsymbol{b}_N \in \mathbb{R}^n$ are initialized such that $\boldsymbol{b}_i = \boldsymbol{b} + \boldsymbol{\xi}_i$, $\boldsymbol{\xi}_i \in \mathcal{U}([0,1]^n)$, for each $i = 1,\dots,N$. The value of N can be initialized as $N = 10$ for simplicity.*
5. *A diagonal matrix $D \in \mathbb{R}^{n\times n}$ is initialized with elements randomly sampled with uniform distribution in $[0,1]$.*
6. *Solve the linear systems $A\boldsymbol{x} = \boldsymbol{b}$, $D\boldsymbol{x} = \boldsymbol{b}$, and $A\boldsymbol{x} = \boldsymbol{b}_i$, for each $i = 1,\dots,N$.*
7. *Compute the norms of the residuals of the numerical solutions; i.e., $\| A\widehat{\boldsymbol{x}} - \boldsymbol{b} \|$, $\| D\widehat{\boldsymbol{x}} - \boldsymbol{b} \|$, and $\| A\widehat{\boldsymbol{x}} - \boldsymbol{b}_i \|$, for each $i = 1,\dots,N$.*
8. *Compute the condition number of A and D and comment the results.*

Exercise 1.2 (Ill-Conditioned Problem)**.** *Write a script file where you solve a linear system $A\boldsymbol{x} = \boldsymbol{b}$ such that*

$$A = \begin{bmatrix} 0.835 & 0.667 \\ 0.333 & 0.266 \end{bmatrix},$$

and $\boldsymbol{b} \in \mathbb{R}^2$ is the result of an experiment, measured using an instrument with tolerance ± 0.001. For simplicity, assume to have the following 11 measurements of $\boldsymbol{b}$:

$$\boldsymbol{b}_i = \boldsymbol{b}^* - 0.001 + (i-1)0.0002\,, \quad \forall\ i = 1,\dots,11\,,$$

where $\boldsymbol{b}^* = (0.168, 0.067)$ is the exact vector of known terms.

Then:

1. *Solve the linear systems $A\boldsymbol{x} = \boldsymbol{b}_i$, for each $i = 1,\dots,N$.*
2. *Compute the norms of the residuals of the numerical solutions; i.e., $\| A\widehat{\boldsymbol{x}} - \boldsymbol{b}_i \|$, for each $i = 1,\dots,N$.*
3. *Compute the condition number of A and comment the results.*

Exercise 1.3. *Download, run, and read the script* `lab01_sparse_examples.m/.py` *carefully. Then write a new script file where, for each* $n = 200, 400, \dots, 2000$*, the following operations are performed:*

- *Initialize a random sparse matrix* $A \in \mathbb{R}^{n \times n}$ *with density* 0.01 *(*0.1 *in Python);*
- *Compute the exact density of non-zero elements of* A*;*
- *Compute* $L, U \in \mathbb{R}^{n \times n}$ *through the LU Factorization of* A*;*
- *Compute the exact density of non-zero elements of* L *and* U*;*
- *Update the script file, increasing the maximum value of* n *and looking for the value that run out of memory your PC.*

In the end, plot the evolution of the density values for A*,* L*, and* U*, with respect to* n *and comment the results. Moreover, spy the collocation of the non-zero values of the last matrices* A*,* L*, and* U *computed.*

Exercise 1.4. *Write a new script that performs the same operations of Exercise 1.3 but using only dense matrices. Observe when your PC runs out of memory.*

2 Iterative Methods for Linear Systems (Lab. 2)

As we stated at the end of the previous laboratory, direct methods are not particularly suitable for linear systems with very large sparse matrix, due to the *fill-in* phenomenon (see Section 1.7). On the contrary, *iterative methods* can preserve the advantages given by a sparse matrix

The main idea behind an iterative method is the following procedure:

- Choose a *starting guess* $\boldsymbol{x}^{(0)}$ for the solution of the problem;
- Build a sequence $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ such that:
  - $\boldsymbol{x}^{(k+1)}$ is computed using information of step k (or even previous steps);
  - $\boldsymbol{x}^{(k)} \xrightarrow{k\to\infty} \boldsymbol{x}^*$, where $\boldsymbol{x}^*$ is an exact solution of the problem.

Since the sequence of vectors $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ is infinite, the introduction of stopping criteria is necessary for returning an approximated solution $\widehat{\boldsymbol{x}} \approx \boldsymbol{x}^*$. Concerning the methods for linear systems there can be:

- **Relative Stopping Criteria:**
  - *Relative Increment:* $\| \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} \| \, / \, \| \boldsymbol{x}^{(k+1)} \| < \tau$;
  - *Relative Residual:* $\| \boldsymbol{r}^{(k+1)} \| \, / \, \| \boldsymbol{b} \| < \tau$, where $\boldsymbol{r}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(\boldsymbol{k})}$ is the *residual* for the "guess" $\boldsymbol{x}^{(k)}$, $k \in \mathbb{N}$;
  - ...
- **Absolute Stopping Criteria:**
  - *Absolute Increment:* $\| \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} \| < \tau$;
  - *Absolute Residual:* $\| \boldsymbol{r}^{(k+1)} \| < \tau$;
  - ...

Tolerance and Stopping Criteria: the value τ used above for describing the stopping criteria is known as *tolerance* and it is a positive real value ($\tau > 0$), usually very small (i.e., $\tau \approx 0$). Indeed, the smaller the relative/absolute quantities at step $k+1$, the more we can *assume* $\boldsymbol{x}^{(k+1)} \approx \boldsymbol{x}^*$.

In the end of this introduction, concerning the iterative methods for linear systems we can observe that:

- There is not a best stopping criteria. The best choice depends on the problem characteristics;
- The smaller tolerance τ, the larger the number of iterations required. On the other hand, the smaller the tolerance, the higher the accuracy in approximating $\boldsymbol{x}^*$;
- Ill-conditiong has effects also on iterative methods. Indeed, at step k, the inequality (5) can be translated into:

$$\frac{\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|}{\| \boldsymbol{x}^{(k)} \|} \leqslant \mathrm{k}(A) \frac{\| \boldsymbol{r}^{(k)} \|}{\| \boldsymbol{b} \|} \,. \tag{6}$$

  Then, even if we use a relative residual stopping criteria with τ very small ($\| \boldsymbol{r}^{(k)} \| \, / \, \| \boldsymbol{b} \|$), we only know that $\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \| \, / \, \| \boldsymbol{x}^{(k)} \| \leqslant \mathrm{k}(A)\tau$.

2.1 The Gradient Method

Let us consider a square linear system like (1), where A is *symmetric* and positive definite (see Section 1.1); necessarily, A is non-singualar. Let $J : \mathbb{R}^n \to \mathbb{R}$ be a quadratic function such that

$$J(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A \boldsymbol{x} - \boldsymbol{x}^T \boldsymbol{b} \,. \tag{7}$$

Therefore, J is a (quadratic) strictly convex function defined on $\mathbb{R}^n$ and it has a unique (global) minimum in $\boldsymbol{x}^*$. Now we recall the following theorems.

Theorem 2.1 (First Order Necessary Conditions [1])**.** *Let $\boldsymbol{x}^* \in \mathbb{R}^n$ and let $f : \mathbb{R}^n \to \mathbb{R}$ be such that it is continuously differentiable in an open neighborhood of $\boldsymbol{x}^*$. If $\boldsymbol{x}^*$ is a local minimizer of f, then $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$.*

Theorem 2.2 (Second Order Necessary Conditions [1])**.** *Let $\boldsymbol{x}^* \in \mathbb{R}^n$ and let $f : \mathbb{R}^n \to \mathbb{R}$ be such that its Hessian $H_f : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ exists in $\boldsymbol{x}^*$ and is continuous in an open neighborhood of $\boldsymbol{x}^*$. If $\boldsymbol{x}^*$ is a local minimizer of f, then $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$ and $H_f(\boldsymbol{x}^*)$ is positive semi-definite.*

Theorem 2.3 (Second Order Sufficient Conditions [1])**.** *Let $\boldsymbol{x}^* \in \mathbb{R}^n$ and let $f : \mathbb{R}^n \to \mathbb{R}$ be such that its Hessian H_f exists in $\boldsymbol{x}^*$ and is continuous in an open neighborhood of $\boldsymbol{x}^*$. If $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$ and $H_f(\boldsymbol{x}^*)$ is positive definite, then $\boldsymbol{x}^*$ is a* strict *local minimizer of f.*

Now, we observe that $\nabla J(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b}$ and $H_J(\boldsymbol{x}) = A$, for each $\boldsymbol{x} \in \mathbb{R}^n$. Therefore, we have that the global minimizer $\boldsymbol{x}^*$ of J is also the solution of the linear system $A\boldsymbol{x} = \boldsymbol{b}$, indeed:

- $\boldsymbol{x}^*$ minimizer of $J \Rightarrow \nabla J(\boldsymbol{x}^*) = A\boldsymbol{x}^* - \boldsymbol{b} = \mathbf{0}$ (Theorem 2.1);
- $\boldsymbol{x}^*$ solution of $A\boldsymbol{x} = \boldsymbol{b}$ and A symmetric positive definite $\Rightarrow$ $\boldsymbol{x}^*$ (strict) minimizer of J (Theorem 2.3);
- A is non-singular $\Rightarrow$ $A\boldsymbol{x} = \boldsymbol{b}$ has a unique solution;
- J is strictly convex $\Rightarrow$ has a unique minimizer.

Main Idea: for finding the solution of the linear system, we find instead the minimizer of J with an iterative method, instead of using a direct method on the linear system. For doing so, we recall the definition of descent direction (see Definition 2.1 below).

Definition 2.1 (Descent Direction)**.** *A vector $\boldsymbol{p} \in \mathbb{R}^n$ is a* descent direction *for a function $f : \mathbb{R}^n \to \mathbb{R}$ in $\boldsymbol{x} \in \mathbb{R}^n$ if exists $\varepsilon > 0$ such that*

$$f(\boldsymbol{x} + \alpha\boldsymbol{p}) < f(\boldsymbol{x}) \,, \tag{8}$$

for each $0 < \alpha \leqslant \varepsilon$. Moreover, let f be continuously differentiable in $\boldsymbol{x}$; then, $\boldsymbol{p}$ is a descent direction for f in $\boldsymbol{x}$ if and only if

$$\nabla f(\boldsymbol{x})^T \boldsymbol{p} < 0 \,. \tag{9}$$

See Figure 3 for an example in $\mathbb{R}^2$.

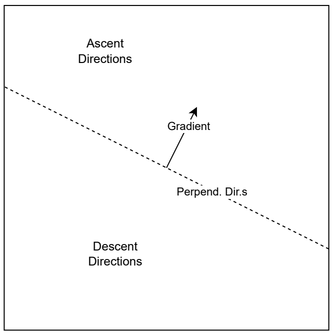

Figure 3: Example of regions of ascent, perpendicular, and descent directions in $\mathbb{R}^2$, with respect to a given gradient vector.

Remark 2.1 (Steepest Descent/Ascent Direction)**.** Given Definition 2.1, we observe that $\nabla f(\boldsymbol{x})$ and $-\nabla f(\boldsymbol{x})$ are the directions of steepest ascent and descent, respectively, for f in $\boldsymbol{x}$.

2.1.1 Sequence of the Gradient Method

Gathering all the knowledge described above, the iterative *gradient method* used for solving the linear system (i.e., minimizing J) consists in building the following sequence:

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha^{(k)} \nabla J(\boldsymbol{x}^{(k)}) \,, & \forall\ k \geqslant 0 \end{cases} , \tag{10}$$

where $\alpha^{(k)} \in \mathbb{R}_+$ is the best step length for reaching the minimum available value for $J(\boldsymbol{x}^{(k+1)})$.
In particular, denoting by $\boldsymbol{r}^{(k)}$ the k-th residual, i.e.

$$\boldsymbol{r}^{(k)} := \boldsymbol{b} - A\boldsymbol{x}^{(k)} \,, \forall\ k \geqslant 0 \,, \tag{11}$$

we have that $\boldsymbol{r}^{(k)} \equiv -\nabla J(\boldsymbol{x}^{(k)})$ and, therefore, the sequence can be rewritten as

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)} \boldsymbol{r}^{(k)} \,, & \forall\ k \geqslant 0 \end{cases} , \tag{12}$$

where $\alpha^{(k)} = (\boldsymbol{r}^{(k)\,T}\boldsymbol{r}^{(k)})/(\boldsymbol{r}^{(k)\,T}A\boldsymbol{r}^{(k)})$, because $\frac{\mathrm{d}}{\mathrm{d}\alpha}J(\boldsymbol{x} + \alpha\boldsymbol{r}) = \cdots = -\boldsymbol{r}^T\boldsymbol{r} + \alpha\boldsymbol{r}^T A\boldsymbol{r}$.
In the following, we report two pseudo-codes for the implementation of the gradient method.

Algorithm 2.1 (Gradient Method (naive))**.**

```
1: $\boldsymbol{x}^{(0)}$ given
2: $k \leftarrow 0$
3: while stopping criteria are not satisfied do
4:     $\boldsymbol{r}^{(k)} \leftarrow \boldsymbol{b} - A\boldsymbol{x}^{(k)}$
5:     $\alpha^{(k)} \leftarrow (\boldsymbol{r}^{(k)\,T}\boldsymbol{r}^{(k)})/(\boldsymbol{r}^{(k)\,T}A\boldsymbol{r}^{(k)})$
6:     $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{r}^{(k)}$
7:     $k \leftarrow k + 1$
8: end while
9: $\widehat{\boldsymbol{x}} \leftarrow \boldsymbol{x}^{(k)}$
10: return : $\widehat{\boldsymbol{x}}$
```

Algorithm 2.2 (Gradient Method)**.**

```
1: $\boldsymbol{x}^{(0)}$ given
2: $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{b} - A\boldsymbol{x}^{(0)}$
3: $k \leftarrow 0$
4: while stopping criteria are not satisfied do
5:     $\boldsymbol{z}^{(k)} \leftarrow A\boldsymbol{r}^{(k)}$
6:     $\alpha^{(k)} \leftarrow (\boldsymbol{r}^{(k)\,T}\boldsymbol{r}^{(k)})/(\boldsymbol{r}^{(k)\,T}\boldsymbol{z}^{(k)})$
7:     $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{r}^{(k)}$
8:     $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{r}^{(k)} - \alpha^{(k)}\boldsymbol{z}^{(k)}$        ▷ because $\boldsymbol{r}^{(k+1)} = \boldsymbol{b} - A(\boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{r}^{(k)})$
9:     $k \leftarrow k + 1$
10: end while
11: $\widehat{\boldsymbol{x}} \leftarrow \boldsymbol{x}^{(k)}$
12: return : $\widehat{\boldsymbol{x}}$
```

2.1.2 Convergence Properties

The convergence of the gradient method is characterized by the following property:

$$\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A \leqslant 2 \left(\frac{\mathrm{k}_2(A) - 1}{\mathrm{k}_2(A) + 1} \right)^k \| \boldsymbol{x}^{(0)} - \boldsymbol{x}^* \|_A \,, \quad \forall\ k \geqslant 0 \,, \tag{13}$$

where $\| \cdot \|_A$ is the *energy norm* such that $\| \boldsymbol{x} \|_A := \sqrt{\boldsymbol{x}^T A\boldsymbol{x}}$, and $\mathrm{k}_2(A) := \| A \|_2 \| A^{-1} \|_2 = \lambda_{\max}/\lambda_{\min}$ (λ denotes eigenvalues of A).

Observation: the more ill-conditioned A, the larger is the upper bound of $\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A$ and, therefore, the slower can be the convergence.

2.2 Conjugate Gradient Method

The idea behind the *conjugate gradient* (CG) method is to modify the gradient method using a different descent direction, other than $\boldsymbol{r} = -\nabla J(\boldsymbol{x})$, for improving the convergence.

Without going into details, the CG method uses as descent directions the sequence

$$\begin{cases} \boldsymbol{p}^{(0)} = \boldsymbol{r}^{(0)} \\ \boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)}\,, \quad \forall\ k \geqslant 0 \end{cases}, \tag{14}$$

where $\beta^{(k+1)} = -(\boldsymbol{p}^{(k)\,T}A\boldsymbol{r}^{(k+1)})/(\boldsymbol{p}^{(k)\,T}A\boldsymbol{p}^{(k)})$. Then, the sequence build by the CG for finding $\boldsymbol{x}^*$ is

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n\,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}\,, & \forall\ k \geqslant 0 \end{cases}, \tag{15}$$

where $\alpha^{(k)} = (\boldsymbol{r}^{(k)\,T}\boldsymbol{p}^{(k)})/(\boldsymbol{p}^{(k)\,T}A\boldsymbol{p}^{(k)})$.

The advantages and disadvantages of CG method are in the following list:

+ *Finite termination:* the CG method reaches the solution $\boldsymbol{x}^* \in \mathbb{R}^n$ in at most n steps;
- if $n \gg 1$, the advantages of the finite termination can be lost;
- Due to arithmetic errors, exact conjugacy can be lost;

± Even losing the exact conjugacy, we can implement CG as an iterative method with stopping criteria.

In the following, we report two pseudo-codes for the implementation of the CG method.

Algorithm 2.3 (Conjugate Gradient Method (naive))**.**

```
1: $\boldsymbol{x}^{(0)}$ given
2: $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{b} - A\boldsymbol{x}^{(0)}$
3: $\boldsymbol{p}^{(0)} \leftarrow \boldsymbol{r}^{(0)}$
4: $k \leftarrow 0$
5: while stopping criteria are not satisfied do
6:     $\alpha^{(k)} \leftarrow (\boldsymbol{r}^{(k)\,T}\boldsymbol{p}^{(k)})/(\boldsymbol{p}^{(k)\,T}A\boldsymbol{p}^{(k)})$
7:     $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}$
8:     $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{b} - A\boldsymbol{x}^{(k+1)}$
9:     $\beta^{(k+1)} \leftarrow -(\boldsymbol{p}^{(k)\,T}A\boldsymbol{r}^{(k+1)})/(\boldsymbol{p}^{(k)\,T}A\boldsymbol{p}^{(k)})$
10:    $\boldsymbol{p}^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)}$
11:    $k \leftarrow k + 1$
12: end while
13: $\widehat{\boldsymbol{x}} \leftarrow \boldsymbol{x}^{(k)}$
14: return : $\widehat{\boldsymbol{x}}$
```

Algorithm 2.4 (Conjugate Gradient Method)**.**

```
1: $\boldsymbol{x}^{(0)}$ given
2: $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{b} - A\boldsymbol{x}^{(0)}$
3: $\boldsymbol{p}^{(0)} \leftarrow \boldsymbol{r}^{(0)}$
4: $k \leftarrow 0$
5: while stopping criteria are not satisfied do
6:     $\boldsymbol{z}^{(k)} \leftarrow A\boldsymbol{p}^{(k)}$
```

```
7:      $\alpha^{(k)} \leftarrow (\boldsymbol{r}^{(k)\,T}\boldsymbol{p}^{(k)})/(\boldsymbol{p}^{(k)\,T}\boldsymbol{z}^{(k)})$
8:      $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}$
9:      $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{r}^{(k)} - \alpha^{(k)}\boldsymbol{z}^{(k)}$
10:     $\beta^{(k+1)} \leftarrow -(\boldsymbol{r}^{(k+1)\,T}\boldsymbol{z}^{(k)})/(\boldsymbol{p}^{(k)\,T}\boldsymbol{z}^{(k)})$
11:     $\boldsymbol{p}^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)}$
12:     $k \leftarrow k + 1$
13: end while
14: $\widehat{\boldsymbol{x}} \leftarrow \boldsymbol{x}^{(k)}$
15: return : $\widehat{\boldsymbol{x}}$
```

2.2.1 Preconditioning

The convergence of the CG is characterized by the following property:

$$\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A \leqslant 2 \left(\frac{\sqrt{\mathrm{k}_2(A)} - 1}{\sqrt{\mathrm{k}_2(A)} + 1} \right)^k \| \boldsymbol{x}^{(0)} - \boldsymbol{x}^* \|_A \,, \quad \forall\, k \geqslant 0 \,, \tag{16}$$

where $\| \cdot \|_A$ is the *energy norm* such that $\| \boldsymbol{x} \|_A := \sqrt{\boldsymbol{x}^T A \boldsymbol{x}}$, and $\mathrm{k}_2(A) :=\| A \|_2 \| A^{-1} \|_2 = \lambda_{\max}/\lambda_{\min}$ (λ denotes eigenvalues of A).

Observation: the more ill-conditioned A, the larger is the upper bound of $\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A$ and, therefore, the slower can be the convergence. Nonetheless, the upper bound in (16) is lower thant the one in (13).

Given the above observation, we observe that a method to reduce the effects of an ill-conditioned matrix exists. The main idea is to introduce a variable substitution

$$\boldsymbol{x} = C^{-1}\boldsymbol{y} \,. \tag{17}$$

Using (17), we have that (7) changes into $J(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T C^{-T} A C^{-1}\boldsymbol{y} - \boldsymbol{y}^T C^{-T}\boldsymbol{b}$, and we look for the solution of the linear system $C^{-T}AC^{-1}\boldsymbol{y} = C^{-T}\boldsymbol{b}$. Therefore, with a proper choice of C we can reduce the condition number of the new matrix and speed up the convergence of the CG method.

Typically, the "best" *preconditioner* (i.e., the matrix C) does not exist. In general, it depends on the nature of the matrix A. Nonetheless, a canonical choice can be the matrix $\widetilde{L}^T$ returned by the *incomplete Cholesky* method.

The incomplete Cholesky method perform an incomplete Cholesky factorization such that it returns a *sparse* lower triangular matrix $\widetilde{L}$ such that $\widetilde{L}^T \approx R$, where R is the upper triangular matrix of the Cholesky factorization (see Section 1.1). The choice of $C = \widetilde{L}^T$ is motivated by the fact that the more $\widetilde{L}^T \approx R$, the more $C^{-T}AC^{-1} \approx \mathbb{I}_n$ (and, therefore, the problem is stable). Of course, the more $\widetilde{L}^T \approx R$, the more the density of $\widetilde{L}$, losing the advantages of sparsity.

2.3 Other Iterative Methods

We point the attention of the reader on the characteristics of the matrix A we used in this section. We have always assumed A *symmetric* and *positive definite* (at most, we can relax the hypotheses to *positive semi-definiteness*). Then, if A does not satisfy these properties, the gradient and CG methods cannot be applied.

Other iterative methods, can be for example the *generalized minimal residual method* (GMRES), where the preconditioning can be performed exploiting an incomplete LU factorization.

2.4 Exercises

From now on, we will write the text of the exercises for Matlab users. Nonetheless, you can use Python instead.

Exercise 2.1 (Gradient Method)**.** *Write a function called `lab02_gradient_linsys` that implement in Matlab Algorithm 2.2 using a relative residual as stopping criterium.*

Test your function on the linear system defined by the variables inside the file `lab02_sparse_linsys.mat`.

Exercise 2.2 (Conjugate Gradient Method)**.** *Write a function called `lab02_cg_linsys` that implement in Matlab Algorithm 2.4 using a relative residual as stopping criterium.*

Test your function on the linear system defined by the variables inside the file `lab02_sparse_linsys.mat`.

Exercise 2.3. *Write a script where:*

1. *Generate two sparse tridiagonal matrices (see `spdiags` function) $A_1, A_2 \in \mathbb{R}^{n\times n}$ (say $n = 1000$) such that*

$$A_i = \begin{bmatrix} \alpha_i & -1 & & & \\ -1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & \alpha_i \end{bmatrix}, \quad \alpha_1 = 4\,, \;\; \alpha_2 = 2\,.$$

2. *Compute and visualize $\mathrm{k}_2(A_1), \mathrm{k}_2(A_2)$;*

3. *For each $\tau = 10^{-4}, \dots, 10^{-10}$, solve the linear systems $A_i\boldsymbol{x} = \boldsymbol{b}_i$, with exact solution $\boldsymbol{x}_i^* = (1, \dots, 1) \in \mathbb{R}^n$, $i = 1, 2$ (i.e., build $\boldsymbol{b}_i$ as in Exercise 1.1, item 3).*

   In particular, solve the linear systems with `lab02_gradient_linsys` of Exercise 2.1, with `lab02_cg_linsys` of Exercise 2.2, and with the `pcg` matlab function, both with and without preconditionig (see `ichol` function).

4. *Plot the number of iterations used by the methods for solving the linear systems, varying the tolerance values.*

3 Steepest Descent with Backtracking (Lab. 3)

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be given. The steepest descent method is an iterative optimization method that, starting from a given vector $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$, computes a sequence of vectors $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ characterized by

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha \boldsymbol{p}^{(k)}\,, \quad \forall\, k \geqslant 0\,, \tag{18}$$

where the descent direction $\boldsymbol{p}^{(k)}$ is the steepest one, i.e. $\boldsymbol{p}^{(k)} = -\nabla f(\boldsymbol{x}^{(k)})$, and the step length factor $\alpha \in \mathbb{R}^+$ is arbitrarily chosen.

3.1 Backtracking

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be given. The backtracking strategy for an iterative optimization method consists of looking for a value $\alpha^{(k)}$ satisfying the Armijo condition at each step k of the method, i.e.

$$f(\underbrace{\boldsymbol{x}^{(k+1)}}_{\boldsymbol{x}^{(k)}+\alpha^{(k)}\boldsymbol{p}^{(k)}}) \leqslant f(\boldsymbol{x}^{(k)}) + c_1 \alpha^{(k)} \nabla f(\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(k)}\,, \tag{19}$$

where $c_1 \in (0,1)$ (typically, the standard choice is $c_1 = 10^{-4}$).

We recall that the Armijo condition suggests that a "good" $\alpha^{(k)}$ is such that you have a sufficient decrease in f and, moreover, the function value at the new point $f(\boldsymbol{x}^{(k+1)}) = f(\boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)})$ is under the "reduced tangent hyperplane" of f at $\boldsymbol{x}^{(k)}$.

To better explain the Armijo condition, we look at the function $\phi(\alpha) := f(\boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)})$, such that $\phi'(\alpha) = \nabla f(\boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)})^T \boldsymbol{p}^{(k)}$ and $\phi'(0) = \nabla f(\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(k)}$ (see Figure 4).

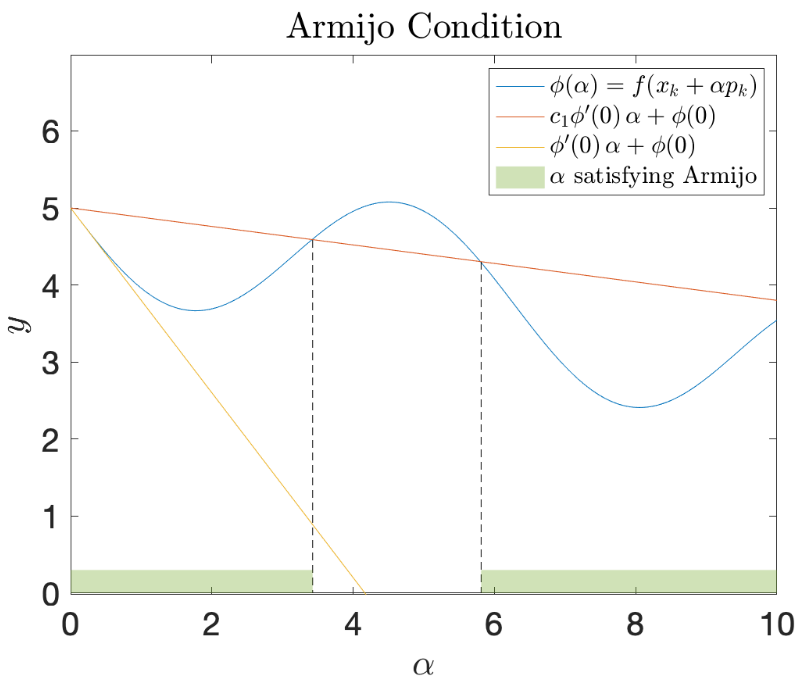

Figure 4: Example of the Armijo condition.

The backtracking strategy is an *iterative process* that looks for this value $\alpha^{(k)}$. Given an arbitrary factor $\rho \in (0,1)$ and an arbitrary starting value $\alpha_0^{(k)}$ for $\alpha^{(k)}$, we decrease iteratively $\alpha_0^{(k)}$, multiplying it by ρ, until the Armijo condition is satisfied. Then $\alpha^{(k)} = \rho^{t_k}\alpha_0^{(k)}$, for a $t_k \in \mathbb{N}$, if it satisfies Armijo but $\rho^{t_k-1}\alpha_0^{(k)}$ does not.

Remark 3.1 (Few things to keep in mind)**.**

1. The Armijo condition is always satisfied for extremely small values of α. Then, it is not enough to ensure that the algorithm makes reasonable progress; indeed, if α is too small, unacceptably short steps are taken (see item 3 below).

2. For simplicity, we consider ρ as a fixed parameter, but it can be chosen using already available information, changing with the iterations;

3. Other conditions could be imposed to guarantee that not too-short steps are taken (e.g., Wolfe conditions[1]), but they are not practical to be implemented. Practical implementations, instead of imposing a second condition, frequently use the backtracking strategy; for example, a proper choice of $\alpha_0^{(k)}$, ρ, and the maximum number T of backtracking steps can guarantee that $\rho^T \alpha_0^{(k)} \geqslant \epsilon$, where ϵ is the minimum step-length acceptable.

4. The choice of $\alpha_0^{(k)}$ is problem-dependent and/or method-dependent.

3.2 Exercises

The text of the exercises is for Matlab users. Nonetheless, you can use Python instead.

Exercise 3.1 (Steepest Descent)**.** *Write a Matlab function* `steepest_descent.m` *that implements the* steepest descent *optimization method, given:*

- *`x0`: a* column vector *of n elements representing the starting point for the optimization method;*
- *`f`: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value $f(\boldsymbol{x})$, where $f : \mathbb{R}^n \to \mathbb{R}$ is the* loss function *that have to be minimized;*
- *`gradf`: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value $\nabla f(\boldsymbol{x})$ as a* column *vector, where $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is the* gradient *of f;*
- *`alpha`: a* real scalar value *characterizing the step length of the optimization method (fixed, for simplicity);*
- *`kmax`: an* integer scalar value *characterizing the maximum number of iterations of the method;*
- *`tolgrad`: a* real scalar value *characterizing the tolerance with respect to the norm of the gradient in order to stop the method.*

The outputs of the function must be:

- *`xk`: the last vector $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ computed by the optimization method before it stops;*
- *`fk`: the value $f(\boldsymbol{x}^{(k)})$;*
- *`gradfk_norm`: the euclidean norm of $\nabla f(\boldsymbol{x}^{(k)})$;*
- *`k`: index value of the last step executed by the optimization method before stopping;*
- *`xseq`: a matrix/vector in $\mathbb{R}^{n \times k}$ such that each column j is the j-th vector $\boldsymbol{x}^{(j)} \in \mathbb{R}^n$ generated by the iterations of the method.*

Once you have written the function, test it using the data `x0`, `f`, `gradf`, `alpha`, `kmax`, `tolgrad`, *inside the file* `test_paraboloids.mat`

[1]i.e., Armijo condition and curvature condition $\nabla f(\boldsymbol{x}_{k+1})^T \boldsymbol{p}_k \geqslant c_2 \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k$, $c_2 \in (c_1, 1)$.

Exercise 3.2 (Steepest Descent with Backtracking)**.** *Write a Matlab function* `steepest_desc-_bcktrck.m` *that implements the* steepest descent *optimization method with the* backtracking strategy, *given:*

- `x0`*:* *a* column vector *of n elements representing the starting point for the optimization method;*
- `f`*:* *a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value $f(\boldsymbol{x})$, where $f : \mathbb{R}^n \to \mathbb{R}$ is the* loss function *that have to be minimized;*
- `gradf`*:* *a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value $\nabla f(\boldsymbol{x})$ as a* column *vector, where $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is the* gradient *of f;*
- `alpha0`*:* *a* real scalar value *characterizing the starting value for the backtracking (fixed, for simplicity);*
- `kmax`*:* *an* integer scalar value *characterizing the maximum number of iterations of the method;*
- `tolgrad`*:* *a* real scalar value *characterizing the tolerance with respect to the norm of the gradient in order to stop the method.*
- `c1`*: the factor c_1 for the Armijo condition that must be a scalar in $(0, 1)$;*
- `rho`*: factor less than 1, used to reduce α (fixed, for simplicity);*
- `btmax`*: maximum number of steps allowed to update α during the backtracking strategy.*

The outputs of the function must be:

- `xk`*: the last vector $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ computed by the optimization method before it stops;*
- `fk`*: the value $f(\boldsymbol{x}^{(k)})$;*
- `gradfk_norm`*: the euclidean norm of $\nabla f(\boldsymbol{x}^{(k)})$;*
- `k`*: index value of the last step executed by the optimization method before stopping;*
- `xseq`*: a matrix/vector in $\mathbb{R}^{n \times k}$ such that each column j is the j-th vector $\boldsymbol{x}^{(j)} \in \mathbb{R}^n$ generated by the iterations of the method.*
- `btseq`*: row vector in $\mathbb{R}^k$ such that the j-th element is the number of backtracking iterations done at the j-th step of the steepest descent.*

Extra options:

- *implement the procedure such that it stops if the backtracking strategy cannot find $\alpha^{(k)}$ satisfying the Armijo condition;*
- `flag`*: an extra output variable for the function containing a string that is a message. The message explains the result (e.g.* "Procedure stopped in k steps, with gradient norm ε" *and/or* "Procedure stopped because..."*).*

Once you have written the function, test it using the data `x0`, `f`, `gradf`, `alpha0`, `kmax`, `tolgrad`, `c1`, `rho`, `btmax`, *inside the file* `test_paraboloids.mat`
Suggestion: *copy the code of the function of Exercise 3.1 and modify it adding the backtracking.*

Exercise 3.3 (Steepest Descent Path Visualization)**.** *Given the functions of the previous exercises, write a Matlab script where you test these functions using the data inside the file* `test_paraboloids.mat` *(see previous exercises). Then, plot:*

- *A top view of the loss f (given in* test_paraboloids.mat*) using the Matlab function* `contour`*, together with the sequence* `xseq` *in* $\mathbb{R}^2$*;*

- *the barplot of the values in* `btseq` *using the function* `bar`*;*

- *the surface of the loss f using the Matlab function* `surf`*, together with the function values of the sequence* `xseq` *in* $\mathbb{R}^3$*. See the* `meshgrid` *function for preparing the plot domain. You can use the function* `f_meshgrid` *for evaluating the function on the output matrices of* `meshgrid`*. For any doubts, see* https://www.mathworks.com/help/matlab/ref/surf.html.

Exercise 3.4 (n-Dimensional Steepest Descent)**.** *Given the functions of the previous exercises, write a Matlab script where you test these functions on an n-dimensional paraboloid defined by the function handle* `paraboloid` *inside the file* `test_paraboloids.mat` *(gradient defined by the function handle* `gradparaboloid`*). Run the test for different values of n, e.g.: $n = 10, 100, 1000$.*

A Material for Refreshing Basic Knowledge

MATLAB pills: If you want to use MATLAB and are not so familiar with it, you can find it useful to follow these video pills:

1. Introduction: `https://youtu.be/J2-R_Hw4Ak8`
2. Variables and Scripts: `https://youtu.be/oI4ZP-GXEuU`
3. Functions: `https://youtu.be/C_59NdVeD0Q`
4. Arrays and Matrices (part 1): `https://youtu.be/l7TClaukIkE`
5. Indexing - Accessing Elements in Arrays and Matrices: `https://youtu.be/OB1I3FJn9Sg`
6. Operations with Arrays and Matrices: `https://youtu.be/J-gzenf-4bY`
7. Arrays and Matrices (part 2): `https://youtu.be/e9CdE95va-g`
8. Logical Indexing: `https://youtu.be/_vjDMKF9zcQ`

Background (Numerical Methods for Linear Algebra): if you need to consolidate your background in basic numerical methods for linear algebra, you may find it helpful to watch the following video lectures: from Portale della Didattica (general part, not the course page!) follow `Materiale -> Lezioni on-line -> Primo Anno -> Linear Algebra and Geometry`.

The suggested lectures are those delivered by Prof. Dabbene, and should be enough to watch lectures 5, 9, 13 (neglect the part on polynomials), 22, 29, and 33. Italian students may prefer to watch the lectures in Italian, then just look for Algebra Lineare e Geometria and look at lectures from Prof.ssa Scuderi (lectures 4, 8, 12, 24 [ignore the initial part on approximation], 28, 30).

References

[1] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Second. 9781447122234. Springer, 2012. ISBN: 9780387303031. DOI: 10.1007/978-1-4471-2224-1_2.