Numerical Optimization for Large Scale Problems
Lecture Notes

Francesco Della Santa *

Politecnico di Torino, A.Y. 2025/2026

Abstract

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

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*, A.Y. 2025/2026. Each section corresponds to one laboratory and it will consist of a brief theoretical recap of the arguments, followed by the exercises.

Usually (but not always), the solutions of the exercises of each laboratory are published on the course's web page, 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 Section 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

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 by 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 $\boldsymbol{x}^* = (b_1/a_{11}, \dots, b_n/a_n n)$.
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, there is 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., $A\boldsymbol{x} = \boldsymbol{b}_1, \ldots, A\boldsymbol{x} = \boldsymbol{b}_N$).

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.

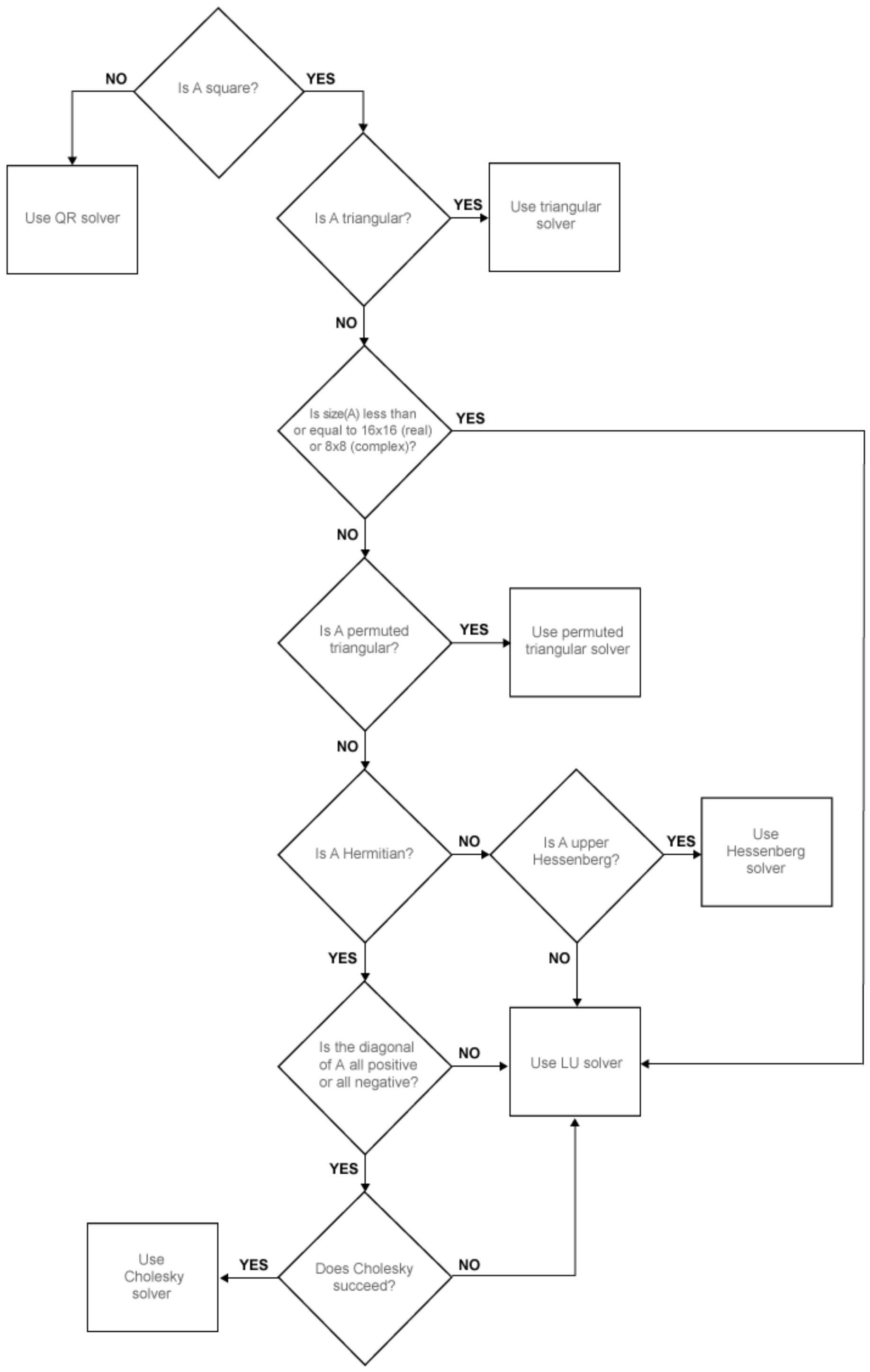

Figure 1: Scheme for the `mldivide` (\) function in Matlab (dense matrices case).

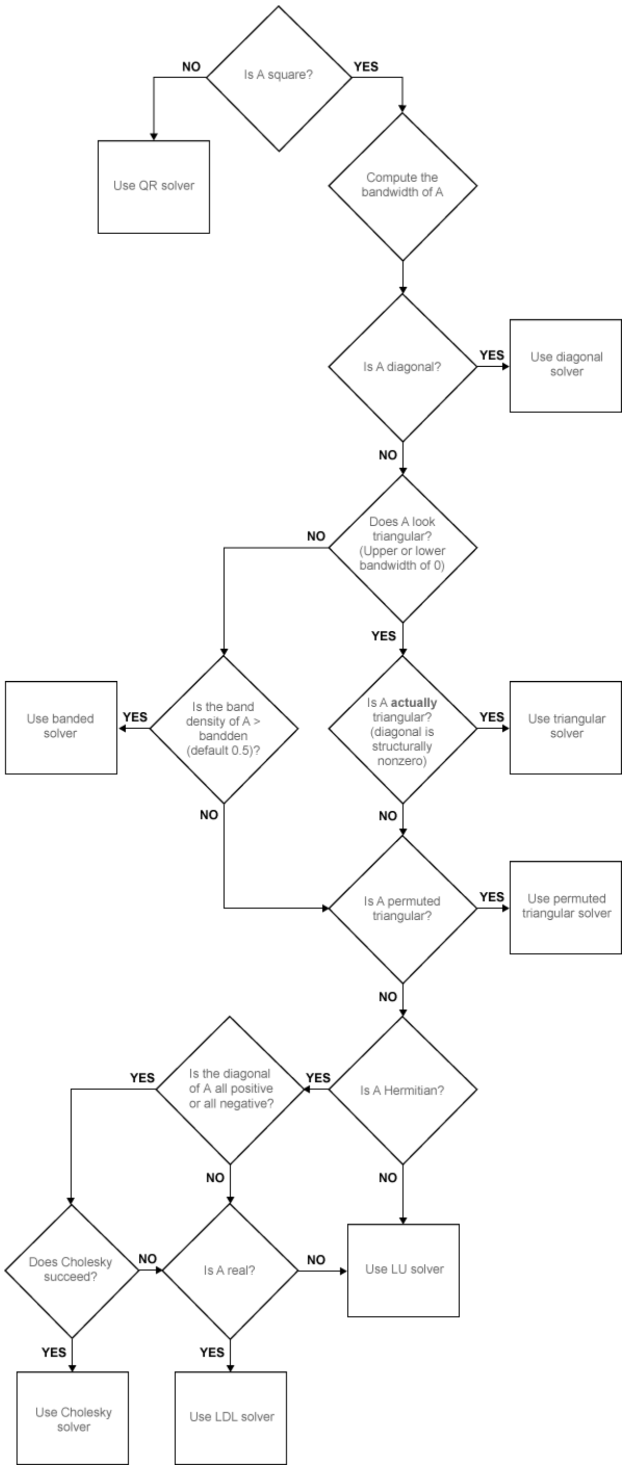
Is A square?
NO
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
Use diagonal solver
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO
YES
Is the band density of A > bandden (default 0.5)?
YES
Use banded solver
NO
Is A **actually** triangular? (diagonal is structurally nonzero)
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
Is A real?
NO
YES
YES
Use Cholesky solver
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 1.*
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 (Problem Conditioning)**.** *Write a function that takes as inputs a random seed, the dimension n and the number of samples N, and that runs the same operations of the script of Exercise 1.1, returning:*

1. *$A, \boldsymbol{b}$, and D;*
2. *the solutions of $A\boldsymbol{x} = \boldsymbol{b}$, $D\boldsymbol{x} = \boldsymbol{b}$, and $A\boldsymbol{x} = \boldsymbol{b}_i$, for each $i = 1, \ldots, N$;*
3. *$k(A)$ and $k(D)$.*

Try the function for different random seeds (e.g., 42) and try to understand if the linear systems' solutions are reliable or not. Then, try also to change dimension n and number of samples N.

Exercise 1.4. *Download, run, and read the script `lab01_sparse_examples.m/.py` carefully. Then write a new script file where, for each $n = 200, 400, \ldots, 2000$, 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.

Suggestion: *write a function that takes a random seed, the dimension n, and the desired density of A as inputs, and that returns all you need to perform the operations asked by the exercise.*

Exercise 1.5. *Write a new script that performs the same operations of Exercise 1.4 but using only dense matrices. Observe when your PC runs out of memory.*

Exercise 1.6. *Write a new script that performs the same operations of Exercise 1.4 but using the incomplete LU factorization of the matrix $\widetilde{A} = A + \mathbb{I}$. Measure and spy the density of the factorization matrices $\widetilde{L}, \widetilde{U}$, and the compute error norm $\| \widetilde{L}\widetilde{U} - \widetilde{A}P \|$ and the relative error norm $\| \widetilde{L}\widetilde{U} - \widetilde{A}P \| / \| A \|$.*

2 Finite Differences

We briefly recall the main Finite Difference formulas for approximating derivatives, to help carry out the exercises of this section.

Notation 2.1 (Notations for Gradient, Hessian, and Jacobian)**.** *Let $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{F} = (f_1, \dots, f_m) : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m$, then:*

- Gradient of f (differentiable in Ω):

$$\nabla f = \begin{bmatrix} f_{x_1} \\ \vdots \\ f_{x_n} \end{bmatrix} : \mathbb{R}^n \to \mathbb{R}^n \,, \tag{6}$$

  where $f_{x_i} := \partial f / \partial x_i$, for each $i = 1, \dots, n$.

- Hessian of f (twice differentiable in Ω):

$$Hf = \nabla^2 f = \left(\frac{\partial^2}{\partial x_j \partial x_i} f \right)_{ij} = \begin{bmatrix} f_{x_1 x_1} & \cdots & f_{x_1 x_n} \\ \vdots & \ddots & \vdots \\ f_{x_n x_1} & \cdots & f_{x_n x_n} \end{bmatrix} : \mathbb{R}^n \to \mathbb{R}^{n \times n} \,, \tag{7}$$

  where $f_{x_i x_j} := \partial^2 f / (\partial x_j \partial x_i)$, for each $i, j = 1, \dots, n$.

  ***Attention:** if $f \in C^2(\Omega)$, $Hf(\boldsymbol{x})$ is* symmetric *for each $\boldsymbol{x} \in \Omega$. Otherwise, symmetry is not guaranteed!*

- Jacobian of $\boldsymbol{F}$ ($f_1, \dots, f_m$ differentiable in Ω):

$$J\boldsymbol{F} = \left(\frac{\partial}{\partial x_j} f_i \right)_{ij} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1 & \cdots & \frac{\partial}{\partial x_n} f_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m & \cdots & \frac{\partial}{\partial x_n} f_m \end{bmatrix} = \begin{bmatrix} \nabla f_i^T \\ \vdots \\ \nabla f_m^T \end{bmatrix} : \mathbb{R}^n \to \mathbb{R}^{m \times n} \,. \tag{8}$$

2.1 Finite Differences (Gradient)

If $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ is $C^1(\Omega)$, we can approximate the gradient of f in $\widehat{\boldsymbol{x}} \in \Omega$ with the following expressions.

- *Forward Finite Differences (FW-FD):*

$$f_{x_i}(\widehat{\boldsymbol{x}}) = \frac{\partial}{\partial x_i} f(\widehat{\boldsymbol{x}}) \approx \frac{f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) - f(\widehat{\boldsymbol{x}})}{h} \,, \quad \forall\, i = 1, \dots, n \,; \tag{9}$$

- *Centered Finite Differences (C-FD):*

$$f_{x_i}(\widehat{\boldsymbol{x}}) = \frac{\partial}{\partial x_i} f(\widehat{\boldsymbol{x}}) \approx \frac{f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) - f(\widehat{\boldsymbol{x}} - h\boldsymbol{e}_i)}{2h} \,, \quad \forall\, i = 1, \dots, n \,. \tag{10}$$

2.1.1 Choice of h

Large values for h return obviously poor approximations, but too small values are characterized by numerical cancellation problems.

In FW-FD, a suggested trade-off is to select $h = \sqrt{\varepsilon_m}$ (see [2] for motivations), or $h_i = \sqrt{\varepsilon_m} |\widehat{x}_i|$ for approximating the partial derivative f_{x_i}. For simplicity, this value can be adopted also for C-FD, even if better values of h exist.

2.2 Finite Differences (Hessian)

If the function $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ is $C^2(\Omega)$ (i.e., symmetric Hessian for each $\boldsymbol{x} \in \Omega$), we can approximate the second order derivatives of f in $\widehat{\boldsymbol{x}} \in \Omega$ with the following expressions.

- *Hessian's diagonal elements:*

$$(Hf)_{ii}(\widehat{\boldsymbol{x}}) = f_{x_i x_i}(\widehat{\boldsymbol{x}}) \approx \frac{f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) - 2f(\widehat{\boldsymbol{x}}) + f(\widehat{\boldsymbol{x}} - h\boldsymbol{e}_i)}{h^2}\,, \tag{11}$$

  for each $i = 1 \ldots, n$;

- *Hessian's non-diagonal elements:*

$$(Hf)_{ij}(\widehat{\boldsymbol{x}}) = f_{x_i x_j}(\widehat{\boldsymbol{x}}) \approx \frac{f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i + h\boldsymbol{e}_j) - f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) - f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_j) + f(\widehat{\boldsymbol{x}})}{h^2}\,, \tag{12}$$

  for each $i, j = 1, \ldots, n$, $i \neq j$.

2.2.1 Choice of h

The choice of h for the Hessian, in order to avoid numerical cancellation problems, is suggested to be greater than the h used for the gradient, due to the h^2 in the denominator. Then, for example, we can use an h that is the square root of the value that we would use for the gradient: $h = \sqrt{h_{grad}}$.

2.3 Finite Differences (Jacobian)

If $\boldsymbol{F} = (f_1, \ldots, f_m) : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is such that $f_1, \ldots, f_m \in C^1(\Omega)$, we can approximate the Jacobian of $\boldsymbol{F}$ in $\widehat{\boldsymbol{x}} \in \Omega$ with the following expressions.

- *Forward Finite Differences (j-th Jacobian's column):*

$$(J\boldsymbol{F})_{\cdot j}(\widehat{\boldsymbol{x}}) \approx \frac{\boldsymbol{F}(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_j) - \boldsymbol{F}(\widehat{\boldsymbol{x}})}{h}\,, \quad \forall\, j = 1, \ldots, n\,; \tag{13}$$

- *Centered Finite Differences (j-th Jacobian's column):*

$$(J\boldsymbol{F})_{\cdot j}(\widehat{\boldsymbol{x}}) \approx \frac{\boldsymbol{F}(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_j) - \boldsymbol{F}(\widehat{\boldsymbol{x}} - h\boldsymbol{e}_j)}{2h}\,, \quad \forall\, j = 1, \ldots, n\,; \tag{14}$$

Attention: remember that is possible to approximate more columns at the same times if $J\boldsymbol{F}$ is sparse (see [2]).

2.4 Exercises

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

Exercise 2.1 (Finite Differences - Gradient). *Write a Matlab function* findiff_grad.m *that implements both the* forward *and* centered *finite difference methods for the gradient computation, given:*

- **f**: *a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns $f(\boldsymbol{x})$, where $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ is the* function *whose gradient we want to calculate;*
- **x**: *a* column vector *of n elements representing the point where we want to compute the gradient of f;*
- **h**: *the finite difference* "step"*;*

- `type`*: a string of characters such that if it is* `'fw'`*, the function uses the forward method (default), and if it is* `'c'`*, the function uses the centered method.*

The output (i.e., the gradient approximation) must be stored as a column vector.

Once you have written the function, test it using the functions in Section B.1 and compare the results with the exact gradients.

Exercise 2.2 (Finite Differences - Jacobian)**.** *Write a Matlab function* findiff_J.m *that implements both the* forward *and* centered *finite difference methods for the Jacobian computation, given:*

- *F: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns $\boldsymbol{F}(\boldsymbol{x})$, where $\boldsymbol{F} : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is the* function *whose Jacobian we want to calculate;*
- *x: a* column vector *of n elements representing the point where we want to compute the Jacobian of $\boldsymbol{F}$;*
- *h: the finite difference "*step*";*
- `type`*: a string of characters such that if it is* `'fw'`*, the function uses the forward method (default), and if it is* `'c'`*, the function uses the centered method.*

Once you have written the function, test it using the functions in Section B.1 for building a vectorial test function $\boldsymbol{F}$ and compare the results with the exact gradients.

Exercise 2.3 (Finite Differences - Hessian)**.** *Write a Matlab function* findiff_Hess.m *that implements the finite difference method for the Hessian computation (symmetric case only), given:*

- *f: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns $f(\boldsymbol{x})$, where $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ is the C^2* function *whose Hessian we want to calculate;*
- *x: a* column vector *of n elements representing the point where we want to compute the gradient of f;*
- *h: the finite difference "*step*".*

Once you have written the function, test it using the functions in Section B.1 and compare the results with the exact Hessians.

Exercise 2.4 (Finite Differences - Hessian as Jacobian of the Gradient)**.** *Modify the Matlab function* findiff_J.m *such that it can force the symmetry in the output matrix.*

Once you have written the function, test it using the functions in Section B.1 for computing the Hessians, given the exact gradients; then, compare the results with the exact Hessians.

Exercise 2.5 (Finite Differences - Approximation Quality and h)**.** *Write a Matlab script* `lab-02_FD_tests.m` *where you compute the gradient, Jacobian, and Hessian of some test functions, varying the value of h. Analyze and comment on the results.*

Exercise 2.6 (Finite Differences - Smart Implementations)**.** *Look at the test function illustrated in B.2. Then, write a Matlab custom functions for approximating efficiently with Finite Differences the gradient and the Hessian (both directly or as Jacobian of the exact gradient).*

Suggestion: *exploit the summation structure for the gradient and the sparsity for the Hessian.*

3 Iterative Methods for Linear Systems

As we stated at the end of the Section 1, 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 criterion. 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:

  $$\frac{\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|}{\| \boldsymbol{x}^{(k)} \|} \leqslant \mathrm{k}(A) \frac{\| \boldsymbol{r}^{(k)} \|}{\| \boldsymbol{b} \|} . \tag{15}$$

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

3.1 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

$$J(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}\,. \tag{16}$$

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 3.1 (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 $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$.*

Theorem 3.2 (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 $Hf : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ exists in $\boldsymbol{x}^*$ and is continuous in an open neighborhood of $\boldsymbol{x}^*$. If $\boldsymbol{x}^*$ is a local minimizer of f, then $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ and $Hf(\boldsymbol{x}^*)$ is positive semi-definite.*

Theorem 3.3 (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 Hf exists in $\boldsymbol{x}^*$ and is continuous in an open neighborhood of $\boldsymbol{x}^*$. If $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ and $Hf(\boldsymbol{x}^*)$ 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 $J \Rightarrow \nabla J(\boldsymbol{x}^*) = A\boldsymbol{x}^* - \boldsymbol{b} = \boldsymbol{0}$ (Theorem 3.1);
- $\boldsymbol{x}^*$ solution of $A\boldsymbol{x} = \boldsymbol{b}$ and A symmetric positive definite $\Rightarrow$ $\boldsymbol{x}^*$ (strict) minimizer of J (Theorem 3.3);
- 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 3.1 below).

Definition 3.1 (Descent Direction)**.** *A vector $\boldsymbol{p} \in \mathbb{R}^n$ is a* descent direction *for a function $f : \mathbb{R}^n \to \mathbb{R}$ at $\boldsymbol{x} \in \mathbb{R}^n$ if exists $\varepsilon > 0$ such that*

$$f(\boldsymbol{x} + \alpha\boldsymbol{p}) < f(\boldsymbol{x})\,, \tag{17}$$

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

$$\nabla f(\boldsymbol{x})^T\boldsymbol{p} < 0\,. \tag{18}$$

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

Remark 3.1 (Steepest Descent/Ascent Direction)**.** Given Definition 3.1, we observe that $\nabla f(\boldsymbol{x})$ and $-\nabla f(\boldsymbol{x})$ are the directions of steepest ascent and descent, respectively, for f in $\boldsymbol{x}$.

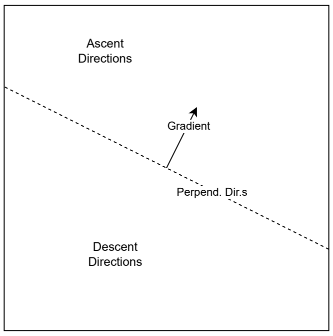

Figure 3: Example of regions of ascent, perpendicular, and descent directions in $\mathbb{R}^2$, with respect to a given gradient vector.

3.1.1 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:

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha^{(k)} \nabla J(\boldsymbol{x}^{(k)}) \,, & \forall\, k \geqslant 0 \end{cases} , \tag{19}$$

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.

$$\boldsymbol{r}^{(k)} := \boldsymbol{b} - A\boldsymbol{x}^{(k)} \,, \forall\, k \geqslant 0 \,, \tag{20}$$

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

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)} \boldsymbol{r}^{(k)} \,, & \forall\, k \geqslant 0 \end{cases} , \tag{21}$$

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 3.1 (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 3.2 (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 : $\widehat{\boldsymbol{x}}$
```

3.1.2 Convergence Properties

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

$$\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A \leqslant 2 \left(\frac{\mathrm{k}_2(A) - 1}{\mathrm{k}_2(A) + 1} \right)^k \| \boldsymbol{x}^{(0)} - \boldsymbol{x}^* \|_A \,, \quad \forall\, k \geqslant 0 \,, \tag{22}$$

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.

3.2 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

$$\begin{cases} \boldsymbol{p}^{(0)} = \boldsymbol{r}^{(0)} \\ \boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)} \,, \quad \forall\, k \geqslant 0 \end{cases} , \tag{23}$$

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

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \,, & \text{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} \,, & \forall\, k \geqslant 0 \end{cases} , \tag{24}$$

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 3.3 (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 3.4 (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 : $\widehat{\boldsymbol{x}}$
```

3.2.1 Preconditioning

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

$$\| \boldsymbol{x}^{(k)} - \boldsymbol{x}^* \|_A \leqslant 2 \left(\frac{\sqrt{\mathrm{k}_2(A)} - 1}{\sqrt{\mathrm{k}_2(A)} + 1} \right)^k \| \boldsymbol{x}^{(0)} - \boldsymbol{x}^* \|_A \,, \quad \forall\, k \geqslant 0\,, \tag{25}$$

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 (25) is lower thant the one in (22).

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

$$\boldsymbol{x} = C^{-1}\boldsymbol{y}\,. \tag{26}$$

Using (26), we have that (16) 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 performs 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 more stable, because:

$$\mathbb{I}_n R\boldsymbol{x} \approx C^{-T}AC^{-1}\boldsymbol{y} = C^{-T}\boldsymbol{b} \approx R^{-T}\boldsymbol{b} \tag{27}$$

Of course, the more $\widetilde{L}^T \approx R$, the more the density of $\widetilde{L}$, losing the advantages of sparsity.

Such kind of preconditioning is called *two-sided* or *split* preconditioning (there are also *left/right* preconditioning methods in the literature).

3.3 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.

3.4 Exercises

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

Exercise 3.1 (Gradient Method)**.** *Write a function called* `gradient_linsys` *that implement in Matlab the Algorithm 3.2 using a relative residual as stopping criterium.*

Test your function on the linear system defined by the variables inside the file `lab03_sparse_linsys.mat`*.*

Exercise 3.2 (Conjugate Gradient Method)**.** *Write a function called* `cg_linsys` *that implement in Matlab the Algorithm 3.4 using a relative residual as stopping criterium.*

Test your function on the linear system defined by the variables inside the file `lab03_sparse_linsys.mat`*.*

Exercise 3.3. *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* $\tau = 10^{-4}, \ldots, 10^{-10}$*, solve the linear systems* $A_i\boldsymbol{x} = \boldsymbol{b}_i$*, with exact solution* $\boldsymbol{x}_i^* = (1, \ldots, 1) \in \mathbb{R}^n$*,* $i = 1, 2$ *(i.e., build* $\boldsymbol{b}_i$ *as in Exercise 1.1, item 3).*

   In particular, solve the linear systems with `gradient_linsys` *of Exercise 3.1, with* `cg_linsys` *of Exercise 3.2, 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.*

4 Steepest Descent

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be given. The Steepest Descent (SD) method is an iterative optimization method that can be interpreted as the nonlinear generalization of the Gradient method; SD starts from a given vector $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$, and computes a sequence of vectors $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ characterized by

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} , \quad \forall\ k \geqslant 0 , \tag{28}$$

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 $\alpha \in \mathbb{R}^+$.

Attention: Contrary to the Gradient method, we have not a formula for computing the optimal step length $\alpha^{(k)}$ (because f is a generic differentiable function). The choice of $\alpha^{(k)}$ is problem-dependent and different strategies exist to select the optimal one.

N.B.: In this laboratory, we will use a fixed value $\alpha = \alpha^{(k)}$ for each step.

4.1 Exercises

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

Exercise 4.1 (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_functions2.mat`

Exercise 4.2 (Steepest Descent Path Visualization)**.** *Given the function of the previous exercise, write a Matlab script where you test this function using the 2-dimensional functions inside the file `test_functions2.mat` (see Section B.1). In particular, for each 2-dimensional function in the `.mat` file:*

- *run the SD procedure by varying α =1e-1,1e-2,1e-3,1e-4;*
- *repeat the previous item, but using the Finite Differences (of your choice) to compute the gradient of the functions, varying h among the values h =1e-8,1e-6,1e-4;*
- *For each procedure generated with the previous two steps, collect:*
    - *the number of steps for reaching the convergence (or a stopping criterion);*
    - *the norm of the gradient at the last step;*
    - *the coordinates of the minimizer $\widehat{\boldsymbol{x}}$ (just because we are in 2D) and the value $f(\widehat{\boldsymbol{x}})$;*
- *plot a top view of the loss f using the Matlab function `contour`, together with the sequence `xseq` in $\mathbb{R}^2$;*
- *the surface of the loss f using the Matlab function `surf`, together with the function values of the sequence `xseq` in $\mathbb{R}^3$ (i.e., coordinates plus function values). 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 4.3 (n-Dimensional Steepest Descent)**.** *Repeat the previous exercise (except for the plots) with respect to the n-dimensional function in `test_funcs_ndim.mat`. Run the test for different values of n, e.g.: $n = 10, 100, 1000$.*

5 Steepest Descent with Backtracking

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be given. In the previous laboratory we have seen that 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

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha \boldsymbol{p}^{(k)} , \quad \forall\, k \geqslant 0 , \tag{29}$$

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 $\alpha \in \mathbb{R}^+$.

5.1 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.

$$f(\underbrace{\boldsymbol{x}^{(k+1)}}_{\boldsymbol{x}^{(k)}+\alpha^{(k)}\boldsymbol{p}^{(k)}}) \leqslant f(\boldsymbol{x}^{(k)}) + c_1 \alpha^{(k)} \nabla f(\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(k)} , \tag{30}$$

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).

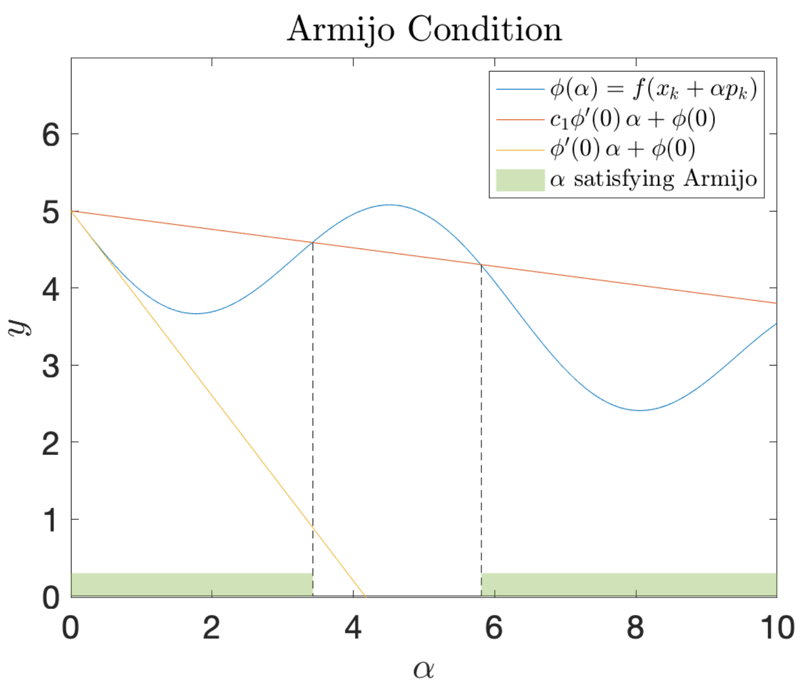

Figure 4: Example of the Armijo condition.

The backtracking strategy is an *iterative process* that looks for this value $\alpha^{(k)}$. Given a factor $\rho \in (0, 1)$ and a 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 5.1 (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.

5.2 Exercises

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

Exercise 5.1 (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 $\nabla f(\boldsymbol{x}^{(k)})$;*

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

- `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_functions2.mat`
Suggestion: *copy the code of the function of Exercise 4.1 and modify it adding the backtracking.*

Exercise 5.2 (Steepest Descent with Backtracking Path Visualization)**.** *Given the function of the previous exercise, write a Matlab script where you test this function using the 2-dimensional functions inside the file* `test_functions2.mat` *(see Section B.1). In particular, for each 2-dimensional function in the* `.mat` *file:*

- *run the SD + backtracking procedure by varying* $\rho = 0.9, 0.75, 0.5, 0.25, 0.1$*, with fixed* $\alpha_0^{(k)} = 1$ *and* c_1 *=1e-4,* `btmax`= 50*;*
- *repeat the previous item, but using the Finite Differences (of your choice) to compute the gradient of the functions, varying* h *among the values* h *=1e-8,1e-6,1e-4;*
- *For each procedure generated with the previous two steps, collect:*
  - *the number of steps for reaching the convergence (or a stopping criterion);*
  - *the norm of the gradient at the last step;*
  - *the coordinates of the minimizer* $\widehat{\boldsymbol{x}}$ *(just because we are in 2D) and the value* $f(\widehat{\boldsymbol{x}})$*;*
- *plot a top view of the loss* f *using the Matlab function* `contour`*, together with the sequence* `xseq` *in* $\mathbb{R}^2$*;*
- *the surface of the loss* f *using the Matlab function* `surf`*, together with the function values of the sequence* `xseq` *in* $\mathbb{R}^3$ *(i.e., coordinates plus function values). 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*.*
- *Make a comparison of the results obtained in this exercise with the results obtained in Exercise 4.2.*

Exercise 5.3 (n-Dimensional Steepest Descent with Backtracking)**.** *Repeat the previous exercise (except for the plots) with respect to the* n*-dimensional function in* `test_funcs_ndim.mat`*. Run the test for different values of* n*, e.g.:* $n = 10, 100, 1000$*. Make a comparison of the results obtained in this exercise with the results obtained in Exercise 4.3.*

6 Newton Method with Backtracking

Let the function $f : \mathbb{R}^n \to \mathbb{R}$ be given. The Newton 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

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha \boldsymbol{p}^{(k)}\,, \quad \forall\, k \geqslant 0\,, \tag{31}$$

where the descent direction $\boldsymbol{p}^{(k)}$ is solution of the linear system

$$Hf(\boldsymbol{x}^{(k)})\boldsymbol{p} = -\nabla f(\boldsymbol{x}^{(k)})\,, \tag{32}$$

and ∇f, Hf are the gradient and the Hessian matrix of f, respectively. Typically, $\alpha = 1$ almost always (see PROs and CONs in the next subsection).

6.1 Recalling the Motivations of Newton

We can approximate $f(\boldsymbol{x}+\boldsymbol{p})$, with a Taylor expansion, as a quadratic model, i.e.:

$$f(\boldsymbol{x}+\boldsymbol{p}) \simeq f(\boldsymbol{x}) + \boldsymbol{p}^T\nabla f(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{p}^T Hf(\boldsymbol{x})\,\boldsymbol{p} =: m_{f,\boldsymbol{x}}(\boldsymbol{p})\,. \tag{33}$$

Assuming that $Hf(\boldsymbol{x})$ is Positive Semidefinite (PSD), for each $\boldsymbol{p} \in \mathbb{R}^n$, we have that $\boldsymbol{p}^T Hf(\boldsymbol{x})\,\boldsymbol{p} \geqslant 0$ and, therefore, $m_{f,\boldsymbol{x}}(\boldsymbol{p})$ is *convex*. Then, we can compute the $\boldsymbol{p}^*$ that minimizes $m_{f,\boldsymbol{x}}(\boldsymbol{p})$, i.e. the $\boldsymbol{p}^*$ such that

$$\underbrace{\nabla f(\boldsymbol{x}) + Hf(\boldsymbol{x})\boldsymbol{p}^*}_{=\nabla_{\boldsymbol{p}}\, m_{f,\boldsymbol{x}}(\boldsymbol{p}^*)} = \boldsymbol{0}\,. \tag{34}$$

In other words, $\boldsymbol{p}^*$ is solution of the linear system (32) and is a *descent direction* for $f(\boldsymbol{x})$.

In the end, we recall the PROs and CONs of the Newton method:

(+) Under the proper assumptions of having $Hf(\boldsymbol{x}_k)$ PD, $\alpha = 1$, and $\boldsymbol{x}^{(0)}$ "good" starting point, the Newton method has fast (quadratic) rate of convergence;

(−) Computationally more expensive than Steepest Descent (computation/storage of $Hf(\boldsymbol{x}^{(k)})$ and computation of $\boldsymbol{p}^{(k)}$ solving (32));

(−) $\boldsymbol{p}^{(k)}$ is a descent direction only if $Hf(\boldsymbol{x}^{(k)})$ is PSD $\Rightarrow$ Sometimes is useful to work with a "corrected" Hessian matrix, if $Hf(\boldsymbol{x}_k)$ is not PD/PSD; i.e., we can use a *PD* matrix $B_k := Hf(\boldsymbol{x}_k) +$ *Correction* (see the *Modified Newton* method in [2]).

6.2 Backtracking and Newton

In this laboratory, we focus on the implementation of the Newton method paired with a backtracking strategy (see Section 5.1). Therefore, we point the reader to Remark 5.1 for recalling some observations related to Backtracking; furthermore, in the case of the Newton method, we update the last item of Remark 5.1 with the following one:

- The choice of $\alpha_0^{(k)}$ is problem-dependent and/or method-dependent, but it is *crucial* to choose $\alpha_0^{(k)} = 1$ in Newton/Quasi-Newton methods for (possibly) get the second-order rate of convergence.

6.3 Exercises

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

Exercise 6.1 (Newton with Backtracking)**.** *Write a Matlab function* `newton_bcktrck.m` *that implements the* Newton *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;*
- `Hessf`*: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value $Hf(\boldsymbol{x})$, where $Hf : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ is the* Hessian *of f;*
- `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 inside the file `test_functions2.mat`
Important: *it is better if you solve the linear system (32) by using the* `pcg` *Matlab function, and not the* "backslash"*. This will be useful for implementations in future laboratories.*
Suggestion: *copy the code of the function of Exercise 5.1 and modify it where it is necessary.*

Exercise 6.2 (Newton with Backtracking Path Visualization)**.** *Given the function of the previous exercise, write a Matlab script where you test this function using the 2-dimensional functions inside the file* `test_functions2.mat` *(see Section B.1). In particular, for each 2-dimensional function in the* `.mat` *file:*

- *run the Newton + backtracking procedure by varying* $\rho = 0.9, 0.75, 0.5, 0.25, 0.1$*, with fixed* c_1 *=1e-4,* `btmax`$= 50$*;*
- *repeat the previous item, but using the Finite Differences (of your choice) to compute the gradient and/or the Hessian of the functions, varying* h *among the values* h *=1e-8,1e-6,1e-4;*
- *For each procedure generated with the previous two steps, collect:*
  - *the number of steps for reaching the convergence (or a stopping criterion);*
  - *the norm of the gradient at the last step;*
  - *the coordinates of the minimizer* $\widehat{\boldsymbol{x}}$ *(just because we are in 2D) and the value* $f(\widehat{\boldsymbol{x}})$*;*
- *plot a top view of the loss* f *using the Matlab function* `contour`*, together with the sequence* `xseq` *in* $\mathbb{R}^2$*;*
- *the surface of the loss* f *using the Matlab function* `surf`*, together with the function values of the sequence* `xseq` *in* $\mathbb{R}^3$ *(i.e., coordinates plus function values). 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.*
- *Make a comparison of the results obtained in this exercise with the results obtained in Exercise 4.2 and Exercise 5.2.*

Exercise 6.3 (n-Dimensional Newton with Backtracking)**.** *Repeat the previous exercise (except for the plots) with respect to the* n*-dimensional function in* `test_funcs_ndim.mat`*. Run the test for different values of* n*, e.g.:* $n = 10, 100, 1000$*. Make a comparison of the results obtained in this exercise with the results obtained in Exercise 4.3 and Exercise 5.3.*

7 Inexact Newton and Orders of Convergence

Sometimes the computation of the Newton descent direction $\boldsymbol{p}_N^{(k)}$, solving the linear system

$$Hf(\boldsymbol{x}^{(k)})\boldsymbol{p} = -\nabla f(\boldsymbol{x}^{(k)})\,, \tag{35}$$

can be quite expensive. Therefore, in order to save computational efforts, instead of computing $\boldsymbol{p}_N^{(k)}$, we compute an "approximated descent direction" $\boldsymbol{p}_{IN}^{(k)} \approx \boldsymbol{p}_N^{(k)}$.

7.1 Inexact Descent Direction and Forcing Terms

Assuming the usage of an iterative method to solve (35), at the k-th step the iterative solver returns[2] a vector $\boldsymbol{p}^{(k)*}$ with residual such that

$$\underbrace{\| Hf(\boldsymbol{x}^{(k)})\boldsymbol{p}^{(k)*} + \nabla f(\boldsymbol{x}^{(k)}) \|}_{\text{residual}} \leqslant \varepsilon\,, \tag{36}$$

where $\varepsilon > 0$ a tolerance used for the iterative method.

Then, instead of using a small and fixed ε, the Inexact Newton method consists of using an adaptive tolerance

$$\varepsilon_k := \underbrace{\eta_k}_{\text{forcing terms}} \| \nabla f(\boldsymbol{x}^{(k)}) \| \tag{37}$$

that is relatively "large" during the first optimization steps; i.e., we use a large tolerance when we assume to be far from the optimum.

In general, under *local and regularity assumptions*, the most used forcing term sequences are the following:

- $\eta_k = 0.5$ fixed (*linear* convergence);
- $\eta_k = \min(0.5, \sqrt{\| \nabla f(\boldsymbol{x}^{(k)}) \|}) \xrightarrow{k\to\infty} 0$ (*superlinear* convergence);
- $\eta_k = \min(0.5, \| \nabla f(\boldsymbol{x}^{(k)}) \|) \in \mathcal{O}(\| \nabla f(\boldsymbol{x}^{(k)}) \|)$ (*quadratic* convergence).

N.B.: the order of convergence also depends on the iterative solver; e.g. some exceptions are observable in the case where the solver, despite the larger tolerance, returns always/often a solution $\boldsymbol{p}_{IN}^{(k)} \approx \boldsymbol{p}_N^{(k)}$ (i.e. with residual $\sim \varepsilon$).

ATTENTION: The *pcg* Matlab function uses the *relative residual* $\| A\boldsymbol{x} - \boldsymbol{b} \| / \| \boldsymbol{b} \|$ for its stopping criteria and not the *residual* $\| A\boldsymbol{x} - \boldsymbol{b} \|$ (see Section 3). Then, pay attention to defining the tolerance! Indeed, in order to have $\| Hf(\boldsymbol{x}^{(k)})\boldsymbol{p}^{(k)} - \nabla f(\boldsymbol{x}^{(k)}) \| \leqslant \eta_k \| \nabla f(\boldsymbol{x}^{(k)}) \|$, you should set the relative tolerance $\| Hf(\boldsymbol{x}^{(k)})\boldsymbol{p}^{(k)} - \nabla f(\boldsymbol{x}^{(k)}) \| / \| \nabla f(\boldsymbol{x}^{(k)}) \| \leqslant \ldots$ *exercise*...

7.2 Orders of Convergence

In this subsection, we recall the definitions concerning the order of convergence q of a sequence $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ converging to $\boldsymbol{x}^*$ and how to estimate it.

Definition 7.1 (Orders of Convergence)**.** *Let $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}} \subset \mathbb{R}^n$ be a sequence converging to $\boldsymbol{x}^* \in \mathbb{R}^n$. Let $\boldsymbol{e}^{(k)}$ denotes the error of $\boldsymbol{x}^{(k)}$ in approximating $\boldsymbol{x}^*$, i.e., $\boldsymbol{e}^{(k)} := \boldsymbol{x}^{(k)} - \boldsymbol{x}^*$. Then, we have that:*

[2] if the maximum iterations have not been reached.

- *the convergence is* Q-linear *if exists* $M \in (0,1)$ *such that*

$$\frac{\| \boldsymbol{e}^{(k+1)} \|}{\| \boldsymbol{e}^{(k)} \|} \leqslant M \tag{38}$$

  for each k *sufficiently large.*

- *the convergence is* Q-superlinear *if*

$$\lim_{k \to +\infty} \frac{\| \boldsymbol{e}^{(k+1)} \|}{\| \boldsymbol{e}^{(k)} \|} = 0\,. \tag{39}$$

- *for each* $q > 1$*, the convergence is of* Q-order q *if exists* $M > 0$ *such that*

$$\frac{\| \boldsymbol{e}^{(k+1)} \|}{\| \boldsymbol{e}^{(k)} \|^q} \leqslant M \tag{40}$$

  for each k *sufficiently large.*

*The "Q" letter in the order of convergence is for "*quotient*" and can be omitted.*

N.B.: Obviously, if a sequence converges with order q, then it converges also with order r, for each $r < q$.

Given Definition 7.1, we can describe a recipe for estimating the order of convergence of a sequence. In particular, we have that the order of convergence q of a sequence $\{\boldsymbol{x}^{(k)}\}_{k\in\mathbb{N}}$ converging to $\boldsymbol{x}^*$ is

$$q \approx \frac{\log\left(\frac{\|\boldsymbol{e}^{(k+1)}\|}{\|\boldsymbol{e}^{(k)}\|}\right)}{\log\left(\frac{\|\boldsymbol{e}^{(k)}\|}{\|\boldsymbol{e}^{(k-1)}\|}\right)} \quad \text{for } k \text{ sufficiently large}\,, \tag{41}$$

because it holds both $\| \boldsymbol{e}^{(k+1)} \| \approx \| \boldsymbol{e}^{(k)} \|^q$ and $\| \boldsymbol{e}^{(k)} \| \approx \| \boldsymbol{e}^{(k-1)} \|^q$; then, $\| \boldsymbol{e}^{(k+1)} \| \,/\, \| \boldsymbol{e}^{(k)} \| \approx (\| \boldsymbol{e}^{(k)} \| \,/\, \| \boldsymbol{e}^{(k-1)} \|)^q$, which implies $\log(\| \boldsymbol{e}^{(k+1)} \| \,/\, \| \boldsymbol{e}^{(k)} \|) \approx q \log(\| \boldsymbol{e}^{(k)} \| \,/\, \| \boldsymbol{e}^{(k-1)} \|)$ and, as a consequence, the estimate (41) holds.

However, the exact value of $\boldsymbol{x}^*$ is rarely known; nonetheless, for k sufficiently large, we can approximate $\boldsymbol{e}^{(k)}$ with $\widehat{\boldsymbol{e}}^{(k)} := \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}$. Therefore, we can further approximate the order of convergence with the following formula:

$$q \approx \frac{\log\left(\frac{\|\widehat{\boldsymbol{e}}^{(k+1)}\|}{\|\widehat{\boldsymbol{e}}^{(k)}\|}\right)}{\log\left(\frac{\|\widehat{\boldsymbol{e}}^{(k)}\|}{\|\widehat{\boldsymbol{e}}^{(k-1)}\|}\right)} \quad \text{for } k \text{ sufficiently large}\,, \tag{42}$$

7.3 Exercises

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

Exercise 7.1 (Forcing Term Functions)**.** *Define three function handles* fterms_lin, fterms_suplin, fterms_quad *such that, given* k *and* $\nabla f(\boldsymbol{x}_k)$ *(as input arguments* **k** *and* **gradfk***), returns the* linear, superlinear, *and* quadratic *forcing terms* η_k*, respectively, described in the appendix of this document*[3]*.*

Then, save this three variables in a file named forcing_terms.mat*.*

[3]Actually, the input k is not necessary to define the linear, superlinear, quadratic forcing terms in the appendix; nonetheless, we introduce this variable because in the next exercise, one may want to use a forcing term depending also from k and not only $\nabla f(\boldsymbol{x}_k)$.

Exercise 7.2 (Implementation of the Inexact Newton Method)**.** *Write a Matlab function called* innewton_bcktrck.m *that implements the inexact Newton method. Do it by modifying the* newton_bcktrck.m *function, in particular by adding the following extra input arguments:*

- `fterms`*: a function handle characterizing the sequence of* forcing terms η_k *of the inexact Newton method. This function handle must be like the ones defined in the previous exercise; i.e., depending on the input arguments* `k` *and* `gradfk`*,.*
- `pcg_maxit`*: the maximum number of iterations allowed for the* pcg *solver to compute* $\boldsymbol{p}_{IN}^{(k)}$ *of the inexact Newton method.*

Moreover, add also the following extra output:

- `pcgiterseq`*: the number of pcg-iterations executed at each main iteration for computing the direction.*

Once you have written the function, test it using the function handles in forcing_terms.mat *and the data in* test_functions2.mat *(just for checking the code).*

Exercise 7.3 (Rates of Convergence)**.** *Write two functions,* `expconv_xk.m` *and* `expconv_ek.m`*, that compute a sequence of experimental orders of convergence* $\{q^{(k)}\}_{k\geqslant 2}$ *using* (42)*, given the sequence* $\{\boldsymbol{x}^{(k)}\}_{k\geqslant 0}$ *(function* `expconv_xk.m`*) or the sequence* $\{\widehat{\boldsymbol{e}}^{(k)}\}_{k\geqslant 1}$ *(function* `expconv_xk.m`*).*

Now, write another function, `trueconv_xk.m`*, that computes a sequence of orders of convergence* $\{q^{(k)}\}_{k\geqslant 2}$ *using* (41)*, given the sequence* $\{\boldsymbol{x}^{(k)}\}_{k\geqslant 0}$ *and the true solution* $\boldsymbol{x}^*$ *of the minimization problem.*

Exercise 7.4 (Application to the n-dimensional Rosenbrock Function)**.** *Write a Matlab script where, with respect to the* 100*-dimensional, parametric, Rosenbrock function (see* (54)*), you compare the results obtained with the following methods:*

- *Steepest descent;*
- *Newton;*
- *Inexact Newton, "linear" forcing term;*
- *Inexact Newton, "superlinear" forcing term;*
- *Inexact Newton, "quadratic" forcing term;*

In particular, analyze the behavior of the methods and estimate and verify their order of convergence, varying the parameter $\alpha = 1, 5, 10, 100$ *of the Rosenbrock function. How does the convergence of the methods changes, varying* α*?*

Concerning the other parameters: `alpha0`$= 10^{-3}$ *(steepest descent),* $\boldsymbol{x}^{(0)} = \boldsymbol{0} = (0, \dots, 0) \in \mathbb{R}^{100}$*,* `tolgrad`$= 10^{-8}$*,* `c1`$= 10^{-4}$*,* `rho`$= 0.8$*,* `kmax`$= 5000$*,* `btmax`$= 50$*,* `pcg_maxit`$= 50$*.*

8 Newton and Inexact Newton Methods for Nonlinear Equations

Let us consider a nonlinear function $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$, with Jacobian matrix in $\mathbb{R}^{n\times n}$ denoted by $J\boldsymbol{F}$, and a Newton-Tangent[4] method to solve the nonlinear system $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$, i.e.:

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \ \textit{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{p}^{(k)}\,, \ \forall\ k \geqslant 0 \end{cases} \tag{43}$$

where $\boldsymbol{p}^{(k)}$ is solution of the linear system

$$J\boldsymbol{F}(\boldsymbol{x}^{(k)}) \cdot \boldsymbol{p} = -\boldsymbol{F}(\boldsymbol{x}^{(k)}) \tag{44}$$

Attention: under suitable assumptions, the Newton-Tangent method displays a fast rate of convergence (quadratic) when it is close to a solution, but it suffers a lot from the choice of the initial guess $\boldsymbol{x}^{(0)}$; then, if a good initial guess is not used, the method is likely to be not able to converge. Moreover, not necessarily the linear system (44) has a solution (i.e., $J\boldsymbol{F}(\boldsymbol{x}^{(k)})$ is singular). See Section 8.1 for more details about this method.

8.1 Newton-Tangent Method and Optimization of the Merit Function

Let $f(\boldsymbol{x})$ be a *merit function* of $\boldsymbol{F}$ such that:

$$f(\boldsymbol{x}) = \frac{1}{2} \parallel \boldsymbol{F}(\boldsymbol{x}) \parallel^2 \ . \tag{45}$$

We observe the following relationships between $\boldsymbol{F}$ and f, especially with respect to the sequence of the Newton-Tangent method. In particular, it holds

$$\nabla f(\boldsymbol{x}) = J\boldsymbol{F}(\boldsymbol{x})^T \boldsymbol{F}(\boldsymbol{x})\,, \tag{46}$$

then:

1. $\boldsymbol{F}(\boldsymbol{x}^*) = \boldsymbol{0} \Rightarrow \nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ and $f(\boldsymbol{x}^*) = \boldsymbol{0}$ (i.e., $\boldsymbol{x}^*$ global minimizer of f);
2. ... but $\nabla f(\widehat{\boldsymbol{x}}) = \boldsymbol{0}$ (i.e., $\widehat{\boldsymbol{x}}$ local minimizer[5] of f) $\not\Rightarrow \boldsymbol{F}(\widehat{\boldsymbol{x}}) = \boldsymbol{0}$; i.e., only the global minima of f are solutions of $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$, while the local minima are not;
3. $\boldsymbol{p}^{(k)}$ is a descent direction for f at $\boldsymbol{x}^{(k)}$ (if $\boldsymbol{F}(\boldsymbol{x}^{(k)}) \neq \boldsymbol{0}$), because:

$$\begin{aligned} \boldsymbol{p}^{(k)\,T}\nabla f(\boldsymbol{x}^{(k)}) = \boldsymbol{p}^{(k)\,T} J\boldsymbol{F}(\boldsymbol{x}^{(k)})^T \boldsymbol{F}(\boldsymbol{x}^{(k)}) &= \left(J\boldsymbol{F}(\boldsymbol{x}^{(k)})\boldsymbol{p}^{(k)}\right)^T \boldsymbol{F}(\boldsymbol{x}^{(k)}) = \\ &= -\boldsymbol{F}(\boldsymbol{x}^{(k)})^T \boldsymbol{F}(\boldsymbol{x}^{(k)}) = - \parallel \boldsymbol{F}(\boldsymbol{x}^{(k)}) \parallel^2 < \\ &< 0\,. \end{aligned} \tag{47}$$

Since $\boldsymbol{p}^{(k)}$ is a descent direction for f at $\boldsymbol{x}^{(k)}$, the Newton sequence (43)-(44) is also a descent method for the merit function f. Therefore, we can implement a *line search* strategy strategy to improve the efficiency of the method. Specifically, (43)-(44) changes into

$$\begin{cases} \boldsymbol{x}^{(0)} \in \mathbb{R}^n \ \textit{given} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}\,, \ \forall\ k \geqslant 0 \end{cases} \tag{48}$$

[4]we use the name "Newton-Tangent" only to distinguish this method from its homonym for optimization.
[5]We are not considering the case of other critical points, such as maximizers or saddle points.

where $\boldsymbol{p}^{(k)}$ is given by (44) and $\alpha^{(k)}$ is selected throughout a line search method applied with respect to the merit function f. For example, if we apply a *backtracking* strategy, we check the Armijo condition (30) with respect to the merit function f.

Suggestion (useful for implementing the Armijo condition): Look carefully at item 3 for a "smart" implementation of the Armijo condition.

8.2 Inexact Newton-Tangent Method for Nonlinear Equations

Sometimes the computation of the Newton-Tangent direction $\boldsymbol{p}_N^{(k)}$, solving the linear system (44), can be quite expensive. Therefore, in order to save computational efforts, instead of computing $\boldsymbol{p}_N^{(k)}$, we compute an "approximated Newton-Tangent direction" $\boldsymbol{p}_{IN}^{(k)} \approx \boldsymbol{p}_N^{(k)}$.

8.2.1 Inexact Descent Direction and Forcing Terms

Assuming the usage of an iterative method to solve (44), at the k-th step the iterative solver returns[6] a vector $\boldsymbol{p}^{(k)*}$ with residual such that

$$\underbrace{\| J\boldsymbol{F}(\boldsymbol{x}^{(k)})\boldsymbol{p}^{(k)*} + \boldsymbol{F}(\boldsymbol{x}^{(k)}) \|}_{\text{residual}} \leqslant \varepsilon\,, \tag{49}$$

where $\varepsilon > 0$ is the arbitrary small tolerance used for the iterative method.

Then, instead of using a small and fixed ε, the Inexact Newton consists of using an adaptive tolerance

$$\varepsilon_k := \underbrace{\eta_k}_{\text{forcing terms}} \| \boldsymbol{F}(\boldsymbol{x}^{(k)}) \| \tag{50}$$

that is relatively "large" during the first optimization steps; i.e., we use a large tolerance when we assume to be far from the optimum.

In general, under local and regularity assumptions, the most used forcing term sequences are the following:

- $\eta_k = 0.5$ fixed (*linear* convergence);
- $\eta_k = \min(0.5, \sqrt{\| \boldsymbol{F}(\boldsymbol{x}^{(k)}) \|}) \xrightarrow{k\to\infty} 0$ (*superlinear* convegence);
- $\eta_k = \min(0.5, \| \boldsymbol{F}(\boldsymbol{x}^{(k)}) \|) \in \mathcal{O}(\| \boldsymbol{F}(\boldsymbol{x}^{(k)}) \|)$ (*quadratic* convergence).

N.B.: the rate of convergence depends also on the iterative solver; e.g. some exceptions are observable in the case where the solver, despite the larger tolerance, returns always/often a solution $\boldsymbol{p}_{IN}^{(k)} \approx \boldsymbol{p}_N^{(k)}$ (i.e. with residual $\sim \varepsilon$).

ATTENTION: The *gmres* Matlab function uses the *relative residual* for its stopping criteria and not the *residual* (see Section 3). Then, pay attention in defining the tolerance!

ATTENTION: item 3 cannot be exploited anymore for a "smart" implementation of the Armijo condition, because we have (almost always) $\boldsymbol{p}_{IN}^{(k)} \neq \boldsymbol{p}_N^{(k)}$

8.3 Test Functions

For this laboratory we use as test function the functions the functions listed in Appendix C.1.

[6]if the maximum iterations have not been reached.

8.4 Exercises

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

Exercise 8.1 (Newton Method for Nonlinear Systems)**.** *Write a Matlab function* newtonsolve_bcktrck.m *that implements the Newton method for nonlinear systems, i.e., systems $\boldsymbol{F}(\boldsymbol{x}) = \mathbf{0}$ of n equations and n variables.*

Moreover, implement the backtracking strategy, reading this method as a descent method for the merit function $f(\boldsymbol{x}) = (1/2) \| \boldsymbol{F}(\boldsymbol{x}) \|^2$.

In particular, use the following inputs.

- *`x0`:* a column vector *of n elements representing the starting point for the method;*
- *`F`: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the vector of values $\boldsymbol{F}(\boldsymbol{x}) \in \mathbb{R}^n$;*
- *`JF`: a* function handle *variable that, for each column vector $\boldsymbol{x} \in \mathbb{R}^n$, returns the value of the Jacobian matrix $J_{\boldsymbol{F}}(\boldsymbol{x}) \in \mathbb{R}^{n\times n}$;*
- *`kmax`: an* integer scalar value *characterizing the maximum number of iterations of the method;*
- *`Ftol`: a* real scalar value *defining the tolerance with respect to the norm of $\boldsymbol{F}$ to stop the method when we are near to the solution of the system.*
- *`c1`: the factor c_1 for the Armijo condition that must be a scalar in $(0, 1)$;*
- *`rho`: fixed factor, less than 1, used to reduce the step factor during the backtracking strategy;*
- *`btmax`: maximum number of steps of the backtracking strategy.*

The outputs of the function must be

- *`xk`: the last vector $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ computed by the method before it stops;*
- *`normFk`: the value $\| \boldsymbol{F}(\boldsymbol{x}^{(k)}) \|$;*
- *`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 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 method.*

Once you have written the function, test it using the data inside the file `test_nonlinsys.mat`.

Exercise 8.2 (Implementation of the Inexact Newton-Tangent Method)**.** *Write a Matlab function,* innewtonsolve_bcktrck.m, *that implements the inexact Newton-Tangent method. Do it modifying the* newtonsolve_bcktrck.m *function, in particular adding the following extra inputs:*

- *`fterms`: a function handle characterizing the sequence of* forcing terms *η_k of the inexact Newton method. This function handle must be like the ones defined in the previous exercise; i.e., depending on the input arguments `k` and `gradfk`,.*
- *`gmres_maxit_inner` and `gmres_maxit_outer`: the maximum number of inner/outer iterations allowed for the* gmres *solver to compute $\boldsymbol{p}_{IN}^{(k)}$ of the inexact Newton-Tangent method.*

Moreover, add also the following extra output:

- `gmresiterseq`*: the number of inner/outer gmres-iterations executed at each main iteration for computing the direction.*

Once you have written the function, test it using the function handles in test_nonlinsys.mat *(just for checking the code).*

Exercise 8.3 (Applications)**.** *Given the function of the previous exercise, write a Matlab script where you test these functions on the data inside the file* `test_nonlinsys.mat` *(containing the test functions of Section C.1). Try both these starting points: with $\boldsymbol{x}^{(0)} = (-5, -2.5), (-5, -5) \in \mathbb{R}^2$. For the backtracking parameters, use the same values as the exercises in previous sections.*

Then, given $\boldsymbol{F} = (f_1, f_2)$ and $\boldsymbol{G} = (g_1, g_2)$, plot:

- *the* `contour` *and the* `surf` *plots of the functions f_1, f_2 and the sequence* `xseq`*. Repeat also for g_1, g_2;*
- *try to merge the previous plots, observing the different path of the sequence* `xseq` *on both f_1 and f_2. Repeat for g_1, g_2.*
- *using the* subplot *command, observe "in parallel" the path of the sequence* `xseq` *both in the (x_1, x_2)-plane (the domain) and in the (f_1, f_2)-plane (the codomain). Repeat for g_1, g_2.*

Special request: *try also to implement the case where the Jacobian is unknown and must be approximated.*

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/0B1I3FJn9Sg
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).

B Test Functions - Optimization

When you have to test an optimization method, it is useful to use some "standard" test functions[7]. Here we present some test functions that will be used in the laboratories.

B.1 The Rosenbrock and the Himmelblau Test Functions

In this section we introduce the the 2-dimensional *Rosenbrock* function and the *Himmelblau* function that are the functions f_2 and f_3, respectively, in the file `test_functions2.mat`. The other function, denoted as f_1, is instead a simple paraboloid.

- **Paraboloid** ($n = 2$):

$$\begin{aligned} f_1(x_1, x_2) &= x_1^2 + 4x_2^2 + 5 \\ \nabla f_1(x_1, x_2) &= \begin{bmatrix} 2x_1 \\ 8x_2 \end{bmatrix} \\ H_{f_1}(x_1, x_2) &= \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix} \quad \textit{(constant)} \\ \text{global minima} &= \{f_1(0,0) = 5\} \\ \text{test } \boldsymbol{x}^{(0)} &= \{(0,0), (5,0)\} \end{aligned} \tag{51}$$

[7]E.g., see https://en.wikipedia.org/wiki/Test_functions_for_optimization

- **Rosenbrock** $(n = 2)$:

$$\begin{aligned}
f_2(x_1, x_2) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\
\nabla f_2(x_1, x_2) &= \begin{bmatrix} 400x_1^3 - 400x_1x_2 + 2x_1 - 2 \\ 200(x_2 - x_1^2) \end{bmatrix} \\
H_{f_2}(x_1, x_2) &= \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix} \\
\text{global minima} &= \{f_2(1, 1) = 0\} \\
\text{test } \boldsymbol{x}^{(0)} &= \{(1.2, 1.2), (-1.2, 1)\}
\end{aligned} \tag{52}$$

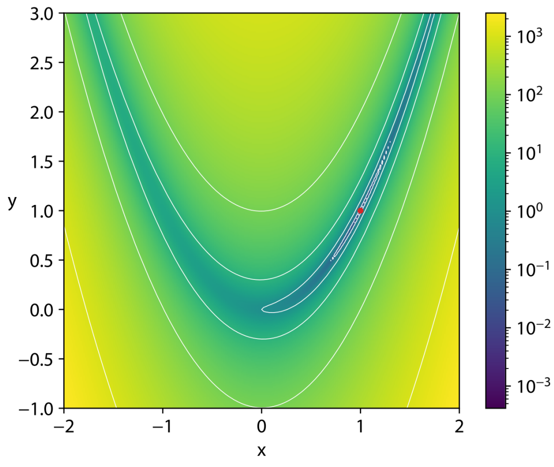

Figure 5: Rosenbrock function $(n = 2)$, top view

- **Himmelblau**:

$$\begin{aligned}
f_3(x_1, x_2) &= (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\
\nabla f_3(x_1, x_2) &= \begin{bmatrix} 4x_1^3 + 4x_1x_2 - 42x_1 + 2x_2^2 - 14 \\ 4x_2^3 + 4x_1x_2 - 26x_2 + 2x_1^2 - 22 \end{bmatrix} \\
H_{f_3}(x_1, x_2) &= \begin{bmatrix} 12x_1^2 + 4x_2 - 42 & 4x_1 + 4x_2 \\ 4x_1 + 4x_2 & 12x_2^2 + 4x_1 - 26 \end{bmatrix} \\
\text{global minima} &= \left\{ f_3(\boldsymbol{x}^{*(i)}) = 0 \mid i = 1, \ldots, 4 \right\}, \quad \text{where} \\
\boldsymbol{x}^{*(1)} &= (3, 2), \quad \boldsymbol{x}^{*(2)} \simeq (-2.805118, 3.131312) \\
\boldsymbol{x}^{*(3)} &\simeq (-3.779310, -3.283186), \quad \boldsymbol{x}^{*(4)} \simeq (3.584428, -1.848126) \\
\text{test } \boldsymbol{x}^{(0)} &= \left\{(0, 0), \text{random in } [-4, 4]^2\right\}
\end{aligned} \tag{53}$$

B.2 n-dimensional Test Function

In this section, we introduce the n-dimensional, parametric, *Rosenbrock* function. See the 2-dimensional version in Section B.1.

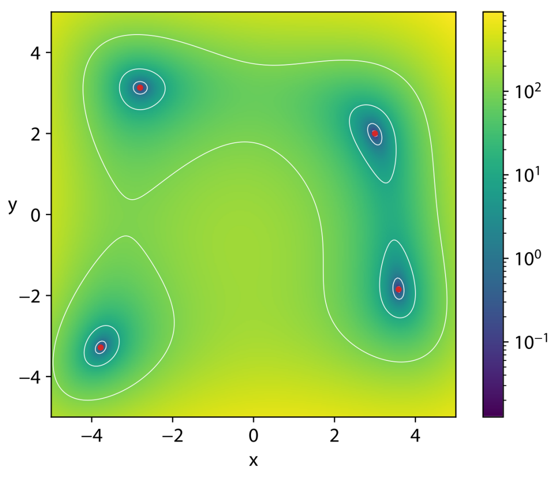

Figure 6: Himmelblau function, top view

- **Parametric Rosenbrock ($\alpha > 0$, $n \in \mathbb{N}$):**

$$\begin{aligned} f(\boldsymbol{x}) &= \sum_{i=1}^{n-1} \left(\alpha(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right) \\ \nabla f(\boldsymbol{x}) &= \ldots \textit{exercise} \ldots \\ Hf(\boldsymbol{x}) &= \ldots \textit{exercise} \ldots \\ \text{global minima} &= \{f(1, \ldots, 1) = 0\} \\ \text{test } \boldsymbol{x}^{(0)} &= \{(1.2, \ldots, 1.2), (-1.2, 1, \ldots, -1.2, 1)\} \quad (\text{assuming } n \text{ even}) \end{aligned} \tag{54}$$

Hints for computing derivatives:

$$\frac{\partial}{\partial x_i} f(\boldsymbol{x}) = \frac{\partial}{\partial x_i} \left(\alpha(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 + \alpha(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2\right), \tag{55}$$

for each $i = 2, \ldots, n-1$; for $i = 1, n$, small changes occur:

$$\begin{aligned} \frac{\partial}{\partial x_1} f(\boldsymbol{x}) &= \frac{\partial}{\partial x_1} \left(\alpha(x_2 - x_1^2)^2 + (1 - x_1)^2\right) \\ \frac{\partial}{\partial x_n} f(\boldsymbol{x}) &= \frac{\partial}{\partial x_n} \left(\alpha(x_n - x_{n-1}^2)^2 + (1 - x_{n-1})^2\right), \end{aligned} \tag{56}$$

Moreover, you can generalize (55) to the case of second order derivatives for computing the Hessian, observing also that most of its elements are zeros; specifically, the Hessian is tridiagonal and symmetric (see Figure 7):

$$\text{if } j < (i-1) \text{ or } j > (i+1) \quad \Rightarrow \quad \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) = 0\,. \tag{57}$$

Both the gradient and the Hessian of the function can be written directly as a function handle; nonetheless, if you prefer to have a more readable code, you can write them as a Matlab function instead (loading them as function handle variables just for running the optimization functions).

Hints for computing Finite Differences:

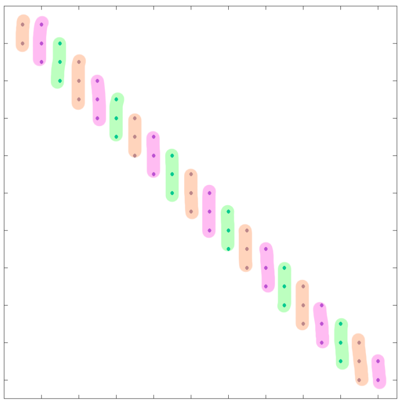

Figure 7: Scheme of the Hessian for the n-dimensional parametric Rosenbrock function. Colors show an example of exploiting the sparsity of the Hessian if approximated as the Jacobian of the exact gradient.

Let $f_1 : \mathbb{R}^2 \to \mathbb{R}$ and $f_2 : \mathbb{R} \to \mathbb{R}$ be such that $f_1(y_1, y_2) = \alpha(y_1 - y_2^2)^2$ and $f_2(y) = (1-y)^2$. Therefore, we have that

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \left(\alpha(x_{i+1} - x_i^2)^2 + (1-x_i)^2\right) = \sum_{i=1}^{n-1} \left(f_1(x_{i+1}, x_i) + f_2(x_i)\right) \tag{58}$$

and

$$\begin{aligned} f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) = & \left(f_1(\widehat{x}_2, \widehat{x}_1)) + f_2(\widehat{x}_1)\right) + \ldots + \\ & + \left(f_1(\widehat{x}_i + h, \widehat{x}_{i-1}) + f_2(\widehat{x}_{i-i})\right) + \left(f_1(\widehat{x}_{i+1}, \widehat{x}_i + h) + f_2(\widehat{x}_i + h)\right) + \ldots + \\ & + \left(f_1(\widehat{x}_n, \widehat{x}_{n-1}) + f(\widehat{x}_{n-1})\right), \quad \forall\, i = 2, \ldots, n-1, \\ \\ f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_1) = & \left(f_1(\widehat{x}_2, \widehat{x}_1 + h)) + f_2(\widehat{x}_1 + h)\right) + \ldots + \\ & + \left(f_1(\widehat{x}_n, \widehat{x}_{n-1}) + f(\widehat{x}_{n-1})\right), \\ \\ f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_n) = & \left(f_1(\widehat{x}_2, \widehat{x}_1)) + f_2(\widehat{x}_1)\right) + \ldots + \\ & + \left(f_1(\widehat{x}_n + h, \widehat{x}_{n-1}) + f(\widehat{x}_{n-1})\right). \end{aligned} \tag{59}$$

Then, for example:

$$\begin{aligned} f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_i) - f(\widehat{\boldsymbol{x}}) = & f_1(\widehat{x}_i + h, \widehat{x}_{i-1}) + \left(f_1(\widehat{x}_{i+1}, \widehat{x}_i + h) + f_2(\widehat{x}_i + h)\right) - \\ & - \left(f_1(\widehat{x}_i, \widehat{x}_{i-1}) + \left(f_1(\widehat{x}_{i+1}, \widehat{x}_i) + f_2(\widehat{x}_i)\right)\right), \quad \forall\, i = 2, \ldots, n-1, \\ \\ f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_1) - f(\widehat{\boldsymbol{x}}) = & \left(f_1(\widehat{x}_2, \widehat{x}_1 + h)) + f_2(\widehat{x}_1 + h)\right) - \left(f_1(\widehat{x}_2, \widehat{x}_1)) + f_2(\widehat{x}_1)\right), \\ \\ f(\widehat{\boldsymbol{x}} + h\boldsymbol{e}_n) - f(\widehat{\boldsymbol{x}}) = & f_1(\widehat{x}_n + h, \widehat{x}_{n-1}) - f_1(\widehat{x}_n, \widehat{x}_{n-1}). \end{aligned} \tag{60}$$

N.B.: all these observations can be generalized to the diagonals of the Hessian matrix (see Figure 7) if computed with second order FD (not as approximated Jacobian of the exact gradient).

C Test Functions - Equations and Systems

In this section, we list some test functions for nonlinear equations and systems.

C.1 Nonlinear Equations

We introduce the following test functions of the type $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$:

- $\boldsymbol{F} : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\boldsymbol{F}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 - x_2 - e^{-x_1} \\ 2x_2 - x_1 - e^{-x_2} \end{bmatrix}, \tag{61}$$

  and with jacobian

$$J\boldsymbol{F}(\boldsymbol{x}) = \begin{bmatrix} 2e^{-x_1} & -1 \\ -1 & 2e^{-x_2} \end{bmatrix}. \tag{62}$$

- $\boldsymbol{G} : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\boldsymbol{G}(\boldsymbol{x}) = \begin{bmatrix} g_1(\boldsymbol{x}) \\ g_2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} x_2^2 - 1 \\ \sin x_1 - x_2 \end{bmatrix}, \tag{63}$$

  and with jacobian

$$J\boldsymbol{G}(\boldsymbol{x}) = \begin{bmatrix} 0 & 2x_2 \\ \cos x_1 & -1 \end{bmatrix}. \tag{64}$$

N.B.: These functions are stored in the file `test_nonlinsys.mat`.

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.

[2] Sandra Pieraccini. “Numerical Optimization for Large Scale Problems and Stochastic Optimization”. In: *Draft slides* (2023).