Project of

Computational Mathematics for Learning and Data Analysis

project without ML n. 14

Andrea Lisi, Francesca Sbolgi

April 16, 2018

Contents

Chapter 1

Introduction

When large matrices are involved, iterative methods are preferred to the direct ones since typically a direct method requires $O(m^3)$ iterations of work: way too much for really big systems. Usually, these big matrices are not random but may arise from particular problems and therefore have a particular structure.
One of the most common feature is *sparsity*: this is exploited by iterative algorithms to improve their speed during the computation of the Krylov Subspace vectors, for instance. The computation of the Krylov Subspace is done by multiplying a vector v by the same matrix A over and over, so, optimizing this step is crucial for iterative methods. This step is called **Black Box algorithm**, i.e. an algorithm that given an input vector $\mathbf{v}$, returns the result of $A * \mathbf{v}$.
A good representation of the sparse A will decrease (a lot) the computation time needed by $A * \mathbf{v}$. Direct methods, instead, may not exploit such feature since a decomposition (such as Cholesky) may lead to dense factors of A.

Our problem requires to solve a sparse linear system of the form

$$ED^{-1}E^T\mathbf{x} = \mathbf{b}$$

using the Conjugate Gradient method.
In **chapter 2** we explain briefly the context where our problem is located and some auxiliary techniques to help the CG performance, such as preconditioning; in **chapter 3** we expose more precisely the structure of our problem and a suited way to solve it considering its peculiarities; finally, in **chapter 4** we show the implementation of the CG algorithm, the data used to test it and a few results such as time spent, accuracy of the solution etc.

Chapter 2

State of the Art

In this chapter we present the the mathematical tools and features we deal with. As introduced in chapter 1, we face a problem of the form

$$ED^{-1}E^T\mathbf{x} = \mathbf{b}$$

where D is a diagonal matrix, describing the weights on a graph, and E is the edge-node adjacency matrix of the same graph: the resulting matrix $A = ED^{-1}E^T$ is symmetric and positive semi-definite. We will talk more in detail about them in chapter 3. These kind of systems arise from solving the KKT system of **Flow Optimization problems**, but they are not the topic of interest of this paper. A good reference is [2].
Optimization problems involving large matrices are efficiently solved by **Interior Point Methods**. A sub-problem of Interior Point methods needs to solve systems of the form $A\Theta A^T$ using iterative methods rather than direct methods since the former can be stopped as soon we reach a reasonably good approximation of the solution [4].
An iterative method that can be used in IP methods is the **Conjugate Gradient method** since the matrix $A\Theta A^T$ is symmetric and positive semi-definite and CG performs really well if the eigenvalues are well distributed [1]. A good reference for a detailed look into IP methods is [3].

2.1 Conjugate Gradient

The conjugate gradient is an iterative method for solving large systems of linear equations, it's especially suited for sparse matrices [9].
Our goal is to find the local minimum of a function of n variables $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$, where A is a square, symmetric, positive-definite matrix. In order to find the minimum we search for points where the gradient of the function is zero, i.e. we try to solve systems of the form

$$A\mathbf{x} - \mathbf{b} = 0,$$

where $\mathbf{b}$ is a known vector and A is a known square, symmetric and positive-definite matrix. Since A is positive-definite it means that the function is quadratic with a parabolic shape that opens upwards. Therefore there is only one point in which the gradient is zero and it represent the global minimum of the function.
The CG algorithm derives from the steepest descent method to which some changes have been made to make it more efficient.
In the steepest descent method we start from a point and we do a step in the direction where the function decreases more quickly. Since the gradient computed in a certain point indicates the steepest increase direction of the function there, we should take a step in the opposite direction. We call this values residual

$$\mathbf{r_0} = -f'(\mathbf{x_0}) = \mathbf{b} - A\mathbf{x_0}$$

that indicates both the direction of the steepest descent and how far we are from the correct value of $\mathbf{b}$.
Once decided the direction we can use the cutting plane defined by it to choose the step size α that we need to compute the new point $\mathbf{x_1} = \mathbf{x_0} + \alpha\mathbf{r_0}$. The intersection of the function with that surface will result in a parabola so, since we want to minimize the function in that direction, we should take a step to get to the vertex. This means that α should be such that the directional derivative along that line is equal to zero:

$$\frac{d}{d\mathbf{r_0}} f(\mathbf{x_1}) = 0 \implies \mathbf{r_0}^T f'(\mathbf{x_1}) = f'(\mathbf{x_1})^T \mathbf{r_0} = \mathbf{r_1}^T \mathbf{r_0} = 0$$

By substituting $\mathbf{r_1} = \mathbf{b} - A\mathbf{x_1}$ and $\mathbf{x_1} = \mathbf{x_0} + \alpha\mathbf{r_0}$ and performing a little algebra we obtain the value of α as

$$\alpha = \frac{\mathbf{r_0}^T \mathbf{r_0}}{\mathbf{r_0}^T A \mathbf{r_0}}.$$

The same procedure can be repeated at each iteration, getting a better approximation $\mathbf{x_i}$ of the exact solution, and we can stop as soon as we get a desired precision. In this way, the algorithm will always converge to the solution. The problem is the number of iterations required, indeed it might do multiple steps among the same direction.
The idea to improve this is to take a set of orthogonal directions $\mathbf{d_0}, \ldots, \mathbf{d_{n-1}}$ and perform only one step in each direction:

$$\mathbf{x_{i+1}} = \mathbf{x_i} + \alpha\mathbf{d_i}$$

This way, after at most n steps, the algorithm will terminate.
The issue now is to choose the right step size to line up evenly with $\mathbf{x}$ in that direction, i.e. that $\mathbf{d_i}$ is orthogonal to the difference $\mathbf{x_{i+1}} - \mathbf{x}$. Unfortunately, we can't compute this directly because we don't know the value of $\mathbf{x}$. However we can solve this quite easily by taking the directions A-orthogonal instead, that is:

$$\mathbf{d_i}^T A\mathbf{d_{i-1}} = 0.$$

So we want to choose α such that $\mathbf{d_i}^T A(\mathbf{x_{i+1}} - \mathbf{x}) = 0$ which we know how to solve because:

$$\mathbf{d_i}^T A(\mathbf{x_{i+1}} - \mathbf{x}) = \mathbf{d_i}^T A(\mathbf{x_i} + \alpha\mathbf{d_i} - \mathbf{x}) = \mathbf{d_i}^T A\alpha\mathbf{d_i} + \mathbf{d_i}^T A(\mathbf{x_i} - \mathbf{x}) = \mathbf{d_i}^T A\alpha\mathbf{d_i} + \mathbf{d_i}^T (A\mathbf{x_i} - \mathbf{b}) = \mathbf{d_i}^T A\alpha\mathbf{d_i} + \mathbf{d_i}^T(-\mathbf{r_i}) = 0.$$

From which we derive that

$$\alpha = \frac{\mathbf{d_i}^T(\mathbf{r_i})}{\mathbf{d_i}^T A\mathbf{d_i}}.$$

We can observe that since we take only one step in each direction, it means that the residual has to be orthogonal to all the previous search directions.
The CG exploits this property by constructing the set of directions as a conjugation of the residual, indeed this will assure to build linearly independent directions. Moreover as each residual is orthogonal to the old directions, it must be also orthogonal to the previous residuals.
Note that each $\mathbf{r_i}$ is built as a linear combination of the previous residuals and $A\mathbf{d_i}$, so it means that the span formed by the residuals is the same one produced by the search directions and the subspace formed is the **Krylov subspace** of $(A, \mathbf{b})$.
We have seen how in exact arithmetic the conjugate gradient terminates after at most n iterations. The only reason why it may not converge on a computer are round-off errors. Mainly if the problem is ill conditioned, hence that the condition number of the matrix is very high, it may not converge not even slowly to the solution. A common way to deal with ill conditioning is to apply some form of **preconditioning**, i.e. multiply to our linear system a preconditioning matrix in order to decrease the original condition number and therefore improve convergence.
Another problem is that iterative methods are particularly efficient when working with really large scale systems, i.e. for systems so big that n iterations are still too many. Otherwise for smaller systems usually direct methods are more suited.
Some examples where CG is practical are applications with big systems of equations or graph representations of networks where the number of nodes may be very high.
Since we face these problems in particular, where our linear system is close related to a **Laplacian matrix**, we postpone a more in depth analysis in chapter 3.

2.1.1 Conjugate Gradient Algorithm

The algorithm 1 reports a standard implementation of the CG algorithm.
As we can see, it takes as input the matrix $A \in \mathbb{R}^{n\times n}$ and the vector $\mathbf{b} \in \mathbb{R}^n$ that define the problem to be solved $A\mathbf{x} = \mathbf{b}$. The other parameter is *eps* that is needed to

Input: A: known matrix
b: known vector
n: matrix dimension
eps: precision required
Result: x: solution of the system $Ax = b$

$d_0, r_0 = b$;
$x_0 = [0, \ldots, 0]$;
$k = 1$;
while $k < n \vee norm(r_k) > eps$ **do**
  $\alpha = (r_{k-1}^T \cdot r_{k-1})/(d_{k-1}^T \cdot Ad_{k-1})$;
  $x_k = x_{k-1} + \alpha \cdot d_{k-1}$;
  $r_k = r_{k-1} - \alpha \cdot Ad_{k-1}$;
  $\beta_k = (r_k^T \cdot r_k)/(r_{k-1}^T \cdot r_{k-1})$;
  $d_k = r_k + \beta_k \cdot d_{k-1}$;
  $k = k + 1$;
end
$x = x_k$

Algorithm 1: CG algorithm.

stop the algorithm before it has done n iteration if the current solution is close enough to the exact one.
As we have explained above, the algorithm at each iteration has to compute the step size α, the approximated solution $\mathbf{x_k}$, the direction $\mathbf{d_k}$ to take the step and the residual $\mathbf{r_k}$ that shows how far we are from the correct solution.

2.2 Preconditioning

Preconditioning is a technique to reduce the condition number of a matrix.
Given an ill-conditioned matrix A we want to apply a transformation $\tilde{A} = M^{-1}A$ where M is an easily invertible matrix and it's chosen such that the condition number of $\tilde{A}$ is smaller than the one of A. In this case the matrix M is called preconditioner of A [6].
The use of preconditioners is particularly useful when solving a linear system $A\mathbf{x} = \mathbf{b}$. Indeed we can calculate x indirectly by solving the preconditioned system $\tilde{A}\mathbf{x} = \tilde{\mathbf{b}}$ obtained by pre-multiplying the original system by M^{-1}:

$$A\mathbf{x} = \mathbf{b} \implies M^{-1}A\mathbf{x} = M^{-1}\mathbf{b} \implies \tilde{A}\mathbf{x} = \tilde{\mathbf{b}}.$$

Using an appropriate M should lead to fewer iterations in solving this new system with an iterative method.
There are two trivial (and useless) ways to choose the preconditioner: $M = I$ or $M = A$. In both cases we obtain the exact same system we started from: in the fist

case we have $I \cdot A\mathbf{x} = I \cdot \mathbf{b} \implies A\mathbf{x} = \mathbf{b}$ while in the second we get $A^{-1} \cdot A\mathbf{x} = A^{-1} \cdot \mathbf{b} \implies \mathbf{x} = A^{-1}\mathbf{b}$.

Some of the most common forms of preconditioner are:

- *Jacobi:* $M = D$. This preconditioning is also called diagonal, since the preconditioner is the diagonal of the matrix A;
- *Incomplete LU factorization:* $M = LU$. It exploits the LU decomposition of A (or something rather similar) for preconditioning;
- *Incomplete Cholesky factorization:* $M = LL^T$. When dealing with positive-definite matrices we can use the Cholesky decomposition (or, as for LU, something that it's close to it).

2.2.1 Preconditioning for CG

Preconditioning is very useful to boost the Conjugate Gradient method, the number of iterations indeed is proportional to the square root of the conditioning of A.
The CG however requires the matrix $\tilde{A}$ to be symmetric, which is usually not the case even if A and M are. We can avoid this problem by performing the preconditioning in a little different way.
Given a decomposition $M^{-1} = PP^T$ we can rewrite the preconditioned system as:

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \implies M^{-1}A\mathbf{x} = M^{-1}\mathbf{b} &\implies \\ PP^TA\mathbf{x} = PP^T\mathbf{b} &\implies \\ P^TA\mathbf{x} = P^T\mathbf{b} &\implies \\ P^TAPP^{-1}\mathbf{x} = P^T\mathbf{b} &\implies \\ \tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} & \end{aligned}$$

where $\tilde{A} = P^TAP$, $\tilde{\mathbf{x}} = P^{-1}\mathbf{x}$ and $\tilde{\mathbf{b}} = P^T\mathbf{b}$.
This way the matrix $\tilde{A}$ is symmetric as required by the CG. So, once computed the algorithm on the preconditioned system we can calculate $\mathbf{x} = P\tilde{\mathbf{x}}$.
The decomposition $M^{-1} = PP^T$ is quite trivial in all the kind of preconditioners cited above:

- *Jacobi:* $P = \sqrt{D}^{-T}$;
- *Incomplete LU factorization:* $P = U^T$;
- *Incomplete Cholesky factorization:* $P = L$

2.3 Laplacian Matrix

A Laplacian matrix L [7] is a special representation of a graph G. It's a square matrix of size $n \times n$ where n is the number of nodes of G. For simple graphs (undirected and unweighted without loops and multiple edges), on the diagonal is stored the degree of each node, while the off-diagonal entries $L(i, j)$ are set to -1 if the nodes i and j are connected through an edge.
L has some properties that can become useful.
If G is undirected then L is symmetric, indeed if there is an arc between i and j also the converse must be true. The sum of the elements of each row (or column since it's symmetric) is zero because each row contains the information about a node: its rank and a -1 for each incident edge, which summed up together returns 0. This property also implies trivially that L is diagonal dominant. Therefore, since the matrix is both symmetric and diagonal dominant it must be also positive semi-definite. In particular, there are as many null eigenvalues as the number of connected components, while the others are all strictly positive. We can easily check that the eigenvector that causes $L\mathbf{v} = 0$ is composed by all ones.
The discussion above remains valid also for weighted graphs [8], we just have to build L slightly differently: on the diagonal in this case we insert for each node the sum of all the weights w_{ij} of its incident arcs, the other elements $L(i, j)$ are set to $-w_{ij}$.

Chapter 3

Context Analysis

In this chapter we will talk about what the matrices involved represent, doing some considerations since the the matrix $ED^{-1}E^T$ is positive semi-definite (and not positive definite, as CG usually assumes) and its condition number is pretty high.

3.1 The data involved: an overview

First of all lets analyze the elements that form $ED^{-1}E^T\mathbf{x} = \mathbf{b}$.

E is a short-large matrix $n \times m$ with $n = |rows|$; $m = |columns|$.
E is a **node-edge matrix** describing a directed un-weighted graph. Specifically, given two nodes u and v and a direct edge *(u, v)*, by labelling u with an integer i, v with an integer j and the edge with an integer k, we have $E[i, k] = 1$ and $E[j, k] = -1$.
The matrix has a node_id for each row and an edge_id for each column.
Assuming the graph is just one connected component we know that the number of edges is bounded below by $n - 1$ (minimal graph). So, if we store the matrix in a rectangular array it has at least $O(nm) > O(n^2)$ elements. However we can store the matrix in a more efficient way, with a linked list for instance, because we know that it is sparse. Indeed each column has at most 2 elements different from 0, so the actual number of values that have to be saved is twice the number of edges, i.e. $O(m)$.

D is a diagonal matrix $m \times m$ which at the beginning of the interior points method stores the weights of the edges in the graph. Therefore, an arc saved in the column k of E, will have its weight in position $D[k, k]$.
D is positive semi-definite if the graph has only non-negative weights. Since we care about just the elements in the diagonal, we can store the matrix efficiently by saving only the diagonal entries and hence use only $O(m)$ elements.

The product EE^T takes the directed graph saved in E as a node-edge matrix and transforms it to the corresponding undirected version represented as a Laplacian matrix. $ED^{-1}E^T$ on the other hand is the weighted version.

b is a vector of length n. For each node indicates if it's either a source or a sink node. In particular, if the corresponding value s_i is positive, it means that there's an external flow of capacity s_i to that node, if it's negative the external flow s_i is from that node, otherwise if is 0 the flow in that node is only internal.
Since the sum of the incoming and outgoing flow must be zero it means that the sum of all elements of **b** must be 0.

Table 3.1 shows a snapshot of the features listed so far.

	E	D	$ED^{-1}E^T$	**b**
Type	Node-Edge Graph	Weights	Weighted Undirected Graph	Flows
Dimensions	$n \times m$	$m \times m$	$n \times n$	$n \times 1$
Structure	Large&Short	Diagonal	Laplacian	Array
Eigenvalues Signs	Indefinite	Positive semi-definite	Positive semi-definite	Indefinite
Distribution of values	Very sparse	Full Diagonal	Symmetric	$\sum_{i=0}^{n-1} b_i = 0$
Sparse Storage	$O(m)$	$O(m)$	$O(m)$	$O(n)$

Table 3.1: Summary of the features of the structures involved.

3.2 Preconditioned CG

It is known that the convergence of the Conjugate Gradient algorithm is related to the distance between the biggest and the smallest eigenvalue. A common way to improve the performances is by preconditioning the system before applying the CG.
The matrix $ED^{-1}E^T$ has a number of zero-eigenvalues equal to the number of connected components of the graph E. For the sake of simplicity we have considered just connected graphs and thus we have matrices A with exactly *one* eigenvalue equal to zero. The case concerning graphs with k multiple components however can be treated as k graphs with a single component, so this simplification does not restrict significantly the cases of study.
For this reason we compute the condition number of the matrix as λ_1/λ_{n-1}, where they

are respectively the biggest and the second smallest eigenvalue. This number may be very high, mostly caused by the coefficients in the matrix D^{-1}. If this condition number is reasonably big then it may be worth preconditioning the system. Of course, the term "big" isn't a-priori defined: we need to evaluate the CG before considering the preconditioning.
In our algorithm we run the CG on a preconditioned system of this form: $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ where: $\tilde{A} = P^{-T}AP^{-1}$, $\tilde{\mathbf{x}} = P\mathbf{x}$ and $\tilde{\mathbf{b}} = P^{-T}\mathbf{b}$ with P is our preconditioning matrix. We consider the **Jacobian Preconditioner**: we take as our matrix P a diagonal matrix where, on the diagonal, there is the square root ("element wise") of the diagonal of $\tilde{A}$.

3.3 CG on singular systems

Our problem deals with a singular system (an eigenvalue is zero) and, since the Conjugate Gradient it's mostly defined for positive definite matrices, we have more considerations to make.

$Im(A)$ doesn't span the whole space and if we pick a vector $\mathbf{b} \notin Im(A)$ the system $Ax = b$ has no solution, and it's known that the CG will diverge as a result of perturbation of domain errors. A common way to deal with this it's to delete some rows and columns (a number equal to the dimension of its null space), fix the corresponding entries of $\mathbf{x}$ and solve the reduced system. If $Im(A)$ is explicitly known, instead, it's possible to subtract to $\mathbf{b}$ its projection onto the null space of A, getting then a new vector $\mathbf{b_R}$. This second solution brings to a faster convergence rate than removing entries [10].
We know that $Im(A)$ is composed by the vectors which sum up to zero (A is a Laplacian matrix), so we can modify $\mathbf{b}$ to ensure that it belongs to the image of A: let $\underline{1} = ones(n) = [1, \ldots, 1]^T$, we define $\mathbf{b_R} = \mathbf{b} - (sum(\mathbf{b})/n) \cdot \underline{1}$, where $n = size(\mathbf{b})$ and $sum(\mathbf{v})$ is function that computes the sum of all the elements in a vector $\mathbf{v}$, the modified vector lying in $Im(A)$. In Matlab notation this can be written as `b = b - sum(b)/n`.
This assignment will produce a vector $\mathbf{b_R} \in Im(A)$. Indeed we can easily prove that $sum(\mathbf{b_R}) = 0$:

$$
\begin{aligned}
sum(\mathbf{b_R}) = sum(\mathbf{b} - sum(\mathbf{b}) \cdot \underline{1}/n) = 0 \implies \\
sum(\mathbf{b}) - sum(sum(\mathbf{b}) \cdot \underline{1}) \cdot 1/n = \\
sum(\mathbf{b}) - sum([sum(\mathbf{b}), \ldots, sum(\mathbf{b})]^T) \cdot 1/n = \\
sum(\mathbf{b}) - (n \cdot sum(\mathbf{b})) \cdot 1/n = \\
sum(\mathbf{b}) - sum(\mathbf{b}) = 0
\end{aligned}
$$

Regarding the preconditioned system, let $\tilde{\mathbf{b_R}}$ be the corrected and preconditioned vector $\mathbf{b}$: starting from $\tilde{\mathbf{x_0}} = 0$ we get a $\tilde{\mathbf{x}}$ which belongs to the $Im(\tilde{A})$ and which converges to the minimum norm solution of $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b_R}}$. Since the system is singular, we have infinite many solutions of the form $x + \alpha y$ where $y \in N(A)$ e.g. the vector of all ones and α is a scalar.
To get the corresponding minimum norm solution of $A\mathbf{x} = \mathbf{b}$ we need to subtract from $\mathbf{x} = P^{-1}\tilde{\mathbf{x}}$ its orthogonal projection onto the null space of A. More can be found in [10].

Note: *in our case of study we apply the CG to solve steps of a Flow Optimization Problem. Therefore, we know that* $\mathbf{b}$ *has already the property of having the elements summing up to zero since it describes the incoming/outgoing flow in the network.*
However, in our problem we don't receive such structured $\mathbf{b}$ *because we just have to implement the CG part.*
In order to simulate this situation and also assure that the algorithm converges we can start from a random vector $\mathbf{b}$ *and compute* $\mathbf{b_R}$ *as stated above.*

3.3.1 Modification to the CG algorithm

We are now going to introduce a modification to CG needed to ensure the correctness of the algorithm.

Let $\underline{\mathbf{1}}$ be the vector of all ones defined as above. For each vector $\mathbf{v} \in Im(A)$ we know that the sum of its elements must be zero. We can express this property as $sum(\mathbf{v}) = \underline{\mathbf{1}}^T \cdot \mathbf{v} = 0$.
For a preconditioned system $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b_R}}$ where $\tilde{A} = P^{-T}AP^{-1}$ this is not true: we have to modify slightly the property by inserting some factors:

$$
\begin{aligned}
\underline{\mathbf{1}}^{\mathbf{T}} \cdot P^T P^{-T} A P^{-1} \cdot \mathbf{v} = \underline{\mathbf{1}}^{\mathbf{T}} \cdot A P^{-1} \cdot \mathbf{v} \implies \\
\underline{\mathbf{0}} \cdot P^{-1} \cdot \mathbf{v} = 0 \qquad \forall \mathbf{v}
\end{aligned}
$$

We define $\mathbf{c} = \underline{\mathbf{1}}^{\mathbf{T}} \cdot P^T$, where $\mathbf{c}$ can be though as *correction* vector. This is the same for both preconditioned and not preconditioned systems if we consider $P = I$ for the

latter case.
At each step of CG we update $\mathbf{r}$ by using vectors from the Krylov Subspaces we generate. Since span($\mathbf{r_s}$) = span($\mathbf{b_s}$) we need to ensure $\mathbf{r}$ to lie in $Im(\tilde{A})$ for the entire run (the same holds for the vector $\mathbf{d}$). This is true at the very first step of the algorithm since $\mathbf{r} = \mathbf{b_R}$, but we might lose that property due to numerical errors: we need to ensure our $\mathbf{r_s}$ satisfy the condition of belonging to the $Im(\tilde{A})$ by subtracting at each step the orthogonal projection of $\mathbf{r}$ onto the null space of $\tilde{A}$ (equal to A for $P = I$):

$$\mathbf{rc} = \mathbf{r} - \frac{(\mathbf{c^T} \cdot \mathbf{r}) \cdot \underline{\mathbf{1}}}{s} \tag{3.1}$$

In Matlab notation would be `rc = r - (c'r)/s`, where **rc** stands for "corrected" **r** and s is an unknown scalar.
We need now to compute the value of s to guarantee $\mathbf{c^T} \cdot \mathbf{rc} = 0$:

$$\begin{gathered}
\mathbf{c^T} \cdot \mathbf{rc} = \mathbf{c^T}(\mathbf{r} - (\mathbf{c^T} \cdot \mathbf{r}) \cdot \underline{\mathbf{1}})/s = 0 \implies \\
\mathbf{c^T} \cdot \mathbf{r} - (\mathbf{c^T} \cdot \mathbf{r}) \cdot (\mathbf{c^T} \cdot \underline{\mathbf{1}})/s = 0 \implies \\
\text{true if } (\mathbf{c^T} \cdot \underline{\mathbf{1}})/s = 1 \implies \\
s = \mathbf{c^T} \cdot \underline{\mathbf{1}} = sum(\mathbf{c})
\end{gathered}$$

For the non-preconditioned case we have $\mathbf{c} = \underline{\mathbf{1}}^{\mathbf{T}}$ and so we get $sum(\mathbf{c}) = n = size(\mathbf{b}) = size(\mathbf{r})$, i.e. the correction mentioned in the previous section.

Chapter 4

Implementation

In this chapter we show the implementation of the CG for singular systems with some charts to evaluate how good the presented solution is. We used the following tools:

- **Matlab** to write the algorithm and the test scripts;
- **Netgen**[1], **Complete**[2] and **Goto**[3] that are graph generators used to create connected graphs, all written in C and available here;
- **Python** to create files storing our data (E, D and $\mathbf{b}$) readable by Matlab and to plot the charts;
- **Excel** to store the results obtained in the experiments.

4.1 The algorithm

Algorithm 2 shows the Conjugate Gradient algorithm with the considerations listed in chapter 3. We pass directly in input the vector $\mathbf{b_R}$ and the correction vector $\mathbf{c} = \underline{\mathbf{1}}^{\mathbf{T}} P^T$ and the instruction in bold identify the corrections.
In case we apply preconditioning the vector $\tilde{\mathbf{b}}_{\mathbf{R}}$ is passed already preconditioned and the preconditioned matrix $\tilde{A}$ is implicit inside the lambda function **mm**.
Moreover, if the resulted vector x is the solution of the preconditioned system, as we stated in chapter 3, we should perform the operation $\mathbf{x} = P \backslash \tilde{\mathbf{x}}$.
Finally, we use the usual 10^{-6} as stopping criterion *eps*.

[1] Kilngman, Napier and Stutz
[2] A. Frangioni, C. Gentile
[3] A. Goldberg

Input: `mm`: lambda function for matrix multiplication
`b`: known vector already in Im(A)
`n`: matrix dimension
`eps`: precision stopping criterion
`c`: correction vector
Result: `x`: one of the solutions of the system Ax = b

———————————————————

$d_0, r_0 = b$;
$x_0 = \underline{\mathbf{0}}$;
$k = 1$;
while $k < n \vee norm(r_{k-1})/norm(b) > eps$ **do**
  $\mathrm{Ad} = \mathrm{mm}(\mathrm{d}_{k-1})$;
  $\alpha = (r_{k-1}^T \cdot r_{k-1})/(d_{k-1}^T \cdot Ad)$;
  $x_k = x_{k-1} + \alpha \cdot d_{k-1}$;
  $r_k = r_{k-1} - \alpha \cdot Ad$;
  $\mathbf{r}_k$ = **r_k - (c' · r_k) / sum(c)**;
  $\beta_k = (r_k^T \cdot r_k)/(r_{k-1}^T \cdot r_{k-1})$;
  $d_k = r_k + \beta_k \cdot d_{k-1}$;
  $\mathbf{d}_k$ = **d_k - (c' · d_k) / sum(c)**;
  $k = k + 1$;
end

Algorithm 2: Code of the CG.

4.2 The Data

We have 3 **families** of graphs, one for each generator.
For each family we generated 8 **groups** of graphs: with approximately 1000, 2000, 3000, 8000, 16000, 128000, 512000 and 1000000 arcs with each group having 10 instances.

Note: *Each graph generator has a different way of creating the graphs:*

- ***Netgen*** *takes as input the number of arcs and calculates the number of nodes based on a required density (25%, 50% or 75%). We computed the instances with all three of them;*
- ***Complete*** *does the opposite, i.e., it takes as input the number of nodes and a value representing the average number of parallel arcs from a node to another. We generated the instances computing by hand the number of nodes needed to approximately reach a number of edges with the same order of magnitude of the ones generated by Netgen;*
- ***Goto*** *on the other hand lets both the number of edges m and of nodes n as input as long as the property $6 \cdot n \leq m \leq n^{5/3}$ is respected. We fixed the number of edges and we generated the nodes uniformly in the whole range available.*

In Table 4.1 is reported a recap for each generator of the interval of nodes and of edges created. We can already notice that *complete* generates very dense graphs, that the last instances of *goto* are very sparse and *netgen* is something in between. We will see how this difference will change the performance of the algorithm.

Netgen								
edges	1000	2000	3000	8000	16000	128000	512000	1000000
nodes	52 - 89	73 - 126	89 - 155	146 - 253	207 - 358	584 - 1012	1168 - 2024	1633 - 2828
Complete								
edges	1061 - 1120	1995 - 2136	3008 - 3080	8832 - 9025	17679 - 17989	141231 - 142029	718266 - 720155	1402293 - 1405683
nodes	30	41	50	85	120	337	759	1060
Goto								
edges	1000	2000	3000	8000	16000	128000	512000	1000000
nodes	65 - 165	100 - 330	125 - 495	220 - 1330	335 - 2660	1160 - 21330	2670 - 85330	3990 - 166660

Table 4.1: Recap of the number of edges and of nodes for each generator.

Moreover, we created 3 different matrices D to perform experiments on:

1. D equal to I;
2. D with 75% of the elements of the order of 10^2 and the rest of the order of 10^{-2};
3. D with 25% of the elements of the order of 10^6 and the rest of the order of 10^{-6}.

These 3 "distributions" allow us to simulate the different calls of the CG during the IP method. As the algorithm proceeds, the edges where the flow will go through will increase their weights while on the others it will decrease. The case with $D = I$ is a special case and it's our baseline.

4.3 Collected Results

We have written a spreadsheet file for each composition D-graph_family, with a sheet for each group, each sheet having these data:

- **ID**: id of the graph;
- **nEdge**: number of edges;

- **nNodes**: number of nodes;
- **Cond**: condition number of the matrix computed as ratio between the biggest and the smallest non-zero eigenvalue;
- **||Ax-b||/||b||**: the norm telling us how far we are from the solution;
- **nIter**: the number of iterations needed by CG;
- **CGTime**: time needed by CG;
- **PrecOverhead**: overhead introduced by preconditioning, outside CG;
- **Total**: total time spent;
- **PcgIter**: number of iterations done by Matlab's pcg algorithm;
- **PcgTime**: time spent by preconditioning and by Matlab's pcg.

In the next three subsections we will summarize the results collected. In particular, the parameters we will use for comparison are: the condition number, the number of iterations and the computational time.
We decided to report the results only of graphs with 16000 and 512000 edges as they are good indicators for small and big graphs of the values obtained. Nevertheless, the experiments were carried out on all graphs dimensions that we described in chapter 4.2 and, if the reader is interested, can be found here. The folder "Spreadsheet" contains all the Excel files generated and "Charts" the plots.

In table 4.2 are reported the results of the condition numbers: for each experiment and each dimension we calculated the mean of the ten instances. We computed this calculation both for the original matrix and the preconditioned one in order to compare the two values. We were not able to compute the condition number of all the graphs *goto* with 512000 edges. Indeed, the last instances produce matrices A so heavy that our computers are not able to compute the eigenvalues. However, based on the other smaller groups we could assume these values to have the same pattern.
This is just a coarse view of how the preconditioner effects the system. As we know, the conjugate gradient method doesn't depend directly from the condition number but on the disposition of the eigenvalues.

Regarding the number of iterations and the computational time we thought it would be clearer to show the results through some charts in each subsection. Figure 4.1 shows results regarding graphs of size 16K edges, while figure 4.2 results for graphs of

Complete						
	I		**75%**		**25%**	
	K(A)	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**
16K	1.3045	1.170	143.100	40.869	94.957	37.324
512K	1.120	1.066	302.252	134.683	839.469	178.895
Netgen						
	I		**75%**		**25%**	
	K(A)	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**
16K	46.092	1.401	5732.423	67.354	2,430e+07	230.002
512K	186.318	1.236	3,603e+05	570.308	2,884e+08	591.722
Goto						
	I		**75%**		**25%**	
	K(A)	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**	**K(A)**	**K($\tilde{A}$)**
16K	1272.759	68.963	3,149e+10	1269.216	3,741e+08	3744.293
512K	-	-	-	-	-	-

Table 4.2: Mean of the condition number of a subset of our dataset.

size 542K edges.
In a box below a little summary to help the reading.

Plot's composition summary:

- ***1st col:*** *D=I;* ***2nd col:*** *D = 75%;* ***3rd col:*** *D = 25%;*
- ***1st row:*** *nIter netgen and complete;* ***2nd row:*** *nIter goto;* ***3rd, 4th and 5th rows:*** *respectively time spent netgen, complete and goto;*
- ***red color:*** *netgen graph;* ***blue color:*** *complete graph;* ***green color:*** *goto graph;*
- ***straight line:*** *not preconditioned;* ***dashed line:*** *preconditioned.*

Furthermore, for the computational time of the preconditioned system we thought it would be interesting to show also the decomposition of the total time in the actual time spent computing the CG (darker area) and the overhead introduced in order to perform the preconditioning (lighter area).

4.3.1 D = I

The first experiment we tried is the simplest one: when D is the identity matrix.
As we can see in table 4.2 the condition number of A for *complete* graphs is already

very low even before preconditioning. This is predictable since the condition number of an actually complete connected graph is 1, indeed, the matrix will have all the diagonal elements with value $n - 1$, as each node is connected to all, and the other entries equal to -1. However, as we said before, the *complete* generator also adds some parallel edges that increase the condition number a bit.
Netgen generates less dense graphs and therefore the condition number is bigger even though it's not too high. By applying the preconditioning the condition number decreases to values slightly bigger than 1, which is good.
Goto presents even sparser graphs and a higher condition number, the preconditioning reduces it but it's not able to produce values close to 1 as with the other generators.
In figure 4.1 and 4.2 we can see how the application of a preconditioner affects the number of iterations and the computational time. The number of iterations decrease in all cases, while the computational time sees an improvement only for *netgen*. *Complete* and *goto* graphs have less CG time but the improvement is not enough to balance the overhead introduced.

4.3.2 D 75% dense

Compared to the case of $D = I$, we notice that the decrease in the condition number is more evident for all generators.
In spite of this, the improvement obtained for *complete* is even less. In multiple instances the number of iterations of the preconditioned system is higher. As for the computational time, in this case not even the CG time seems to decrease. So even assuming that the preconditioning comes for free is not convenient to use.
The case for *netgen* and *goto* is different, both the number of iterations and the computational time gain a lot from preconditioning. The peaks of some instances are smoothed and the overhead is less significant compared to the CG time.
For the instances of *goto*, we can observe even more clearly that as the ratio between nodes and edges decrease both the number of iterations and the computational time get worse.

4.3.3 D 25% dense

In this case there is no doubt that the condition number benefits from preconditioning and the results for the number of iterations and the total time confirms the observations made before.
For *netgen* there's a sensible improvement in both the parameters and it's clear that the use of preconditioning is particularly useful for big and ill-conditioned matrices.
The last 4 instances of *goto* without preconditioning does not converge, as we can see

both in figure 4.1 and 4.2. Preconditioning prevents this from happening and moreover ensures that the number of iterations and the computational time stays almost idle for all 10 instances.
Again, *complete* charts prove that even for bigger graphs and with worse condition number the preconditioning does not lead to better performances.

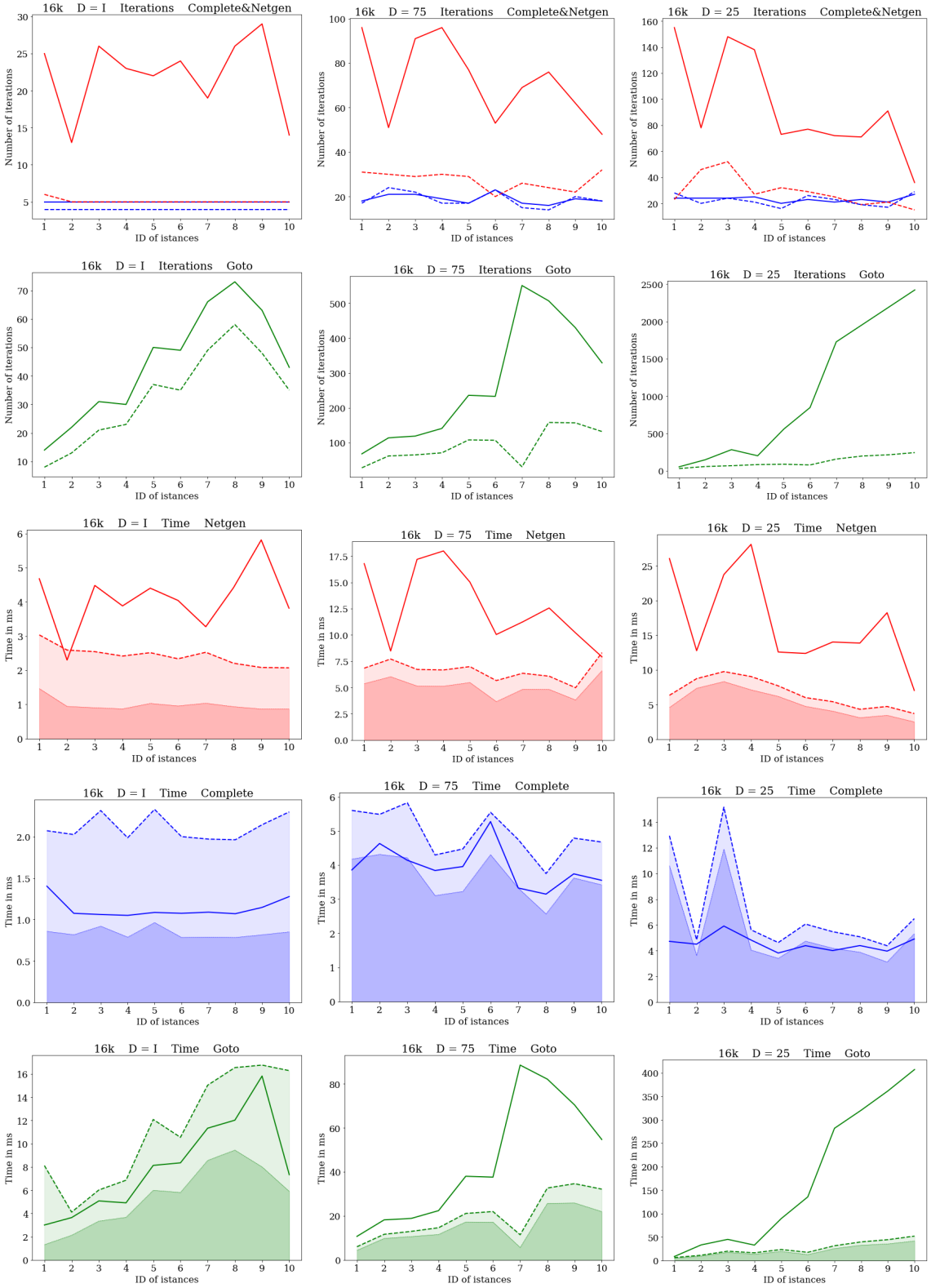

Figure 4.1: Results obtained for the graphs with 16000 edges.

Figure 4.2: Results obtained for the graphs with 512000 edges.

4.4 Final considerations

From this dataset we conclude that the Jacobian preconditioner is a simple and efficient way to improve performances specially if the input graphs aren't dense, otherwise Conjugate Gradient works efficiently by itself. The density of the graphs could be a first hint to decide whenever applying preconditioning will be worth it or not.
The results respect the convergence rate of a Krylov subspace method, which depends on the number of clusters of the eigenvalues.
We experimented also with the Cholesky preconditioner but, while the CG performs really good in terms of number of iterations (even better than the Jacobian), the overhead introduced is so high that the time spent before beginning CG isn't worth the usage of this technique. Of course, auxiliary architectures such as a dedicated GPU would improve the speed of computing the Cholesky factorization as well the matrix multiplication inside CG.

4.5 Comparison with Matlab's pcg

Finally, we made a comparison with the "greatest competitor" of our algorithm: the Matlab function pcg(). By executing that function on the same dataset we noticed that the performances are equivalent: they take almost exactly the same amount of iterations performing the conjugate gradient, but Matlab's pcg may take few milliseconds more w.r.t. to our version. Figure 4.3 shows the differences between the 2 versions on the graphs of 16K edges.

Plot's composition summary:

- ***1st col:*** *D=I;* ***2nd col:*** *D = 75%;* ***3rd col:*** *D = 25%;*
- ***red colors:*** *netgen with our CG;* ***blue colors:*** *complete with our CG;* ***dark green color:*** *goto with our CG;*
- ***orange colors:*** *netgen pcg;* ***cyan colors:*** *complete pcg;* ***light green color:*** *goto pcg;*
- ***straight line:*** *not preconditioned;* ***dashed line:*** *preconditioned.*

We can conclude therefore that our algorithm is computationally equivalent to Matlab's. We skipped the comparison between our CG and Matlab's pcg for graphs of size 512K because the time for the execution is way too high, but we expect to have comparable times as well.

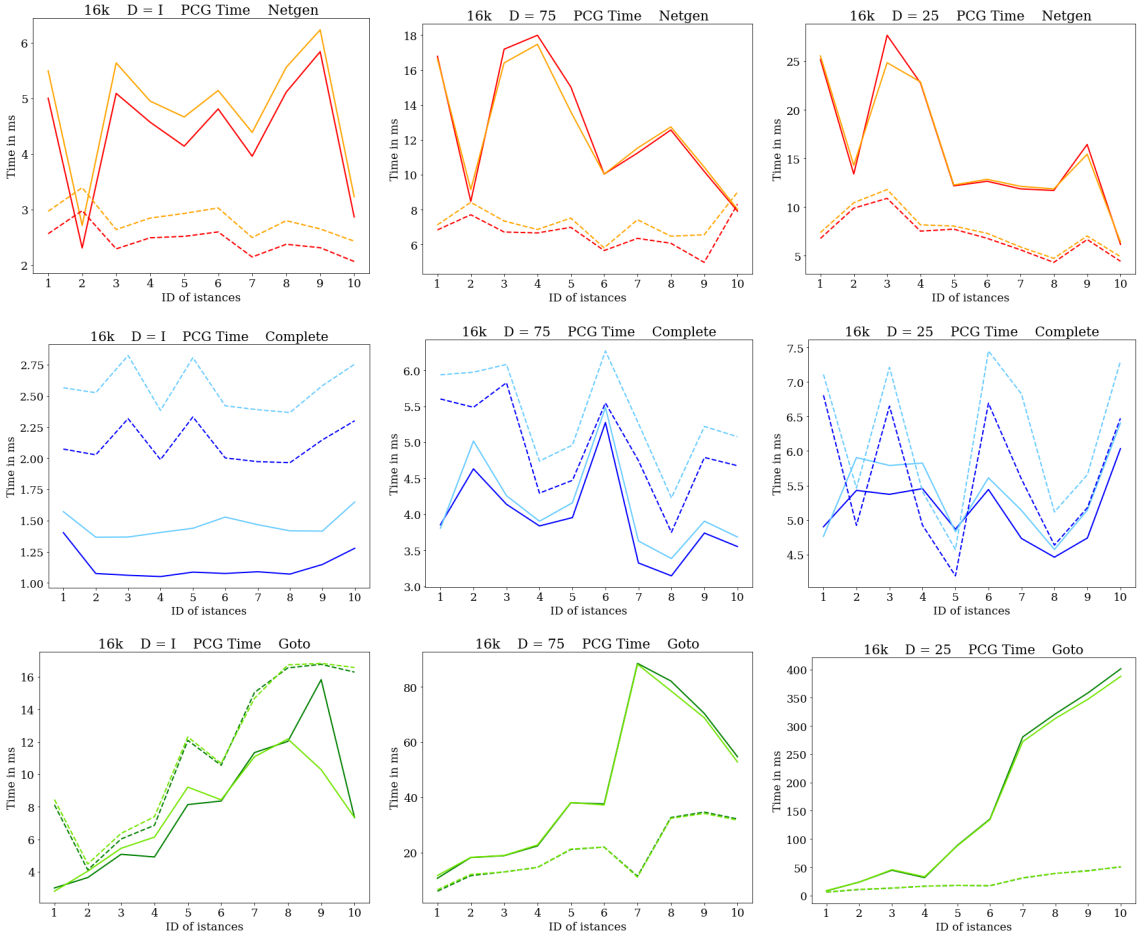

Figure 4.3: Comparison of computational time between our implementation and Matlab's.

Bibliography

[1] L. N. Trefethen, David Bau, *Numerical Linear Algebra*

[2] R. K. Ahuja; T. L. Magnanti & J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993

[3] S. Wright, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, 1997

[4] W. Wang, *The Convergence of and Interior Point Method using modified search directions in final iterations*
https://doi.org/10.1016/S0898-1221(02)00153-0

[5] D. P. Bertsekas, *Nonlinear Programming*, 1995

[6] Greg Fasshauer, *Lectures on Numerical Linear Algebra* http://www.math.iit.edu/~fass/477577_Chapter_16.pdf

[7] Fan R. K. Chung, *CBMS: Lecture Notes on Spectral Graph Theory*
http://www.math.ucsd.edu/~fan/research/cb/ch1.pdf

[8] Dan Spielman, *Lectures on Spectral Graph Theory*
http://www.cs.yale.edu/homes/spielman/561/2009/lect02-09.pdf

[9] Jonathan Richard Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*
https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

[10] E. F. Kaasschieter, *Preconditioned Conjugate Gradient for solving singular systems*
https://www.sciencedirect.com/science/article/pii/0377042788903585