UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES
Treball final de grau

Resolution of general linear systems: numerical methods and applications

Autor: Bertran Miquel Oliver

Director: Dr. Antoni Benseny Ardiaca
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 20 de juny de 2019

Abstract

The *Singular Value Decomposition* and the *complete orthogonal decomposition* algorithms are two numerical methods based on orthogonal matrix transformations. Among other usages, both are used for least square approximately solving general linear systems. Not all the systems have a unique solution, but all the possible solutions can be found. Besides, in this project, these two methods are going to be compared each other and then see how apply them to nonlinear systems in a geodesic scope.

2010 Mathematics Subject Classification. 15A03, 15A06, 15A18, 15A24, 15B10, 65F05, 65H10, 65K05, 65Y15.

Acknowledgements

I would like to thank my tutor, Dr. Antoni Benseny, for all his patience, his interest in this work and all the advices that he has provided to me. I would also like to thank my parents and friends for their general support through this journey, specially to Marc for solving all my doubts about LaTeX . Finally, I want to thank to Sandra for encouraging me in the most difficult moments and be always willing to advise me.

Contents

Introduction

Motivation

During my Bachelor degree in Mathematics, I have studied different courses related to the resolution of linear systems. In the Linear Algebra course, we deal with the problem of solving linear systems in a theoretical way. Moreover, coursing the first part of Numerical Methods provided to me different tools for solving compatible determined linear systems. While the second part gave to me methods for solving compatible undetermined linear systems, such as the *QR* factorization method. In addition, coursing subjects as Graphs, Scientific Programming or Algorithms brought me interest to the programming scope and also to study the efficiency of algorithms.

Objectives

With these motivations in mind and the proposal of Dr. Benseny, we decided to focus this project on three main goals.

Following the Jordi Jover project, see [8], the first objective consisted in study different efficient methods to approximately solve general linear systems without increasing the condition number, not only using the techniques learned during the degree, but also extending to other methods. To do this, we focused on two methods: the *Singular Value Decomposition* and the *complete orthogonal decomposition* methods.

Moreover, the second goal consisted in applying the mentioned methods to some random linear systems by using a software. To do so, we had to understand and adapt the software used in project [8], in order to compare the studied methods.

Finally, the third goal was about making an application of solving non linear systems to a real case: adjusting the position of geodesic vertices in a simplification of the geodetic network.

Memoir structure

This work is organised in five chapters. The first three chapters provides two different resolutions of general linear systems methods. To do so, the first chapter introduces the basic concepts that will be applied in following chapters.

In the second chapter, we will apply the preliminary concepts to explain the *singular value decomposition* algorithm and the *complete orthogonal decomposition* algorithm. By doing it, we will use the *bidiagonalization* and the *QR* iteration methods.

Then, the main goal of the third chapter is to approximate a solution for general linear systems. By using the *normal equations* and applying the two decomposition algorithms explained in the last chapter, we are going to provide not only the least squares solutions, but all of the possible approximated solutions of a general linear system.

Subsequently, in the chapter four, the software, which approximate the solution of any linear systems by the SVD and the complete orthogonal decomposition methods, is explained in detail. Furthermore, a comparison between the two methods is done. Also, the SVD algorithm of the software is compared to a function of the basic package of R.

In the last chapter, a geodesy application is done by using the SVD method. To do so, by using graph concepts, the problem and the solution of it will be represented.

Chapter 1

Preliminaries

The aim of this chapter is to provide a brief overview of the orthogonal matrices and present some of their characteristics. We will also make emphasis in two of them: the Givens and the Householder matrices.

1.1 Orthogonal matrices

Definition 1.1. A matrix $A \in \mathbb{R}^{nxn}$ is an *orthogonal matrix* if its columns and rows are orthogonal unit vectors, i.e.

$$QQ^t = Q^tQ = I ,$$

where I is the identity matrix.

This definition implies two direct consequences.

Observation 1.2. The columns of an orthogonal matrix form an orthonormal basis.

Observation 1.3. The inverse of an orthogonal matrix is its transpose

$$Q^{-1} = Q^T .$$

The second observation is helpful when solving linear systems, since we can always find the inverse of an orthogonal matrix in an efficient way. Apart from these observations, there are also two additional properties of this sort of matrices.

Properties 1.4. Let $Q \in \mathbb{R}^{nxn}$ be an orthogonal matrix

1. $\det(Q) = \pm 1$.

   Proof. $1 = \det(I) = \det(Q^T Q) = \det(Q^T) \cdot \det(Q) = \det(Q)^2 \Rightarrow \det(Q) = \pm 1$. □

2. $\| Q \| = 1$.

   Proof. All rows of the matrix have norm equal to one, because of the definition of orthonormal matrix. □

Observation 1.5. The converse of the first property is not true: having a determinant of a matrix equals to ± 1, does not guarantee the matrix to be orthogonal.

The second property is very useful because of its relation with the condition number of a matrix. In the next section, we will introduce the concept of condition number and its relation with the linear systems.

1.2 Condition number

Firstly, we will introduce the definition and give an interpretation of it. After that, we will make some observations of this concept to see the importance of orthogonal matrices.

Definition 1.6. Given a linear equation $Ax = b$, where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, the *condition number* gives a bound on how inaccurate the solution x will be after the approximation. The *condition number* is defined by

$$\kappa(A) = \| A \| \| A^{-1} \| .$$

If a matrix has $\kappa(A) \approx 1$, it is said to be a *well-conditioned* matrix. While, if it is a high number, *ill-conditioned*.

Furthermore, the condition number gives information about how a variation of A or b alters the solution vector x. In other words, $\kappa(A)$ is the rate at which the solution will change with respect to a change in A or b.

Now, let us introduce some observations about the condition number.

Observation 1.7. 1. $\kappa(A) \geq 1$.

Proof.

$$\kappa(A) = \| A \| \| A^{-1} \| \geq \| AA^{-1} \| = \| I \| = 1.$$

□

2. We can calculate the condition number if, and only if, the matrix is regular, as we need his inverse to compute it. So, it can be computed for any orthogonal matrix.

3. An orthogonal matrix has $\kappa(Q) = 1$. Consequently, the product of an orthogonal matrix with any matrix does not modify its the condition number.

   Proof. Let $A \in \mathbb{R}^{m \times n}$ be an invertible matrix and $Q \in \mathbb{R}^{mxm}$ an orthogonal matrix

   $$\kappa(QA) = \| QA \| \| (QA)^{-1} \| = \| A \| \| A^{-1}Q^T \| = \| A \| \| A^{-1} \| = \kappa(A) \ .$$

   Remark that $\| QA \| = \| A \|$. □

As a consequence of the last observation, if we apply an orthogonal matrix to solve a linear systems, the resultant matrix will have the same condition as the initial matrix. For example, in the QR method, when we product the orthogonal matrix Q with the initial matrix A, it results a matrix R which $\kappa(R) = \kappa(A)$. On the whole, applying an orthogonal matrix to a system, does not increase the condition number, i.e. does not make the linear system worse. For these reason, we will focus on those methods that use orthogonal matrices to solve linear systems.

For solving linear systems $Ax = b$, we will use matrices that diagonalize or, at least, make the matrix A triangular. To do it, we need to zero some elements of the initial matrix. In the next section, we will see two different sort of orthogonal matrices: the *Givens rotations*, to introduce a zero in a component of the matrix; and the *Householder reflections*, used to transform a vector to a multiple one of the $e_1 = (1, 0, \ldots, 0)$, in one step.

1.3 Givens rotations

When diagonalizing matrices, we usually want to zero a specific component. In order to achieve it, we are going to apply rotation matrices.

Definition 1.8. A *rotation matrix* is an orthogonal matrix of the form

$$G(i,k,\theta) = \begin{array}{c} \left[\begin{array}{ccccccc} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & \text{-}s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{array}\right] \begin{array}{c} \\ \\ i \\ \\ k \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & i & & k & & \end{array} \end{array}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ.

If we multiply $G(i,k,\theta)^T$ on the left side of another matrix, amounts to a counterclockwise rotation of θ radians in the (i,k) coordinate plane. We can see it if we take a column vector $x \in \mathbb{R}^n$ and $y = G(i,k,\theta)^T x$, then

$$y_j = \begin{cases} cx_i - sx_k \ , & j = i \\ sx_i + cx_k \ , & j = k \\ x_j \ , & j \neq i,k \end{cases}$$

Our goal is to zero some elements. We can force y_k by using

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \ , \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}} \ .$$

Definition 1.9. A *Givens rotation* is a rotation matrix $G(i,k,\theta) \in \mathbb{R}^{nxn}$ where given a vector $x \in \mathbb{R}^n$,

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \ , \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}} \ , \qquad \text{where } i,k < n. \tag{1.1}$$

Observation 1.10. Given a vector $x \in \mathbb{R}^n$, applying n-1 times a sequence of Givens rotations, we transform it to a parallel vector of e_1.

Applying Givens rotations

Having the definition of a Givens rotation in mind, let us see how to apply it. First, we will see a better form to calculate (1.1). Secondly, we will provide an efficient algorithm to apply a Givens rotation.

In the next algorithm, we are going to calculate c and s to zero a particular element. This algorithm is better than (1.1) because it guards against overflow. Given two scalar $a,\ b\ \in \mathbb{R}$, this function compute $c = \cos(\theta)$ and $s = \sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ \text{-}s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

consequently, we obtain the following conditions

$$\begin{cases} ca - sb = r \\ sa + cb = 0 \end{cases} \tag{1.2}$$

Another implicit condition is $c^2 + s^2 = 1$, because of the trigonometric reason.

If $b = 0$, then $c = \pm 1,\ s = 0$. Consequently, $a = \pm r$. Otherwise, if $a = 0$, then $c = 0,\ s = \pm 1$. Now, $b = \mp r$. Else, we set r in function of the module of a and b.

- If $|a| \geq |b|$, then, we set $r = \text{-}\frac{a}{b}$. As a result of (1.2)

  $$c = -\frac{a}{b}\ s = \frac{s}{r} \ \Rightarrow\ s = cr\ .$$

  By the trigonometric rule and replacing, we obtain

  $$c = \frac{1}{\sqrt{1+r^2}}\ .$$

- Otherwise, $|a| < |b|$, then, we choose $r = \text{-}\frac{b}{a}$. Analogously, we obtain

  $$c = sr\ , \qquad\qquad s = \frac{1}{\sqrt{1+r^2}}\ .$$

Observation 1.11. In fact, to set the Givens rotation, the value of θ is not needed, as we can see in the calculus of above. In consequence, we can say that $G(i,k,\theta)$ is in function of c and s, also of θ, but just in an implicit form. So it is not necessary to work with trigonometric functions.

Now, let us see how to multiply efficiently a matrix with a Givens rotation. Let $A \in \mathbb{R}^{m\times n}$, $c = \cos(\theta)$ and $s = \sin(\theta)$. If $G(i,k,\theta) \in \mathbb{R}^{mxm}$, then $G(i,k,\theta)^T A$ affects just two rows of A, the *ith* and the *kth* row.

Analogously, multiplying a Givens matrix by the left, affects just two columns of A, with a similar program.

1.4 Householder reflections

Definition 1.12. Let $v \in \mathbb{R}^n$ be a nonzero column vector. An $n \times n$ matrix P of the form

$$P(v) \equiv I - \frac{2}{v^T v} v v^T$$

is called a *Householder reflection*. The vector v is called a *Householder vector*.

Properties 1.13. This matrices satisfy:

- $P(cv) = P(v)$, if $c \neq 0$. We can say Householder matrices are associated to different directions or one-dimensional subspaces.
- $P(v)$ is symmetric: $P(v)^T = P(v)$.
- $P(v)$ is orthogonal: $P(v)P(v) = I$.
- $P(u)v = u$, $\forall v \in u^\perp$. As a result, $P(v)v = -v$, so $P(v)$ represents a reflection with respect to the hyperplane $v^\perp$. Therefore, $x \in \mathbb{R}^n$ can be splited in

  $$x = cu + v \;,$$

  with $c = \frac{u^T x}{u^T u}$, $v = x - cu^\perp$, and $P(u)x = -cu + v$.
- As a consequence of the last item, $det\ P(v) = -1$.

This are some of the properties of Householder matrices. With Householder reflections, we can find a Householder vector such that for a vector $x \in \mathbb{R}^n$, $P(v)x = \pm\|x\|_2\, e_1$. See [2].

We can apply the product by the left and obtain a row vector multiple of e_1^T .

Applying Householder reflections

Let us see now which is the most efficient way to apply this sort of matrices. Actually, any Householder matrix will be

$$\left[\begin{array}{c|c} I & 0 \\ \hline 0 & P \end{array}\right] .$$

The dimension of the identity matrix I can be modified in function of the elements that we want to change of the original matrix. Furthermore, the dimension of the identity matrix could be zero.

Applying a Householder matrix to another matrix in an efficient way, means just multiplying the rows or columns that need to be modified. As a consequence, we are going to multiply just the P matrix to the corresponding rows and columns of the original matrix.

Chapter 2

Decomposition methods

In this chapter, we will see how by applying Householder and Givens matrices, we can transform any matrix to a diagonal or upper triangular matrix. To achieve it, we will go through different previous steps: the bidiagonalization and the QR iteration methods. Finally, we are going to expose the *singular value decomposition* algorithm. In addition, we will see the *complete orthogonal decomposition* algorithm.

2.1 Bidiagonalization

In this section, we will see how, by applying orthogonal matrices, any matrix is transformed to a bidiagonal one. Also, we will justify why the diagonal matrix can not be formed directly.

Suppose $A \in \mathbb{R}^{m\times n}$ and $m \geq n$. There exist two orthogonal matrices U_B, V_B such that

$$B := U_B^T A V_B = \left[\begin{array}{ccccc} d_1 & f_1 & 0 & \dots & 0 \\ 0 & d_2 & f_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & & d_{n-1} & f_{n-1} \\ 0 & \dots & & 0 & d_n \\ \hline \multicolumn{5}{c}{\mathbf{0}} \end{array}\right],$$

where B is a bidiagonal matrix.

We have shown in 1.4 that using Householder matrix, with the correct Householder vector, it transforms any vector to a e_1 multiple vector[1]. In fact, U_B and V_B are a product of Householder reflections: $U_B = U_1 \cdots U_n$ and $V_B = V_1 \cdots V_{n-2}$, where $U_B \in \mathbb{R}^{mxm}$ and $V_B \in \mathbb{R}^{nxn}$, which transforms the columns and the rows of A to multiplies of e_1, respectively. Let us illustrate how U_i and V_j matrices affect to a $\mathbb{R}^{5x4}$ matrix, when are applied.

$$\begin{bmatrix} [\times] & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} \times & [\times] & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{V_1}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & [\times] & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{U_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & [\times] & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{V_2}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & [\times] & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & [\times] \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{U_4} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

where $[\times]$ is the initial element of the vector that will be multiple of e_1, as a column once applied U_i, and as a row when V_i does.

However, why can we not directly diagonalize the matrix? When we apply the V_i matrices, if take as $[\times]$ a diagonal element (i,i), then the subdiagonal element $(i+1,i)$, previous zeroed by the U_j matrix, is going to be altered. Even so, if we now apply a U_k matrix to zero the $(i+1,i)$ element, the $(i,i+1)$ element, previous zeroed by the V_i matrix, is going to be altered and therefore we will not achieve the diagonal matrix.

the rows are zeroed, if we choose the element (i,i) as the first element of the vector that will be modified, then, when we apply the V_i, the element $(i+1,i)$ will be altered. As a consequence, we will obtain a bidiagonal matrix.

[1]Form now on, when we talk about the e_1 vector, we assume that it has the corresponding size which will be understood by the context.

Special case

Now, suppose $m < n$. Doing the same process and so applying the correspondents Householder matrices, we will arrive at the next scenario:

$$\begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \end{bmatrix}.$$

As the element $(m, m+1)$ is not necessary zero, the resultant matrix not bidiagonal. In order to achieve a bidiagonal one, we should zero this element applying Givens rotation by the right. By doing it,

$$\begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \end{bmatrix} \xrightarrow{G_1} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \end{bmatrix} \xrightarrow{G_2}$$

$$\begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \end{bmatrix} \xrightarrow{G_3} \begin{bmatrix} \times & \times & 0 & 0 & \times & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \end{bmatrix} \xrightarrow{G_4}$$

$$\begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \end{bmatrix},$$

the element goes up and finally turn it zero. In fact, the G_i matrices are Givens rotations in the coordinate plans $(i, m+1)$, which i is the number of row in each case.

2.2 The QR iteration method on tridiagonal matrices

Once we have the bidiagonalization done, by making the product $T := B^T B$, where B is the bidiagonal matrix, we obtain a symmetric tridiagonal matrix. We can diagonalize this matrix with the QR iterative method. There are three main characteristics about the QR iteration applied to the tridiagonal matrices.

1. *Preservation of form.* If we apply the QR factorization to a symmetric tridiag-

onal matrix $T = QR$. Then,

$$T_+ = Q^T T Q = Q^T (QR) Q = RQ$$

is also symmetric and tridiagonal. By repeating several times this algorithm, the resultant matrix tends to a diagonal matrix.

2. *Shifts.* In order to increase the speed of convergence of the algorithm of 1, we redefine the algorithm, but adding a shift $\nu \in \mathbb{R}$, such that $T - \nu I = QR$. Then,

$$T_+ = Q^T T Q = RQ + \nu I$$

is also tridiagonal. It is called a *sift-QR* step.

If T is an unreduced matrix[2], choosing the value of ν from one of the eigenvalues of T and applying a shifted QR step, then the last diagonal element of the R matrix is $r_{nn} = 0$. In addition, the last column of T_+ is equivalent to $\nu \cdot I_n(:) = \nu \cdot e_n = T_+(:,\ n)$. This is called a *Perfect Shift*. Afterwards, we are going to make more emphasis on the choose of the ν.

3. *Cost.* In fact, our goal is to remove all the elements of the lower subdiagonal. To do it, we just need to apply a sequence of $n - 1$ Givens rotations.

2.2.1 Explicit single shift QR iteration

The *shift-QR* method accelerates the convergence of the QR iteration method, taking away a νI matrix from T. Choosing a correct value of s could have important effects to the effectiveness of our algorithm. We saw that if ν equals an eigenvalue, then $r_{nn} = 0$. But, usually, the computation of the eigenvalues will be very inefficient and, sometimes, impossible. However, with a good approximation of an eigenvalue, the element $(n, n-1)$ will be small after a QR shift step. We have two options to choose this value:

(a) The first one is to set ν as the last element of the main diagonal of the tridiagonal matrix, $\nu = a_n$. We do not have to make any computation to take this element.

(b) The second choice, generally more effective, is to set the shift as the eigenvalue of

$$T(n-1:n,\ n-1:n) = \begin{bmatrix} a_{n-1} & b_{n-1} \\ b_{n-1} & a_n \end{bmatrix}.$$

[2]A triangular unreduced matrix is a triangular matrix with all the subdiagonal and upperdiagonal elements nonzero.

Computing it, we obtain

$$\nu = a_n + d - \operatorname{sign}(d)\sqrt{d^2 + b_{n-1}^2}\,,$$

where $d = \frac{a_{n-1}-a_n}{2}$. Remark that it is the closer eigenvalue of the element a_n, the last diagonal element of the tridiagonal matrix. This shift method is called the *Wilkinson shift*. With this shift technique, the method is cubically convergent to a diagonal matrix.

2.2.2 Implicit shift version

In this section, we will see how the *shift-QR* method is executed. As we said, the iteration step of the method differs from the original QR method, in this case $T_+ = RQ + \nu I$. But, when it is computed, actually, there is no need of explicitly calculate the matrix $T - \nu I$. This fact has advantages when the shift is much larger than some of the diagonal elements of the tridiagonal matrix. We must keep in mind that our goal is to achieve a diagonal matrix, i.e., zero the upperdiagonal and the subdiagonal of the tridiagonal T matrix.

Remark that the tridiagonal matrix is a product of two bidiagonal matrix, $T = B^T B$. Therefore, the tridiagonal matrix elements are

$$T(i,j) = \begin{cases} d_i f_i & \text{if} \quad i+1 = j \\ f_{i-1}^2 + d_i^2 & \text{if} \quad i = j,\ i > 1 \\ d_i^2 & \text{if} \quad i = j = 1 \\ d_{i-1} f_{i-1} & \text{if} \quad i = j+1,\ i > 1 \end{cases}$$

Now, let us see how the Givens rotations are applied to the $T - \nu I$ matrix. Let us focus in $T - \nu I(1,\ 2)$ and let $c = \cos(\theta)$ and $s = \sin(\theta)$ be computed such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} d_1 - \nu \\ d_1 f_1 \end{bmatrix} = \begin{bmatrix} \times \\ 0 \end{bmatrix}$$

and set $G_1 = G(1,2,\theta_1)$. Let us see in detail what happens when the product $G_1 T$ is done.

As calculating the explicit formation of $T = B^T B$ is unwise from the numerical standpoint[3], we are going to apply the Givens rotations directly to B. As a

[3]Making this product can increase a lot the cost of the algorithm, but also, it can make lost some information of the matrix.

consequence of applying G_1, it appears a subdiagonal element

$$B \leftarrow BG_1 = \begin{bmatrix} \times & \times & 0 & 0 \\ + & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In order to zero it, we can determine Givens rotations U_1, $V_2\,_2, \ldots$, V_{n-1}, U_{n-1} to chase the element down

$$B \xrightarrow{U_1^T} \begin{bmatrix} \times & \times & + & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{V_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & + & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{U_2^T} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & + \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix} ,$$

where U_i^T's are applied by the left, while V_i's by the right. We must make this steps until we achieve the bidiagonal matrix again. So, the new bidiagonal $\bar{B}$ is related to B by

$$\bar{B} = (U_{n-1}^T \cdots U_1^T) B (G_1 V_2 \cdots V_{n-1}) = \bar{U}^T B \bar{V} \ .$$

As $\bar{V} e_1 = Q e_1$, by the implicit Q theorem (see [5]), we can assert that $\bar{V}$ and Q are essentially the same. This happens because the V_i matrices are Givens rotations, i.e., $V_i = G(i, i+1, \theta_i)$ for $i = 2, \ldots, n-1$. As a consequence, we can work directly with the matrix B, instead of the tridiagonal matrix T, that is more efficient.

These affirmations only are held when the tridiagonal matrix T is unreduced, therefore the $B^T B$. To verify it, we just have to check that $f_1 \cdots f_{n-1} \neq 0$ and $d_1 \cdots d_n \neq 0$, since the subdiagonal entries of $B^T B$ depend of the two diagonals of B.

2.3 The singular value decomposition algorithm

Knowing all the necessary background theory, now we will expose an algorithm to diagonalize any matrix: the *SVD* algorithm.

Theorem 2.1. (Singular Value Decomposition, SVD) Given $A \in \mathbb{R}^{m\times n}$, then there exist orthogonal matrices $U \in \mathbb{R}^{mxm}$ and $V \in \mathbb{R}^{nxn}$ such that

$$\Sigma = U^T AV = diag(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m\times n}, \quad \text{where } p = \min\{m, n\}$$

with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$.

Proof. See [5]; Theorem 2.5.2. □

Observation 2.2. The σ_i elements of the final diagonal matrix Σ are not necessarily different from 0. As a consequence, the rank of Σ could be less than m or n, i.e., $\text{rank}(\Sigma) = r \leq \min\{m, n\}$

Given $A \in \mathbb{R}^{m\times n}$ a matrix, the SVD transforms it to a diagonal matrix. Now, let us introduce the step-by- step algorithm.

1. Firstly, we transform A to a bidiagonal matrix by the algorithm shown previously,
$$A = U_B B V_B^T .$$

2. Secondly, we set $b_{i,i+1}$ to zero the elements which $|b_{i,i+1}| \leq \epsilon \cdot (|b_{i,i}| + |b_{i+1,i+1}|$, for any $i = 1, \dots, n-1$. Now, we have to find the largest q and the smallest p such that if
$$B = \begin{array}{cc} \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix} & \begin{array}{c} p \\ n-p-q \\ q \end{array} \\ \begin{array}{ccc} p & n-p-q & q \end{array} & \end{array},$$

   then B_{33} is diagonal, B_{22} has nonzero superdiagonal and the B_{ii} matrices are squared matrices. Therefore, if we find a zero in the superdiagonal, i.e., $f_k = 0$, for $k \leq n-1$, then we decouple B in two different matrix: B_{11} and B_{22}.

3. In order to apply the algorithm explained in 2.2.2, we need $T = B_{22}^T B_{22}$ to be an unreduced matrix. AS we explained, T is an unreduced matrix if $d_k \neq 0$ and $f_k \neq 0$, for any $p < k < q$. By the definition of B_{22} matrix, all the upperdiagonal elements are different from zero. Therefore, just we need to proof that all the diagonal elements of B_{22} are $d_k \neq 0$. From now on, in this point we will consider the dimension of the B_{22} matrix $r \times r$.

   Firstly, let us look at the $k < r$ elements. If there is any 0 between them, applying a Givens rotations sequence can zero f_k. For example, if $r = 6$ and $k = 3$, then by rotating in row planes $(3,4)$, $(3,5)$ and $(3,6)$ we zero f_3 and, consequently, the third row:

$$B_{22} = \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & + & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix}$$

$$\xrightarrow{(3,5)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & + \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,6)} \begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix}.$$

These Givens transformations just alter the diagonal and upperdiagonal elements of the rows 4, 5 and 6, respectively. So, it does not affect any other element of the matrix.

Now, if the last diagonal element is zero, $d_r = 0$. In this case, the last column can be zeroed with a series of column rotations in planes $(r - 1,\ r), \ldots,\ (1,\ r)$, so we are going to chase and go up this element, and. finally turn it zero.

If we have done any of this two last modifications, we have to return to the second point.

4. If any modification is made in the last point, the B_{22} matrix is unreduced, so we can apply the implicit-shift QR step. We compute the eigenvalue, ν, of

$$B^T B(n-1:n, n-1:n) = \begin{bmatrix} d_{n-1}^2 + f_{n-2}^2 & d_{n-1} f_{n-1} \\ d_{n-1} f_{n-1} & d_n^2 + f_{n-1}^2 \end{bmatrix}.$$

Then, we apply the method explained in the 2.2.2, that computes the $T_+ = T - \nu I$ step.

We have to apply this algorithm until $q = n$. When it happens, will mean that $B = B_{33}$, where, as we set, B_{33} is a diagonal matrix.

2.3.1 Complete orthogonal decomposition algorithm

Given a $A \in \mathbb{R}^{m \times n}$ matrix, the *complete orthogonal method*, by applying orthogonal matrices, we achieve a triangular matrix T:

$$A = QTZ^T$$

.

Let us see how we can find these orthogonal matrices. Firstly, a modification of the QR factorization algorithm is applied. Then, the QR factorization is used to achieve the T matrix.

The QR factorization, for deficiency rank matrices, does not always works correctly, as it does not necessarily produce an orthonormal basis for the ran(A)[4]. But, we can generate the orthonormal basis by modifying the QR factorization with permutation. By doing it, we obtain

$$Q^T A \Pi = \begin{array}{cc} \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} & \begin{array}{c} r \\ m-r \end{array} \\ \begin{array}{cc} r & n-r \end{array} & \end{array} , \tag{2.1}$$

where $R_{11} \in \mathbb{R}^{rxr}$ is upper triangular and regular, $Q \in \mathbb{R}^{mxm}$ is orthogonal and $\Pi \in \mathbb{R}^{nxn}$ is a permutation matrix. Notice that the Q matrix is a product of Householder matrices and Π is a product of interchange matrices.

Let us expose how these matrices are computed. Recall that this algorithm is a modification of the QR factorization algorithm, where the columns are permuted with the Π_i matrices. Let us explain how these permutations are done. To do it, there is no better way than looking what happens on a specific step. Once we have the k iteration completed, we have computed Householder matrices $H_1, \dots, H_{k-1}$, in order to zero the subdiagonal elements of the first k columns of the matrix. Also, the permutations $\Pi_1, \dots, \Pi_{k-1}$ are applied, and the following matrix is obtained:

$$(H_{k-1} \cdots H_1) A (\Pi_1 \cdots \Pi_{k-1}) = \begin{array}{cc} \begin{bmatrix} R_{11}^{(k-1)} & R_{12}^{(k-1)} \\ 0 & R_{22}^{(k-1)} \end{bmatrix} & \begin{array}{c} k-1 \\ m-k+1 \end{array} \\ \begin{array}{cc} k-1 & n-k+1 \end{array} & \end{array} ,$$

where $R_{11}^{(k-1)}$ is a non-singular and upper triangular matrix. Now, we want to zero

[4]Remark that ran$(A) = \{y \in \mathbb{R}^m$ such that $y = Ax$ for some $x \in \mathbb{R}^n\}$, so is the subspace of $\mathbb{R}^m$ created by the matrix A.

the $R_{22}^{(k-1)}$. Suppose that

$$R_{22}^{(k-1)} = \left[z_k^{(k-1)}, \dots, z_n^{(k-1)}\right]$$

is a column partitioning and let $k \leq p$ be the smallest index such that

$$\left\| z_p^{(k-1)} \right\|_2 = \max \left\{ \left\| z_k^{(k-1)} \right\|_2, \dots, \left\| z_n^{(k-1)} \right\|_2 \right\} .$$

Note that if $k-1 = \text{rank}(A)$, then this maximum is zero and we are finished. Otherwise, let Π_k be the $n - by - n$ identity matrix with the columns p and k interchanged and determine a Householder matrix H_k such that if $R^(k) = H_k R^{(k-1)} \Pi_k$, then $R^{(k)}(k+1 : m, k) = 0$.

In other words, Π_k moves the largest column in $R_{22}^{(k-1)}$ to the lead position and H_k zeroes all of its subdiagonal elements. Then, we have to recompute the norms by updating the old column norms with the next calculus

$$\left\| z^{(j)} \right\|_2^2 = \left\| z^{(j-1)} \right\|_2^2 - r_{kj}^2 .$$

Once we have achieved the 2.1 matrix, it can be further reduced. Multiplying by an appropriate sequence of Householder matrices, we can zero the R_{12} part of the matrix, but transforming the R_{11} matrix.

Applying the QR factorization algorithm

$$Z_r \cdots Z_1 \begin{bmatrix} R_{11}^T \\ R_{12}^T \end{bmatrix} = \begin{bmatrix} T_{11}^T \\ 0 \end{bmatrix} \begin{array}{c} r \\ n-r \end{array} ,$$

where the Z_i are Householder transformations and T_{11}^T is an upper triangular matrix. Thus, we obtain that $\text{rank}(A) = r$. Therefore, adding this procedure to the QR modified factorization, we obtain

$$Q^T A Z = T = \begin{array}{cc} \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} & \begin{array}{c} r \\ m-r \end{array} \\ \begin{array}{cc} r & n-r \end{array} & \end{array} ,$$

where $Z = \Pi \cdot Z_1 \cdots Z_r$.

Chapter 3

Least Squares approximated solutions of linear systems

Given a matrix $A \in \mathbb{R}^{m \times n}$ and two vectors $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, we are going to see how to give all the possible solutions of any linear system or at least an approximation of them.

Therefore, our goal is to find the solution vector x such that makes Ax closer to b. Thus, in a mathematical way, we want to find

$$\min_{x \in \mathbb{R}^n} \| Ax - b \|_2 \ . \tag{3.1}$$

To find those vectors is called the *least squares* (LS) problem, sometimes, with more than one possible x_{LS} solution. In order to know the error between $A\tilde{x}$ and b, where $\tilde{x}$ is an approximate solution, we define the *minimum residual* $r_{LS}(\tilde{x}) = b - A\tilde{x}$.

To find all the possible solutions of any system, let us introduce the *normal equations*, that transforms a general linear system into a compatible linear system, by doing

$$A^T Ax = A^T b \ . \tag{3.2}$$

To solve it, we are going to decompose the matrix A, using orthogonal matrices, by two methods: the *singular value decomposition method* and the *complete orthogonal decomposition methods*.

3.1 The singular value decomposition method

As we explained in the last chapter, applying this algorithm, we decompose any matrix as $A = U\Sigma V^T$. By applying the normal equations,

$$V\Sigma^T U^T U\Sigma V^T x = V\Sigma^T U^T b \; . \tag{3.3}$$

Now, applying $Q^T = Q^{-1}$ for any orthogonal matrix, we obtain

$$\Sigma^T \Sigma V^T x = \Sigma^T U^T b \; . \tag{3.4}$$

Remark that Σ is a diagonal matrix, but not always invertible. Suppose we have r diagonal elements different from zero, the $\mathrm{rank}(A) = r$. Then, we can define $\Sigma_1 = \Sigma(r,r)$, an invertible matrix. The solution is determined by

$$\Sigma_1^T \Sigma_1 (V^T x)_1 = \Sigma_1^T c \; ,$$

we call c to the first r columns of the $U^T b$ vector. Applying Σ_1^{-1} in each side of the equation,

$$(V^T x)_1 = \Sigma_1^{-1} c \; .$$

Therefore, a solution is fixed by

$$\begin{bmatrix} v^{(1)T} \\ \vdots \\ v^{(r)T} \end{bmatrix} x = \begin{bmatrix} \bar{a}_1 \\ \vdots \\ \bar{a}_r \end{bmatrix} = \Sigma_1^{-1} c \; .$$

Knowing one LS solution, now our goal is to find all the other possible solutions.

Let us express the x vector in the orthonormal basis of the V matrix

$$x = a_1 v^{(1)} + a_2 v^{(2)} + \cdots + a_r v^{(r)} + a_{r+1} v^{(r+1)} + \cdots + a_n v^{(n)} \; ,$$

thus, all the other possible solutions are generated by adding multiples of the last $n-r$ columns of the matrix V to the previous found solution. Therefore, all the possible solutions are in the linear variety

$$E = \bar{a}_1 v^{(1)} + \bar{a}_2 v^{(2)} + \cdots + \bar{a}_r v^{(r)} \; + \; \langle v^{(r+1)}, \ldots, v^{(n)} \rangle \; , \tag{3.5}$$

where its dimension is $n-r$.

Observe that if the $\mathrm{rank}(A) = n$, then the $\dim(E) = n - n = 0$, so there is only one LS solution.

3.2 The complete orthogonal decomposition method

In a similar way as in the SVD algorithm, all the solutions, in particular the LS solutions, are determined using the complete orthogonal decomposition.

As we have viewed, $A = QTZ^T$. Considering the normal equations with this decomposition, we obtain

$$ZT^TQ^TQTZ^Tx = ZT^TQ^Tb \; . \tag{3.6}$$

Using the orthogonal matrix properties,

$$T^TTZ^Tx = T^TQ^Tb \; . \tag{3.7}$$

Remark that T is a triangular matrix but nor always invertible. However, we have defined the T_{11} invertible matrix. Notice that $\mathrm{rank}(A) = r$.

Considering the first r columns of T in 3.7, we obtain

$$T_{11}^TT_{11}(Z^Tx)_1 = T_{11}^Tc \; ,$$

where $(Z^Tx)_1$ are the first r rows of Z^T multiplied by x and c equals to the first r rows of Q^T multiplied by b, $c = (Q^Tb)_1$. As T_{11}^T is a regular matrix, we can find the inverse, so

$$T_{11}(Z^Tx)_1 = c \; .$$

Therefore, a solution is fixed by

$$\begin{bmatrix} z^{(1)T} \\ \vdots \\ z^{(r)T} \end{bmatrix} x = \begin{bmatrix} a_1 \\ \vdots \\ a_r \end{bmatrix} = T_{11}^{-1}c \; .$$

Now, we have found a LS solution. Our goal will be to find all the other possible solution.

We can express the vector x in the orthonormal basis[1] of the Z matrix as

$$x = a_1z^{(1)} + a_2z^{(2)} + \cdots + a_rz^{(r)} + a_{r+1}z^{(r+1)} + \cdots + a_nz^{(n)} \; ,$$

[1]As is an orthonormal basis, as we said in 1.1, it does not alter the norm of the vector

where the first r components are determined by

Therefore, all the possible solutions are generated by adding multiples of the last $n - r$ columns of the matrix Z to the previous found solution. Finally, all the possible solutions are in the linear variety

$$E' = \bar{a}_1 z^{(1)} + \bar{a}_2 z^{(2)} + \cdots + \bar{a}_r z^{(r)} \ + \ \langle z^{(r+1)}, \ldots, z^{(n)} \rangle \ . \tag{3.8}$$

Chapter 4

Program

In the last chapters, we studied how to solve linear systems with different methods. In order to compare which method is more efficient, we are going to use a program in C++ language. This program solves linear systems with the Singular Value Decomposition and the complete orthogonal decomposition methods.

In the full rank case, instead of using the complete orthogonal decomposition method, the program will directly apply the QR factorization. In other words, when it is needed, it does the column pivoting and the consequence factorization. Also, the program takes care of some special cases explained in 2.1. So, when the bidiagonalization is not completed in a matrix $A \in \mathbb{R}^{m \times n}$, $m < n$, the program itself applies the correspondent Givens rotations to manage it.

4.1 Structure of the program

The program is divided in 6 different modules which include functions with a common objective. Let us explain how they are distributed and which sort of functions include each module:

1. **DSC1 Basics:** The first module includes the most basic functions that will be used in the following modules. There are functions that make basic actions such as reading and printing a linear system, transposing matrices, matrices and vector permutations, generating random matrices, copying data information and comparing two different matrices. In addition, there are some functions which makes matrix by matrix or matrix by vector products.

2. **DSC2 Linear Transformation:** In this module, we can find functions which apply the Givens and Householder matrices. Those can be applied by the left or by the right, with some many changes on its products, as we has shown in 1.3 and in 1.4. Always these functions are going to apply these two matrices in the efficient way explained before.

3. **DSC2 Orthogonal triangular decomposition:** Now, we already have all the tools to decompose matrices. Here, we can find many functions that computes the complete orthogonal decomposition. Firstly, we have the QR factorization function (QRD) and a function that checks the decomposition has done well. Then, we include the permutation option in the QR factorization, also with the correspondent check function. This module also includes functions that complete the decomposition. So, there are the TZ and the final QTZ decompositions.

4. **DSC2 Singular value decomposition:** Within the five functions of this module, two of them are the most important. These ones apply the algorithm explained in 2.3 by parts. Firstly, we have the bidiagonalization of the original matrix. Then, we apply the rest of the algorithm with the SVD_B function. This function includes all the possibilities and different scenarios that we can find applying the SVD algorithm. Also, there are some "check" function just to prove that the decomposition is done correctly. Finally, we find the main function of the module that just call the two main functions and write the results, if it is necessary.

5. **DSC3 Solutions:** This module will give us not only the least square solution, but also all the possible solution of the linear system. Roughly, it is divided in two: the first part computes and writes the QTZ decomposition solutions, while the second part does it for the SVD solutions. From the respective decomposition, the functions of this module solve linear systems by different techniques - back and forward replacement and applying orthogonal inverse matrices [1] - and writes all the solutions.

6. **LSq Test:** Finally, we arrive at the main function of the program. This function call another one which computes all the functions. It does including a file and two Boolean variables. The first one indicates if the algorithm has to be written step by step. If it is false, the program will only write the solution of the two methods. The second Boolean variable tells if the matrix is going to be generate in a random way or is read of a file. When this function is executed, it computes the time of execution of the two algorithms and compares it. Also, it computes the difference between the solution given by the two algorithms.

[1]The orthogonal matrices always has inverse, specifically $Q^{-1} = Q^T$

4.2 The solution file

As we have explained in the last section, the program could write just the solution or a step by step solution, where we can see how the algorithm works. Firstly, in the two options, we will find the linear system. Then, the step by step option will be different form the other one, as it is going to include more information

At the beginning, we are going to find the SVD method explained step by step. First, the bidiagonalization method with the correspondent orthogonal matrices is written. Next, the SVD algorithm step by step, indicating the iterate number and the q and e vectors that represents the upperdiagonal and the diagonal elements of the B matrix. Then, the Implicit Version algorithm is showed, explained in 2.2.2, where the kp and kq values are equivalent to the p and q values of 2.3. In each step, are printed the T and R matrices, the orthogonal matrices needed to apply in the Implicit Version algorithm. Finally, the program shows the U, S and V matrices, with the least square solution and all the possible solutions, which we have found following the method in 3.5. The first row is a particular solution, while the following rows are the vectors of the subspace. Also, the RMS error is showed.

Now, let us focus on the QTZ decomposition algorithm. In this case, the program only shows the three different matrices decomposed and the solutions. Here we are in the same situation as the SVD. The first row gives the particular solution and the following ones, the vectors which generate the subspace, explained in 3.8. In the other case, we are only going to see the Least Square solution, all the possible solutions and the execution time. Finally, the program compares the two results and the computation time of the methods.

4.3 Comparison SVD vs QTZ

In order to see which method is more efficient, we generate different random matrices with dimensions not bigger than 250×250, where the elements take the values between $(-250, 250)$. There is the possibility to change the range of this value in the LSq Test module. In the table 4.1, there are compiled a few random matrices that have been executed. Knowing their dimensions, we can see the computation time[2] in microseconds (10^{-3} seconds) and the correspondent comparison between the two methods: SVD / QTZ.

[2]This computation time is taken when the program does not write the solution step-by-step, so the writing part does not affect it too much.

m	n	SVD	QTZ	SVD / QTZ
1	94	75	43	1.7442
17	80	401	272	1.4743
250	249	228506	62018	3.6845
14	47	166	114	1.4561
23	91	978	637	1.5353
19	62	403	229	1.7598
74	16	385	240	1.6042
6	49	95	67	1.4179
4	61	131	101	1.2970
59	14	230	129	1.7829
154	102	17892	7356	2.4323
59	134	6175	3664	1.68539
120	104	13102	4507	2.9070
101	74	5785	2497	2.3167
182	11	1063	1555	0.6836
64	16	314	162	1.9382
143	143	31255	8456	3.6962
218	205	114867	35046	3.2776
Average				1.7520

Table 4.1: Computation time compilation of random matrices.

Observe that SVD takes in average 1.75 more time than QTZ method. Probably this sample is not sufficient to set an exact relation between the computation time of the two algorithms. However, we can conclude that QTZ is more efficient than SVD.

In addition, we have to take into account that this program is made in a few days and it cannot be comparable with one function of libraries as *redsvd* or *eigen*. Moreover, we have made a program in R to see the computation time of an efficient algorithm.

4.4 R simulator

This R simulator generates 100 random matrices and computes the SVD algorithm with a defined function, found in [9]. These random matrices have random dimension between $(2,\ 250) \times (2,\ 250)$, which those elements are random numbers between $(-250,\ 250)$.

The software executes the *La.svd* that returns U, Σ and the transposed V^T matrices. Once done it, a comparison between the original matrix A with the product of matrices $U\Sigma V^T$ is done. Finally, it averages the computation time and the error[3] of each iteration.

In conclusion, as we can see in the Annex, the average error is about 10^{-11} and the computation time is around 0.006 seconds (6 microseconds) for each iteration. As it is expected, this algorithm is much more efficient, the computation time of this function is much more lower than the program in C++.

[3]In this case, we understand the error as the difference between the original matrix and the matrix originated by the product of the SVD resultant matrices. In fact, could be interpreted as the tolerance used by the SVD function.

Chapter 5

Application to geodesy

In the last chapters, we have seen two possible methods for solving linear systems. As an application of these methods, let us introduce a geodesy problem found in [1]. To compute and represent the problem, we have a simulator. To store and represent the information, we will introduce some concepts of graph theory. Finally, we will see some inconveniences of this application and how to improve some of them.

5.1 Introduction

Geodesy is the science that studies the Earth at various global and local scales. As its known, the shape of the Earth is not exactly spherical, but similar to it. Approximating its shape is the task that geodesy does as a global scope. To achieve it, a geodesy network is used. A geodesy network is formed by a set of geodesic vertices, which positions are needed to be measured with a high precision.

Another utility of the geodesic vertices takes place in the local scale to accurately reference nearby points. This network is developed as an utility it can give to help to the take into different studies about some geographic limitations of the territory.

The objective of this chapter is to make an adjustment method of a positions of geodesic vertices of a given network. We have measure of some distances and of some angles between them, and we want to refine the exact positions of the vertices. Notice that the angles between three nodes are only measured when we know the two adjacent distances, see Figure 5.4. The measures are taken in order to determinate the new position of the geodesic vertices. In other words, these measures generates some constraints between each points. Thus, we want to

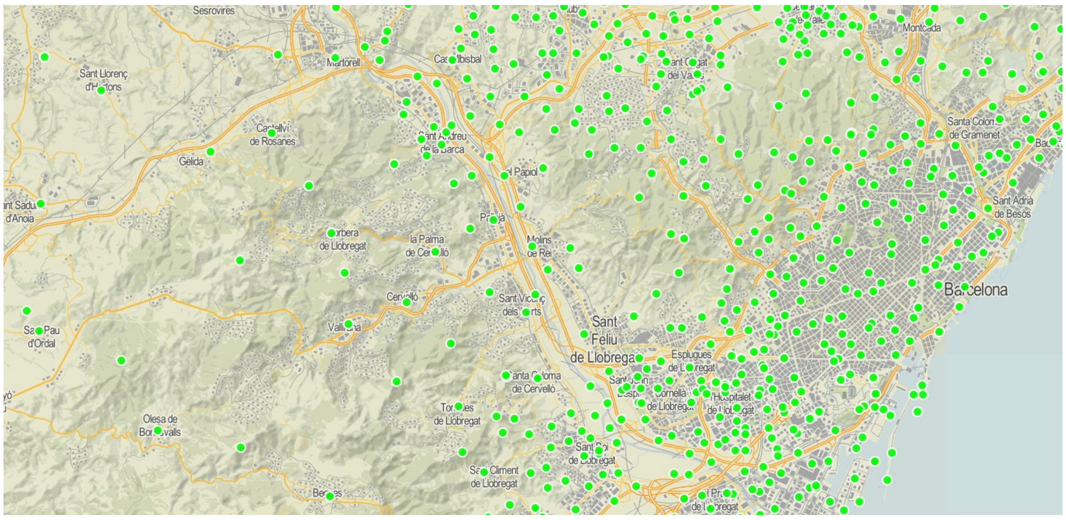


Figure 5.1: Map of the geodesy network of a slice of the province of Barcelona. Source: Institut Cartogràfic i Geològic de Catalunya (ICGC). Last update: 23/7/2018. Used with license.

adjust the position of the geodesic vertices that satisfies the constraints associated with the measured distances and angles.

We may notice that the constraints depend directly on how the measurements are taken. Nowadays, the tool used to take the measures is total station, see the Figure 5.2, which integrates a theodolite and an electronic distance measurement. This machine computes the angles in a high precision, with the theodolite incorporated system, and gives us the distance between the positions of the geodesy vertices.

Figure 5.2: A theodolite. Source: Wikimedia Commons. Used with license.

These constraints generate a system of equations. Each equation of this system is determined by a measure. Therefore, we will have as many equations as taken measures. Then, we will find any type of systems: if there are few measures, the system may be underdetermined; while taking as much measures as possible, an overdetermined system is generated. Regardless, our objective will be finding a least squares adjustment of the position of the geodesic vertices.

Remark that the accuracy of the position of the vertices is directly related with the number of equations of the system. With an overdetermined system, we will have much more precision, as the solution is going to be determined by more equations and, therefore, it will also satisfy more constraints. Thus, the more measurements are taken, the more precision we will achieve.

Moreover, we simplify this problem restricting it to the 2 dimensional space. However, notice that to extend the problem to the 3 dimension, we only need to add one more coordinate to the points.

Figure 5.3: A geodesic vertex of the IGCG located in Molins de Rei.

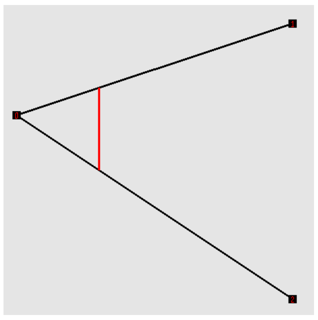


Figure 5.4: Angles are measured when adjacent distances are known.

Notice that the problem will not be linear, as the equations include at least quadratic expression for distances. So to solve it, we linearize the problem by firstly using an initial approximation of the points and then applying the Newton's method. Finally, we the SVD algorithm is applied to find the solution.

5.2 Reasoning

In this section, we will introduce the problem in a mathematical way. Our goal will be finding the least square adjustment of the nodes which best satisfy the constraints. We also have to take into account that the problem will not be linear, so we must linearize the problem.

Let us consider a set of approximated geodesic points $x = (x_1, \dots, x_n)^T$, where $x_i = (x_i^{(1)}, x_i^{(2)})$, as we are restricting the problem in $\mathbb{R}^2$. Also, we suppose that we know some of the distances between the points and some of the angles between three points, when we have the two adjacent distances.

Firstly, we are going to compute an error function which compares the measures with the expected distances or angles, respectively. Those are computed with the approximated components of the points.

Let us define the function e_{ij}, for each distance measure between the points x_i and x_j

$$e_{ij}(x) = d(x_i, x_j) - z_{ij} \ ,$$

where the value of z_{ij} indicates the square obtained distance and the function d, the expected measurement. The function d computes the distance between the approximated points x_i and x_j. Notice that the distance between x_i and x_j is the same distance between x_j and x_i, therefore, $z_{ij} = z_{ji}$, and so $e_{ij} = eji$. For computing the d function, we consider the Euclidean distance, therefore

$$d(x_i, x_j) = (x_i^{(1)} - x_j^{(1)})^2 + (x_i^{(2)} - x_j^{(2)})^2 \ . \tag{5.1}$$

Our goal is to zero the e_{ij} functions, so we need to solve the equation

$$(x_i^{(1)} - x_j^{(1)})^2 + (x_i^{(2)} - x_j^{(2)})^2 - z_{ij} = 0$$

for every x_i, x_j where the distance measure exists.

Supposing now three points x_i, x_j and x_k, knowing the distances from x_i to x_k and x_i to x_j and suppose the angle between the three points has been measured, the angle $\widehat{ijk}$. We define an error function $e_{\widehat{ijk}}$ for each angle constraint, as

$$e_{\widehat{ijk}}(x) = g(x_i, x_j, x_k) - z_{\widehat{ijk}} \ ,$$

where $z_{\widehat{ijk}}$ is the measure of the angle between the points. In this case, $e_{\widehat{ijk}} \neq e_{\widehat{jik}}$, as there take different angles measure. The function g is the computed angle with the approximated points x_i, x_j and x_k

$$g(x_i, x_j, x_k) = \frac{(\overrightarrow{x_i x_j} \cdot \overrightarrow{x_i x_k})^2}{\|\overrightarrow{x_i x_j}\|^2 \cdot \|\overrightarrow{x_i x_k}\|^2} \tag{5.2}$$

$$= \frac{(x_i^{(1)^2} - x_i^{(1)} x_k^{(1)} - x_j^{(1)} x_i^{(1)} + x_j^{(1)} x_k^{(1)} + x_i^{(2)^2} - x_i^{(2)} x_j^{(2)} - x_k^{(2)} x_i^{(2)} + x_j^{(2)} x_k^{(2)})^2}{\|\overrightarrow{x_i x_j}\|^2 \cdot \|\overrightarrow{x_i x_k}\|^2} \ ,$$

where $\overrightarrow{x_i x_j}$ is the vector formed by the point x_i to x_j and $\|\overrightarrow{x_i x_j}\|^2$ is the Euclidean norm of it. Remark that the function g is equivalent to $\cos^2\theta$, for an angle θ. Consequently, in order to compare the function g with $z_{\widehat{ijk}}$, the measure will be given as $z_{\widehat{ijk}} = \cos^2\alpha$, for an angle α.

In order to reduce the operations, we are going to use the measured distances instead of the norms of the vector between the correspondent points, so $z_{ij} = \|\overrightarrow{x_i x_j}\|^2$ and $z_{ik} = \|\overrightarrow{x_i x_k}\|^2$. So, the equation needed to satisfy is

$$(\overrightarrow{x_i x_j} \cdot \overrightarrow{x_i x_k})^2 \ - z_{\widehat{ijk}} \cdot z_{ij} \cdot z_{ik} = 0 \ ,$$

for every x_i, x_j and x_k points which we have the angle measure. Therefore, we can redefine $e_{\widehat{ijk}}$ as

$$e_{\widehat{ijk}} = (\overrightarrow{x_i x_j} \cdot \overrightarrow{x_i x_k})^2 \ - z_{\widehat{ijk}} \cdot z_{ij} \cdot z_{ik} \ .$$

To sum up all these functions, we define a function $F : \mathbb{R}^{2n} \to \mathbb{R}^m$, where n is the number of points and, as we are in the $\mathbb{R}^2$ dimension, is multiplied by 2; while m is the number of equations, i.e., the number of e_{ij} equations added to the number of e_{ijk} ones. As we said in the last section, depending on the values of m and $2n$ we will have an overdetermined or underdetermined system. The function $F(x)$ will return a vector with all the values of e_{ij} followed by the values of $e_{\widehat{ijk}}$. Therefore, our objective will be to find the vector of points x such that zeroes $F(x)$.

5.2.1 LS resolution

Firstly, we observe that $F(x)$ is a non-linear function, as it contains at least quadratic elements because of the distance and angles computations. In order to linearize it, let us apply the Newton's method that sets the equation

$$DF(x)\Delta x = -F(x) \ , \tag{5.3}$$

where $DF \in \mathbb{R}^{mx2n}$ is the Jacobian matrix of the function F and $\Delta x = \bar{x} - x$, the difference between the oldest points and the new approximation points.

The reason for adding the restriction of having the two adjacent distances for measuring an angle, now it gets significance. Using the z_{ijk} values in 5.2, instead of computing the norms, greatly simplifies the calculation of the Jacobian matrix. Moreover, as we have initially approximated the vector x, notice that the unknown vector in this linear system is the $\bar{x}$. When it is solved, then $x = \bar{x}$ and the system is set out again, until $F(\bar{x}) = (0, \ldots, 0)$.

For solving the linear system, we are going to apply the SVD algorithm until we obtain the least square solution vector.

5.3 Data representation

Once found the approximate solution, the representation of it will be based on graphs. Before explaining how it will be, let us introduce some basic concepts of graph theory.

5.3.1 Graph preliminaries

Let us introduce a few definitions of basic concepts of the graph theory. Working with the graph G, we will notice the edges of the graph as $E(G)$ and the vertices as $V(G)$.

Definition 5.1. Let $u, v \in V(G)$, if there exist $u - v \in E(G)$, these two vertices are *adjacent*, or *neighbours*. Let $e, f \in E(G)$, $e \neq f$, are *adjacent* if they have an end in common: $e = v - u$ and $f = v - w$, where v, u and w are vertices of G.

Now, let us apply these two definitions to explain another concept.

Definition 5.2. The *line graph* $L(G)$ of G is the graph on $E(G)$ in which e, $f \in E(G)$ are adjacent as vertices if and only if they are adjacent as edges in G.

In other words, the line graph $L(G)$ takes the edges of the graph G as vertices. Those adjacent vertices in G are going to be connected in $L(G)$. In the Figures 5.5, 5.6, 5.7 and 5.8, it is showed how a lineal graph is formed.

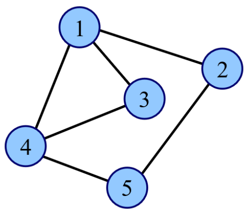


Figure 5.5: Graph G. Source: *Wikimedia Commons*. Used with license.

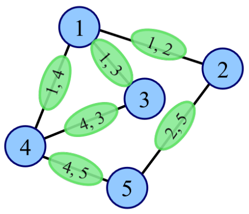


Figure 5.6: Edges in G. Source: *Wikimedia Commons*. Used with license.

5.3.2 Information graphs

As we said, we are going to represent the data points and its distances and angles using graphs. In particular, we are going to use two different graphs to represent all the information.

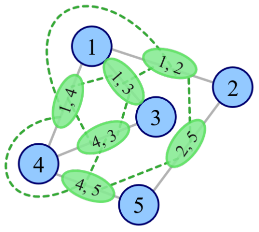


Figure 5.7: Edges of $L(G)$. Source: *Wikimedia Commons*. Used with license.

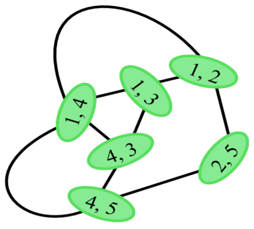


Figure 5.8: The graph L(G). Source: *Wikimedia Commons*. Used with license.

The first graph G is a weighted non directional graph. Its vertices represent the geodesic vertices, while the distance measurements represented as weighted edges: each edge $e \in E(G)$ corresponds to the measured distance $d_e^2 = z_e$, the weight between its vertices.

The second graph is a weighted non directional subgraph of the line graph of G. We denote it as $H \subset L(G)$. We are going to use this graph for representing the angles measures. Recall that we need that three with at least two adjacent edges for having an angle measure. The vertices of H are some of the edges of G, $V(H) \subset E(G)$, in consequence from the definition. The edge of H join two elements of $E(G)$. In other words, each $a \in E(H)$ links two adjacent $e,\ f \in E(G)$: $e = v - u,\ f = v - w$, and its weight corresponds to the squared cosine $\cos^2 \alpha_a$ of the measured angle α_a between them there must exist $u, v \in E(G)$.

Once the two graphs are generated, we will represent both in one image, superposing the graph H on G. Let us illustrate it with the next example, see Figure 5.9.

In this graph, we can observe that we know the distance between the vertices 6 and 7. On the contrary, the distance between the vertices 3 and 4 have not been measured, so the edge $3-4$ does not exist.

As we can see, there is one edge of the graph H, coloured in red, between the edges $0-1$ and $0-2$ of the graph G. This red edge represents that the measure of the of the angle $\widehat{012}$ is given. Also, we can see that the vertex 4 is adjacent to 5 and 2, but in this case we do not know the angle measure $\widehat{452}$. Otherwise, with the vertices 2, 4 and 3, we do not have the possibility to receive any angle information as $3-4 \notin E(G)$.

F is composed of two parts. The first one contains all the equations related to the distances measures. As many equations as the number of elements of $E(G)$. Remark that the weight of each edge of G equals to $z_e = d_e^2$. The second part of the function compiles all the equations related to the angles measures $z_a =$

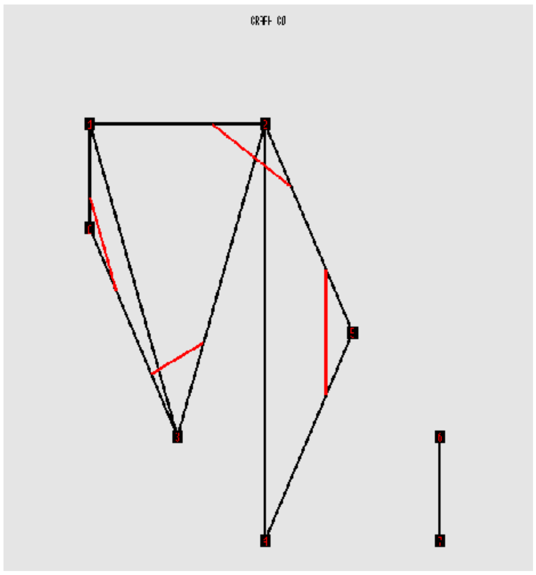


Figure 5.9: Graph $G0$.

$cos^2\alpha_a$. Of these ones, we are going to have as many as the number of edges of H. Consequently, the dimension of F will partly be determined by $m = \#E(G) + \#E(H)$.

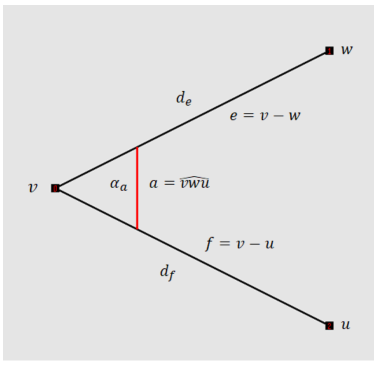


Figure 5.10: Angle of the weighted graph G.

Knowing how to represent the problem, in the next section, we are going to see how the problem is set out and how the simulator works.

5.4 Adaptation of the problem

In the simulator, we have made an adaptation of this problem. First, let us introduce how the data information is received. Then, we will see how the program formulates and solves this approximation to the original problem.

5.4.1 The data file

The data is received with the next structure. First, we know the quantity of points v_n, the unknowns. In the same line, we have the number of edges, e_n, and angles, a_n, that have been measured, the constraints.

Then, we will find v_n lines were the coordinates of the points are specified. Each point has a line, the first column indicates the number of vertex, the second the first coordinate and the third the second coordinate.

Below of that, we are going to find e_n lines where the edge information is specified, actually, if the edge exist or not. The first column gives the edge number, the second one the initial vertex and in the third one the final vertex of the edge.

Finally, we will find a_n lines, which say if the angle exist or not. Firstly, we will find the angle numeration and, in the next three columns, there are three numbers that represent the vertex number. Between the first and the second number must exist an edge and also between the first and the third number. Let us illustrate it with the data file of the example shown in the Figure 5.9:

$$\begin{array}{rl} & \begin{array}{ccc} 8 & 9 & 4 \end{array} \\ \text{Vertices} & \left\{ \begin{array}{ccc} 0&0&2\\1&1&0\\2&2&1\\3&1&4\\4&2&5\\5&3&3\\6&4&4\\7&4&5 \end{array} \right. \end{array}$$

$$\text{Edges}\left\{\begin{array}{ccc} 0 & 0 & 1\\ 1 & 0 & 3\\ 2 & 1 & 3\\ 3 & 2 & 1\\ 4 & 2 & 5\\ 5 & 3 & 2\\ 6 & 4 & 2\\ 7 & 4 & 5\\ 8 & 6 & 7 \end{array}\right.$$

$$\text{Angles}\left\{\begin{array}{cccc} 0 & 0 & 1 & 3\\ 1 & 2 & 1 & 5\\ 2 & 3 & 0 & 2\\ 3 & 5 & 2 & 4 \end{array}\right.$$

5.4.2 Formulate and solve the problem

As we saw, we may notice that any measurement information have read. What the simulator is going to do is to calculate the distances of the vertices and the angles that exist and, then modify it. By doing that, we will obtain an approximation of points that does not satisfy the constraints. Now, let us explain step-by-step how the simulator exactly works.

Firstly, once the information is read, the first step is to compute the distances between the adjacent points using the function 5.1 and the angles of the read points using the function 5.2. Now, the program will alter these distances and angles multiplying by $1 + \lambda \cdot \epsilon$, where λ is a random variable and ϵ is a variable that we will modify in the simulator. So, when we modify this value, the points will not satisfy the constraint.

In order to find the new points which satisfy the new constraints, the program applies the equation 5.3, computing the Jacobian matrix and updating the vector x. Finally, the simulator call some functions of the program of chapter 4, to solve the linear system, applying SVD.

Depending on the number of distances and angles read, we will find different types of linear systems. Therefore, the simulator will find the exact solution of it or give us a least square adjustment of the solution. Also, the value of ϵ, could conditioned finding the solution. If it has a big, will be more difficult for the algorithms gives a satisfying solution.

When the new points are found, the simulator contains some modules which represents the graph. Also, we can see if the program finds the exact solution or, otherwise, if it just solves it for a least square solution. For that, there is a window that gives the error of each node of the graph.

5.5 Inconveniences

Even if this method solves the problem, it has two main inconveniences. Firstly, if we consider the problem as the way announced in 5.1, the constraints depend directly on how the measurements are taken. However, the total station has a limited precision and, so on, it can have some small errors that accumulated can have relevance to the final result.

Secondly, we may notice that the simulator does not take into account any fixed point when finding the position of the geodesy vertices. This fact generates a slack problem, since the graph solution can be translated or rotated and will continue being a solution. That is, having a satisfying set of points for the given measures, if we move all the set of points together around the $\mathbb{R}^2$ plane, this new points are still going to be solution, as they accomplish the constraints since the distances and the angles between them are the same. Adding the possibility to the simulator of fixing a few points, at least two, would be a solution for this inconvenient. By fixing the positions of at least two points, these positions should remain fixed in the adjustment process and they should not be unknowns in the system of equations.

Conclusions

Two numerical methods for solving general linear systems: the SVD and the complete orthogonal decomposition methods give all their least squares solutions using orthogonal transformations.

The given software in C++ has been understood and modified in order to generate random matrices and compare the two methods studied. With this modification, we have seen that the complete orthogonal method is more efficient than the SVD method. In fact, this was expected since fewer steps are required to apply the first method. In addition, we studied the effectiveness of th SVD algorithm using a basic package function of R.

In the last chapter, we have made a geodesy application. We have applied the methods to adjust the position of vertices in a geodesic network. To do it, we have implemented a software which includes a simulator to represent the problem and the solutions.

Therefore, we achieved all the proposed goals, being able even to extend some parts. Moreover, this project could be more extended in the following points. Firstly, as we have done for the SVD algorithm, an equivalent R simulator can be done for the complete orthogonal decomposition. In the part of the geodesy application, one improvement would be doing the $\mathbb{R}^3$ extension, previously mentioned. In addition, we can study the different techniques used for solving this adjustment geodesic problem. Last but not least, solving the slack problem explained in 5.5. For solving it, including the option of setting two fixed points in the simulator, would be enough.

Finally, we must notice that the SVD algorithm has many other application, as Principal Component Analysis and how it can be applied to the compilation of images, compiled in [8].

Appendix A

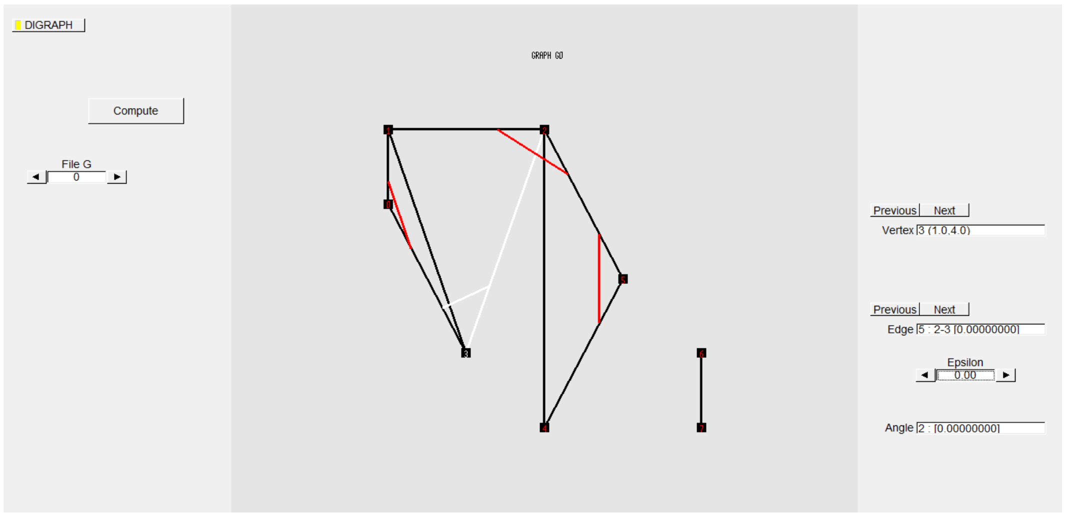
R program: SVD Algorithm

This program computes SVD into 100 different random matrices. After that, it averages the computation time and the error of each one. These matrices are created with random sizes, between 1 to 250, and random element, which values are in (-250, 250).

```
#It uses the library tictoc for calculate the
#computation time.

library(tictoc)
tic.clearlog


#It sets the number of different matrices and creating
#the vector to save the errors.

j <- 100
error <- vector(mode = "numeric", j)

#It makes a loop for computing the 100 matrices.

for (i in 1:j){
  #Setting m and n values for an integer random
  #number between 1 and 250.
  m <- sample(2:250, 1)
  n <- sample(2:250, 1)

  #Matrix created randomly with elements in (-250, 250).
  A <- matrix((runif(m*n)-0.5)*500, ncol=n)
  #Applying the SVD function and computing its time.
  tic()
  y <- La.svd(A)
  toc(quiet = TRUE, log = TRUE)

  #Defining the U, D and Vt matrix resulting from the SVD
  #algorithm.
  D <- diag(y$d)
  U <- y$u
  Vt <- y$v

  #Making the product to subsequently compare with the
  #original matrix with the Frobenius norm for matrix.
  SVD <- U%*%D%*%Vt
  error[i] <- sqrt(sum(colSums((A - SVD)^2)))
}

#Time average
log.txt <- tic.log(format = TRUE)
log.lst <- tic.log(format = FALSE)
tic.clearlog()
timings <- unlist(lapply(log.lst, function(x) x$toc -
                         x$tic))
av_time <- mean(timings)
av_time
# [1] 0.0063

#Error average
av_error <- mean(error)
av_error
#[1] 4.310559e-11
```

Appendix B

In this appendix, we are going to show how the simulator adjust the points with the modified measures multiplied by $1 + \lambda \cdot \epsilon$, where, as mentioned in 5.4.2, λ is a random value. We are going to show how the position of the points are modified, while the value of ϵ increases. In each image, we would see the error for each measure nearest to the mouse. In particular, we are going to see the value of e_{23}, the value of e_{302} and the position of the geodesy vertex.



File G: 0

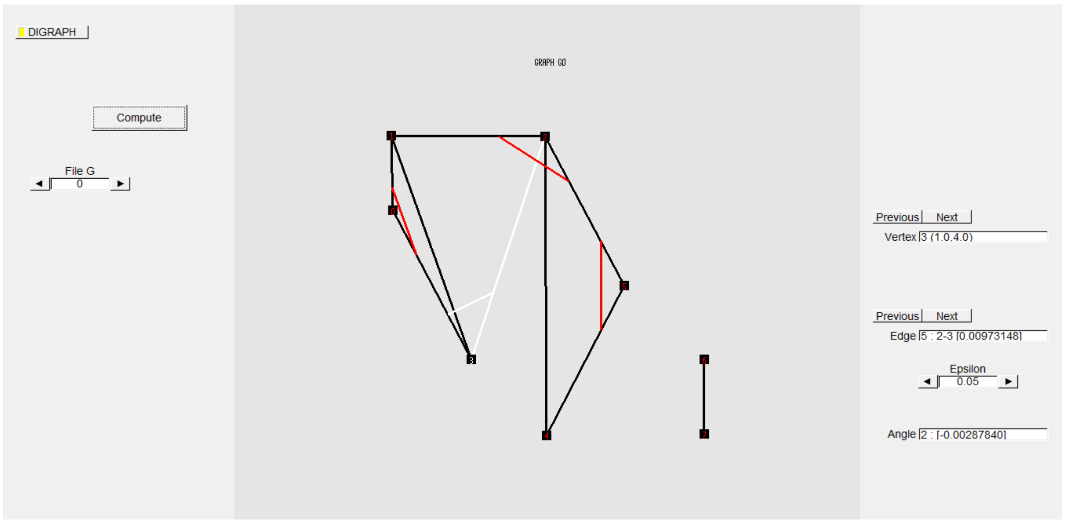
Vertex 3 (1.0,4.0)

Edge 5 : 2-3 [0.00000000]

Epsilon: 0.00

Angle 2 : [0.00000000]

Figure 5.11: $\epsilon = 0.00$.



File G 0

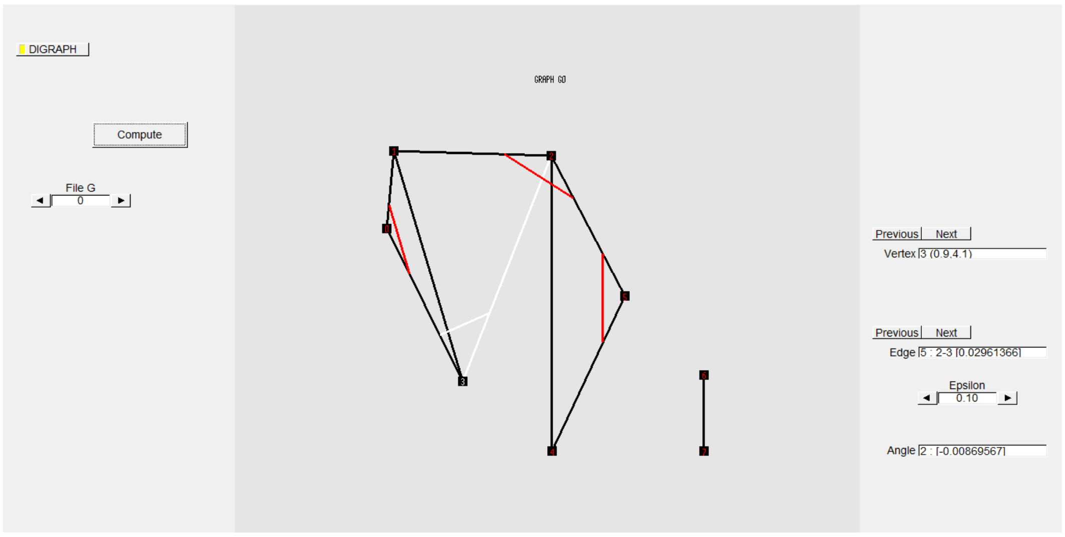
Vertex 3 (1.0,4.0)

Edge 5 : 2-3 [0.00973148]

Epsilon 0.05

Angle 2 : [-0.00287840]

Figure 5.12: $\epsilon = 0.05$.



File G 0

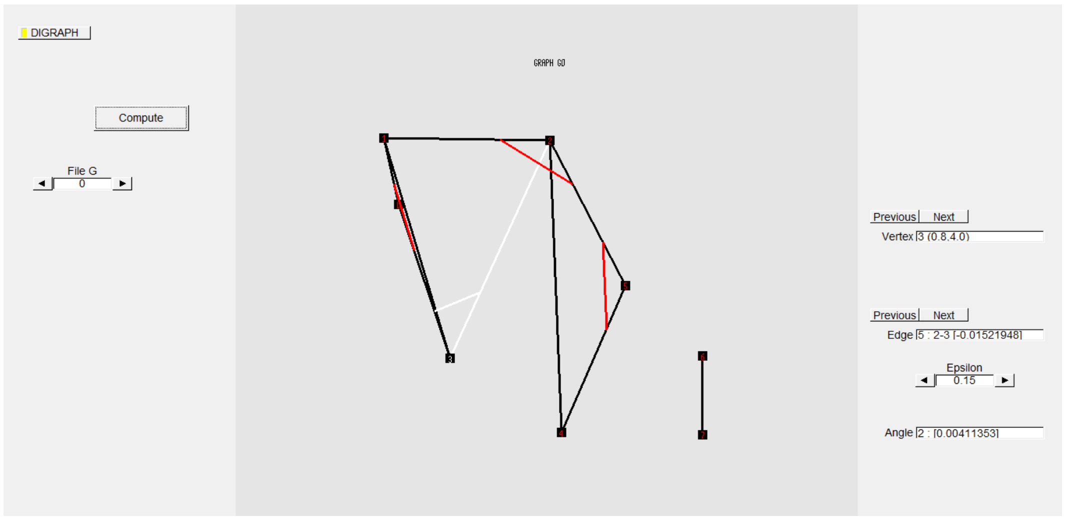
Vertex 3 (0.9,4.1)

Edge 5 : 2-3 [0.02961366]

Epsilon 0.10

Angle 2 : [-0.00869567]

Figure 5.13: $\epsilon = 0.10$.



File G 0

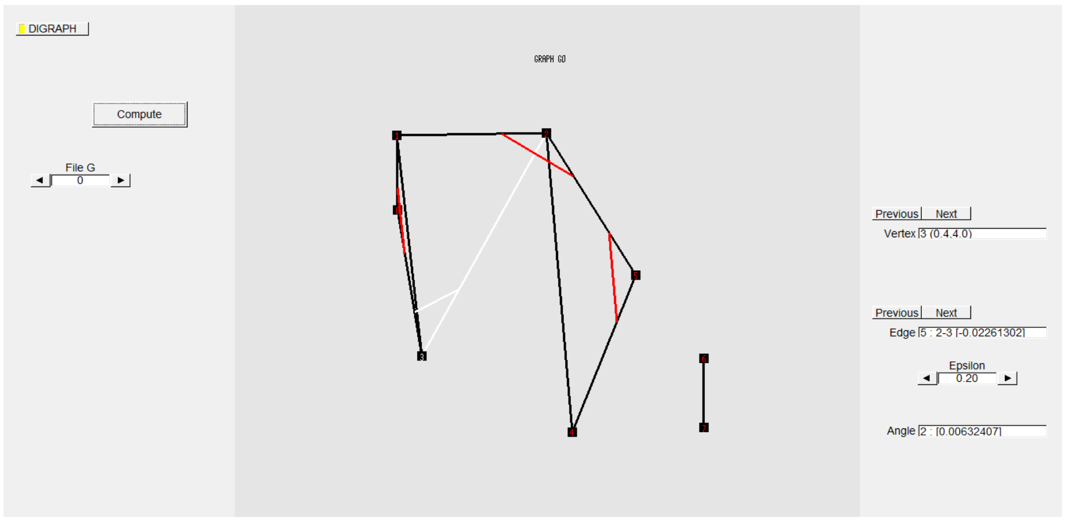
Vertex 3 (0.8,4.0)

Edge 5 : 2-3 [-0.01521948]

Epsilon 0.15

Angle 2 : [0.00411353]

Figure 5.14: $\epsilon = 0.15$.



File G 0

Vertex 3 (0.4,4.0)

Edge 5 : 2-3 [-0.02261302]

Epsilon 0.20

Angle 2 : [0.00632407]

Figure 5.15: $\epsilon = 0.20$.

Bibliography

[1] Pratik Agarwal, Wolfram Burgard, and Cyrill Stachniss. A survey of geodetic approaches to mapping and the relationship to graph-based slam. 2014.

[2] Anton Aubanell, Antoni Benseny, and Amadeu Delshams. *Útiles básicos del cálculo numérico*. Labor universitaria. Manuales. Servei de Publicacions de la Universitat Autònoma de Barcelona, 1993.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA; McGraw-Hill Book Co., Boston, MA, second edition, 2001.

[4] Reinhard Diestel. *Graph theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Berlin, fifth edition, 2018. Paperback edition of [MR3644391].

[5] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[6] Institut Cartogràfic i Geològic de Catalunya. Sobre les xarxes geodèsiques. `http://www.icgc.cat/ca/Administracio-i-empresa/Serveis/Posicionament/Senyals-geodesics2/Sobre-les-xarxes-geodesiques`.

[7] Sergei Izrailev. Package tictoc, version 1.0. `https://www.rdocumentation.org/packages/tictoc/versions/1.0/topics/tic`.

[8] Jordi Jover Molina. TFG: Anàlisi de components principals: estudi numèric i aplicacions. 2018. `http://hdl.handle.net/2445/122109`.

[9] Base v3.6.0 by R-core. R documentation: SVD. `https://www.rdocumentation.org/packages/base/versions/3.6.0/topics/svd`.