MASTER'S THESIS

Interuniversity Master in Statistics and Operations Research UPC-UB

Title: Multidimensional Scaling for Big Data

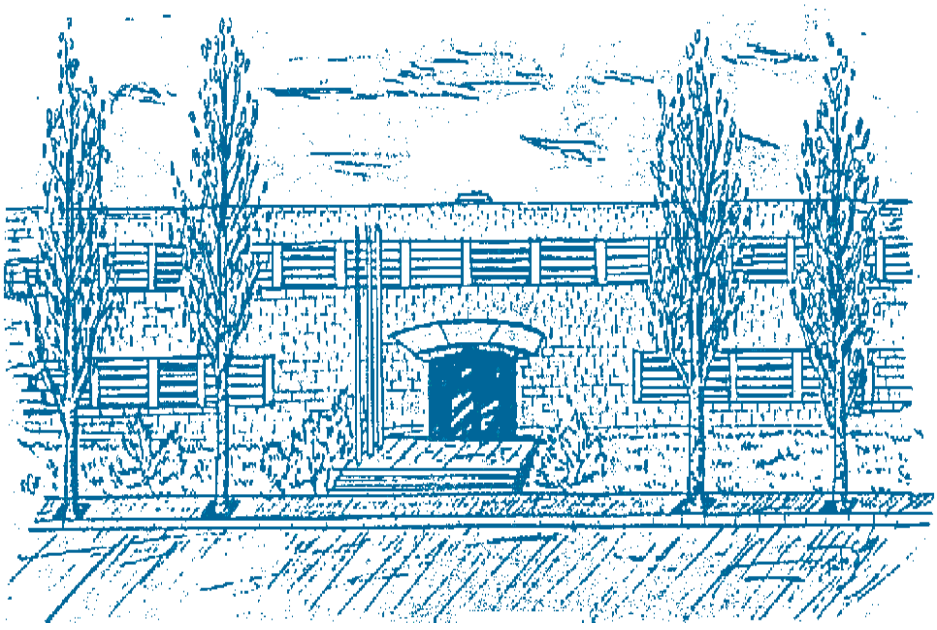
Author: Cristian Pachón García

Advisor: Pedro Delicado Useros

Department: Dept. d'Estadística i Investigació Operativa

University: Universitat Politècnica de Catalunya - Universitat de Barcelona

Academic year: 2018-2019



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT DE BARCELONA

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master's thesis

Multidimensional Scaling for Big Data

Cristian Pachón García

Advisor: Pedro Delicado Useros

Dept. d'Estadística i Investigació Operativa

Abstract

We present a set of algorithms for *Multidimensional Scaling* (MDS) to be used with large datasets. MDS is a statistic tool for reduction of dimensionality, using as input a distance matrix of dimensions $n \times n$. When n is large, classical algorithms suffer from computational problems and MDS configuration can not be obtained.

In this thesis we address these problems by means of three algorithms: *Divide and Conquer MDS, Fast MDS* and *MDS based on Gower interpolation*. The main idea of these methods is based on partitioning the dataset into small pieces, where classical methods can work.

In order to check the performance of the algorithms as well as to compare them, we do a simulation study. This study points out that *Fast MDS* and *MDS based on Gower interpolation* are appropriated to use when n is large. Although *Divide and Conquer MDS* is not as fast as the other two algorithms, it is the best method that captures the variance of the original data.

Contents

Chapter 1

Classical Multidimensional Scaling

1.1 Introduction to Multidimensional Scaling

Multidimensional Scaling (MDS) is a family of methods that represents measurements of dissimilarity (or similarity) among pairs of objects as Euclidean distances between points of a low-dimensional space. The data, for example, may be correlations among intelligence tests and the MDS representation is a plane that shows the tests as points. The graphical display of the correlations provided by MDS enables the data analyst to literally "look" at the data and to explore their structure visually. This often shows regularities that remain hidden when studying arrays of numbers.

Given a square matrix $\mathbf{D}$ $n \times n$, the goal of MDS is to obtain a configuration matrix $\mathbf{X}$ $n \times p$ with orthogonal columns that can be interpreted as the matrix of p variables for the n observations, where the Euclidean distance between the rows of $\mathbf{X}$ is approximately equal to $\mathbf{D}$. The columns of $\mathbf{X}$ are called *principal coordinates*.

This approach arises two questions: is it (always) possible to find this low-dimensional configuration $\mathbf{X}$? How is it obtained? In general, it is not possible to find a set of p variables that reproduces *exactly* the initial distance. However, it is possible to find a set of variables which distance is approximately the initial distance matrix $\mathbf{D}$.

As a classical example, consider the distances between European cities as in Table 1.1. One would like to get a representation in a 2-dimensional space such that the distances would be almost the same as in Table 1.1. The representation of these coordinates are displayed in Figure 1.1.

	Athens	Barcelona	Brussels	Calais	Cherbourg	$\cdots$
Athens	0	3313	2963	3175	3339	$\cdots$
Barcelona	3313	0	1318	1326	1294	$\cdots$
Brussels	2963	1318	0	204	583	$\cdots$
Calais	3175	1326	204	0	460	$\cdots$
Cherbourg	3339	1294	583	460	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1.1: Distances between European cities (just 5 of them are shown).

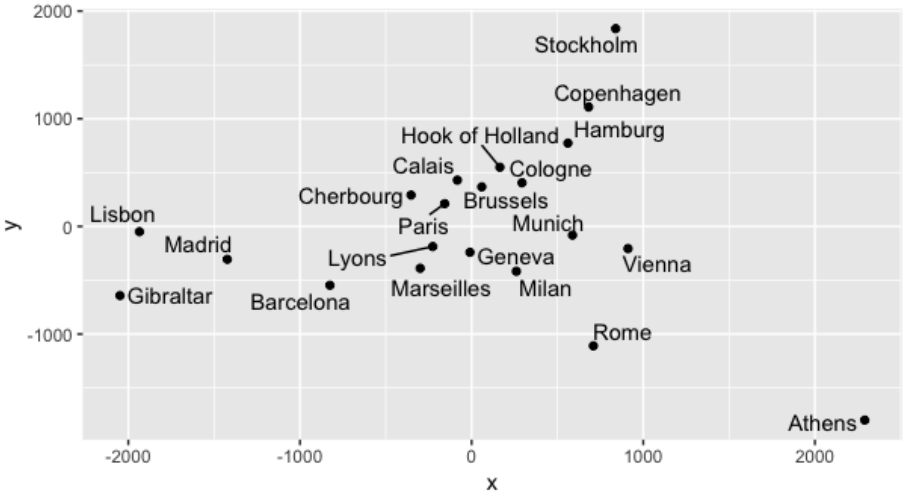


Figure 1.1: MDS on the European cities.

1.2 Principal coordinates

Given a matrix $\mathbf{X}$ $n \times p$, the matrix of n individuals over p variables, it is possible to obtain a new one with mean equal to 0 by column from the previous one:

$$\widetilde{\mathbf{X}} = \left(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}'\right)\mathbf{X} = \mathbf{P}\mathbf{X},$$

where

$$\mathbf{P} = \left(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}'\right).$$

This new matrix, $\widetilde{\mathbf{X}}$, has the same dimensions as the original one but its columns mean is $\mathbf{0}$. From this matrix, it is possible to build two square semi-positive definite matrices: the covariance matrix $\mathbf{S}$, defined as $\widetilde{\mathbf{X}}'\widetilde{\mathbf{X}}/n$ and the cross-products matrix $Q = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$. The last matrix can be interpreted as a similarity matrix between the n elements. The term ij is obtained as follows:

$$q_{ij} = \sum_{s=1}^{p} x_{is}x_{js} = \mathbf{x_i}'\mathbf{x_j} \tag{1.1}$$

where $\mathbf{x_i}'$ is the $i-th$ row from $\widetilde{\mathbf{X}}$.

Given the scalar product formula, $\mathbf{x_i}'\mathbf{x_j} =\mid \mathbf{x_i} \mid\mid \mathbf{x_i} \mid \cos\theta_{ij}$, if the elements i and j have similar coordinates, then $\cos\theta_{ij} \simeq 1$ and q_{ij} will be large. On the contrary, if the elements are very different, then $\cos\theta_{ij} \simeq 0$ and q_{ij} will be small. So, $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$ can be interpreted as the similarity matrix between the elements.

The distances between elements can be deduced from the similarity matrix. The Euclidean distance between two elements is calculated in the following way:

$$d_{ij}^2 = \sum_{s=1}^{p}(x_{is} - x_{js})^2 = \sum_{s=1}^{p} x_{is}^2 + \sum_{s=1}^{p} x_{js}^2 - 2\sum_{s=1}^{p} x_{is}x_{js}. \tag{1.2}$$

This expression can be obtained directly from the matrix $\mathbf{Q}$:

$$d_{ij}^2 = q_{ii} + q_{jj} - 2q_{ij}. \tag{1.3}$$

We have just seen that, given the matrix $\widetilde{\mathbf{X}}$, it is possible to get the similarity matrix $\mathbf{Q} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}'$ and from it, to get the distance matrix $\mathbf{D}$. Let diag($\mathbf{Q}$) be the vector that contains the diagonal terms of $\mathbf{Q}$ and $\mathbf{1}$ be the vector of ones, the matrix $\mathbf{D}$ is given by

$$\mathbf{D} = \text{diag}(\mathbf{Q})\mathbf{1}' + \mathbf{1}\,\text{diag}(\mathbf{Q})' - 2\mathbf{Q}.$$

The problem we are dealing with goes in the opposite direction. We want to rebuild $\widetilde{\mathbf{X}}$ from a square distance matrix $\mathbf{D}$, with elements d_{ij}^2. The first step is to obtain $\mathbf{Q}$ and afterwards, to get $\widetilde{\mathbf{X}}$. The theory needed to get the solution can be found in (Peña 2002). Here, we summarise it.

The first step is to find out a way to obtain the matrix $\mathbf{Q}$ given $\mathbf{D}$. We can assume without loss of generality that the mean of the variables is equal to 0. This is a consequence of the fact that the distance between two points remains the same if the variables are expressed in terms of the mean:

$$d_{ij}^2 = \sum_{s=1}^{p}(x_{is} - x_{js})^2 = \sum_{s=1}^{p}[(x_{is} - \overline{x_s}) - (x_{js} - \overline{x_s})]^2. \tag{1.4}$$

The previous condition means that we are looking for a matrix $\widetilde{\mathbf{X}}$ such that $\widetilde{\mathbf{X}}'\mathbf{1} = 0$. It also means that $\mathbf{Q1} = 0$, i.e, the sum of all the elements of a column of $\mathbf{Q}$ is 0. Since the matrix is symmetric, the previous condition should state for the rows as well.

To establish these constrains, we sum up (1.3) at row level:

$$\sum_{i=1}^{n} d_{ij}^2 = \sum_{i=1}^{n} q_{ii} + nq_{jj} = t + nq_{jj} \tag{1.5}$$

where $t = \sum_{i=1}^{n} q_{ii} = \text{Trace}(\mathbf{Q})$, and we have used that the condition $\mathbf{Q1} = 0$ implies $\sum_{i=1}^{n} q_{ij} = 0$. Summing up (1.3) at column level:

$$\sum_{j=1}^{n} d_{ij}^2 = t + nq_{ii}. \tag{1.6}$$

Summing up (1.5) we obtain:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 = 2nt \tag{1.7}$$

Replacing in (1.3) q_{jj} obtained in (1.5) and q_{ii} obtained in (1.6), we have the following expression:

$$d_{ij}^2 = \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 - \frac{t}{n} + \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2 - \frac{t}{n} - 2q_{ij} \tag{1.8}$$

Let $d_{i.}^2 = \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2$ and $d_{.j}^2 = \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2$ be the row-mean and column-mean of the elements of $\mathbf{D}$. Using (1.7), we have that

$$d_{ij}^2 = d_{i.}^2 + d_{.j}^2 - d_{..}^2 - 2q_{ij} \tag{1.9}$$

where $d_{..}$ is the mean of all the elements of $\mathbf{D}$, given by

$$d_{..}^2 = \frac{1}{n^2} \sum \sum d_{ij}^2.$$

Finally, from (1.9) we get the expression:

$$q_{ij} = -\frac{1}{2}(d_{ij}^2 - d_{i.}^2 - d_{.j}^2 + d_{..}^2). \tag{1.10}$$

The previous expression shows how to build the matrix of similarities $\mathbf{Q}$ from the distance matrix $\mathbf{D}$.

The next step is to obtain the matrix $\mathbf{X}$ given the matrix $\mathbf{Q}$. Let's assume that the similarity matrix is positive definite of range p. Therefore, it can be represented as

$$\mathbf{Q} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}',$$

where $\mathbf{V}$ is a $n \times p$ matrix that contains the eigenvectors with not nulls eigenvalues of $\mathbf{Q}$. $\mathbf{\Lambda}$ is a diagonal matrix $p \times p$ that contains the eigenvalues.

Re-writing the previous expression, we obtain

$$\mathbf{Q} = (\mathbf{V}\mathbf{\Lambda}^{1/2})(\mathbf{\Lambda}^{1/2}\mathbf{V}'). \tag{1.11}$$

Getting

$$\mathbf{Y} = \mathbf{V}\mathbf{\Lambda}^{1/2}.$$

We have obtained a matrix with dimensions $n \times p$ with p uncorrelated variables that reproduce the initial metric. It is important to notice that if one starts from $\mathbf{X}$ (i.e $\mathbf{X}$ is known) and calculates from these variables the distance matrix in (1.2) and after that it is applied the method explained, the matrix obtained is not the same as $\mathbf{X}$, but its principal components. This happens since the distance between the rows in a matrix does not change if:

- The row-mean values are modified by adding the same row vector to all the rows in $\mathbf{X}$.
- Rows are rotated, i.e, $\mathbf{X}$ is postmultiplied by an orthogonal matrix.

By (1.3), the distance is a function of the terms of the similarity matrix $\mathbf{Q}$ and this matrix is invariant given any rotation, reflection or translation of the variables

$$\mathbf{Q} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}'} = \widetilde{\mathbf{X}}\mathbf{A}\mathbf{A}'\widetilde{\mathbf{X}'}$$

for any orthogonal $\mathbf{A}$ matrix. The matrix $\mathbf{Q}$ only contains information about the space generated by the variables $\mathbf{X}$. Any rotation, reflection or translation preserves the distance. Therefore, the solution is not unique.

1.3 Building principal coordinates

Let $\mathbf{D}$ be a square distance matrix. The process to obtain the *principal coordinates* is as follows:

1. Build the matrix $\mathbf{Q} = -\frac{1}{2}\mathbf{PDP}$ of cross-products.

2. Obtain the eigenvalues of $\mathbf{Q}$. Take the r greatest eigenvalues. Since $\mathbf{P1} = 0$, where $\mathbf{1}$ is a vector of ones, range$(\mathbf{Q}) = n - 1$, being the vector $\mathbf{1}$ an eigenvector with eigenvalue 0.

3. Obtain the coordinates of the rows in the variables $\mathbf{v_i}\sqrt{\lambda_i}$, where λ_i is an eigenvalue of $\mathbf{Q}$ and $\mathbf{v_i}$ is the associated unitary eigenvector. This implies that $\mathbf{Q}$ is approximated by

$$\mathbf{Q} \approx (\mathbf{V_r}\mathbf{\Lambda}^{1/2})(\mathbf{\Lambda_r}^{1/2}\mathbf{V'_r}).$$

4. Take as coordinates of the points the following variables:

$$\mathbf{Y_r} = \mathbf{V_r}\mathbf{\Lambda_r}^{1/2}.$$

The method can also be applied if the initial information is not a distance matrix but a similarity matrix. A *similarity function* is characterized by the following properties (s_{ij} denotes the similarity between element i and j):

- $s_{ii} = 1$.
- $0 \leq s_{ij} \leq 1$.
- $s_{ij} = s_{ji}$.

If the initial information is $\mathbf{Q}$, a similarity matrix, then $q_{ii} = 1$, $q_{ij} = q_{ji}$ and $0 \leq q_{ij} \leq 1$. The associated distance matrix is

$$d_{ij}^2 = q_{ii} + q_{jj} - 2q_{qij} = 2(1 - q_{ij}),$$

and it is easy to see that $\sqrt{2(1 - q_{ij})}$ is a distance and it verifies the triangle inequality.

1.4 Procrustes transformation

As we have mentioned before, the MDS solution is not unique. One example of it is shown in Figure 1.2.

Since rotations, translations and reflections are distance-preserving functions, one can find two different MDS configurations for the same set of data. How is it possible to align both solutions? *Align both solutions* (or multiple ones) means to find a common coordinate system for all the solutions, i.e, let $\mathbf{MDS}_1$ and $\mathbf{MDS}_2$ be two MDS solutions of dimensions $n \times r$. We say they are aligned if the coordinates of row i are the same in both solutions:

$$mds_{i1}^1 = mds_{i1}^2, \ldots, mds_{ir}^1 = mds_{ir}^2$$

where mds_{ij}^k is the coordinates j for the individual i given the solution k, $j \in \{1, \ldots r\}$, $i \in \{1, \ldots, n\}$ and $k \in \{1, 2\}$.

This problem is solved by means of *Procrustes transformations*. The Procrustes problem is concerned with fitting a configuration (testee) to another (target) as closely as possible. In the simple case, both configurations have the same dimensionality and the same number of points, which can be brought into 1-1 correspondence. Under orthogonal

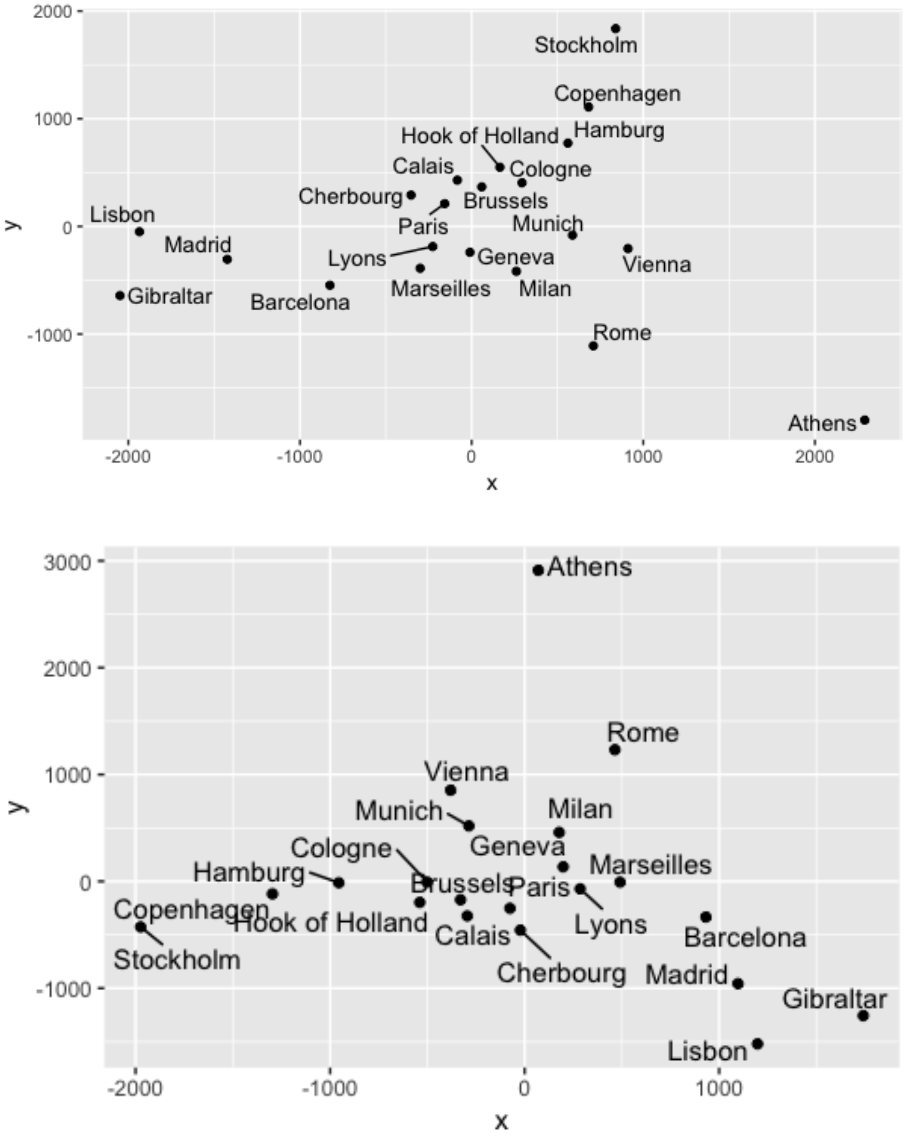


Figure 1.2: Two different MDS configurations for Figure 1.1.

transformations, the testee can be fitted to the target. In addition to such rigid motions, one may also allow for dilations and for shifts.

All the details are developed in Borg and Groenen (2005). This is out of the scope of this thesis. However, since it has been a repeatedly used tool, we briefly summarise it.

Let $\mathbf{A}$ and $\mathbf{B}$ be two different MDS configurations of dimensions $n \times t$ for the same set of data. Without loss of generality, let's assume that the target is $\mathbf{A}$ and the testee is $\mathbf{B}$. One wants to obtain $s \in \mathbb{R}$, $\mathbf{T} \in \mathrm{M}_{r\times r}(\mathbb{R})$ and $\mathbf{t} \in \mathbb{R}^r$ such that

$$\mathbf{A} = s\mathbf{B}\mathbf{T} + \mathbf{1}\mathbf{t}'$$

where $\mathbf{T}$ is an orthogonal matrix. As mentioned before, in Borg and Groenen (2005) are all the details needed to estimate these parameters.

1.5 Multidimensional Scaling with R

All the algorithms are coded in R. We use two packages for developing our MDS approaches:

- Package: stats. From this one we use the function cmdscale to do the MDS. The output of this function is a list of two elements:
  - The first r principal coordinates for the individuals, i.e, the low-dimensional configuration for the data.
  - All the eigenvalues found. If the dimensions of the initial dataset are $n \times k$, then there are n eigenvalues.
- Package: MCMCpack. From this one we use the function procrustes to do Procrustes transformation. The output of this function is a list of three elements:
  - The dilation coefficient s.
  - The orthogonal matrix $\mathbf{T}$.
  - The translation vector $\mathbf{t}$.

Chapter 2

Algorithms for Multidimensional Scaling with Big Data

2.1 Why is it needed?

In this chapter we present the algorithms developed so that MDS can be applied when we are dealing with large datasets. The first question one might ask is why we need them if there are already some implementations to obtain a MDS configuration. To answer this question, let's take a look at Figure 2.1 and Figure 2.2.

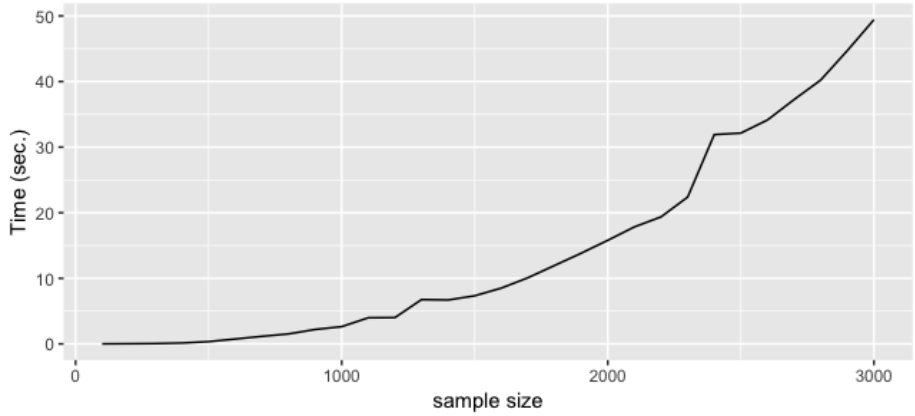


Figure 2.1: Elapsed time to compute MDS.

Figure 2.1 shows the time needed to compute MDS as a function of the sample size. As we can see, the time grows considerably as the sample size increases when using cmdscale function. Apart of the time issue, there is another one related to the memory needed to compute the distance matrix. Figure 2.2 points out that it is required at least 400MB of RAM memory to obtain the distance matrix when the dataset is close to 10^4 observations.

In order to solve these problems, we have considered to work on three algorithms:

- *Divide and Conquer MDS:* Before this thesis, *Pedro Delicado* did some work about MDS with large datasets and he already created a first approach, which is this one. The algorithm is based on the idea of dividing and conquering. Given a large dataset, it is divided into p partitions. After that, MDS is performed over all the

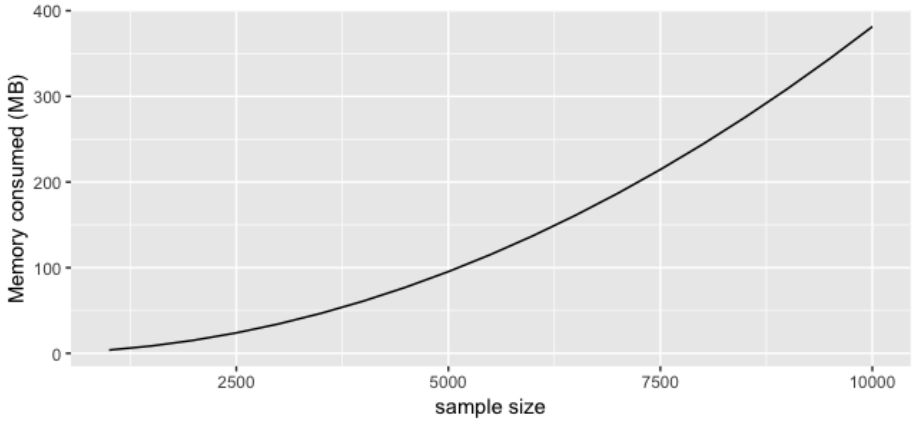


Figure 2.2: Memory consumed to compute the distance matrix.

partitions and, finally, the p solutions are stitched so that all the points lie on the same coordinate system.

- *Fast MDS:* during the phase of gathering information, we found an article that solved the problem of scalability (Tynia, Jinze, Leonard, and Wei 2006). The authors use a divide and conquer idea together with recursive programming.

- *MDS based on Gower interpolation:* this algorithm uses Gower interpolation formula, which allows to add a new set of points to an existing MDS configuration. For further details see, for instance, the Appendix of (Gower and Hand 1995).

In the next sections we provide a description of the algorithms. If further details about the implementation are needed, the code is provided in Appendix C.

2.2 Divide and Conquer MDS

2.2.1 Algorithm

- The first step is to divide the original dataset into p partitions: $\mathbf{X_1}, \ldots, \mathbf{X_p}$. The number of partitions, p, is also the number of steps needed to compute the algorithm. Each partition has the same number of rows.

- Calculate the MDS for the first partition: **MDS(1)**. This solution will be used as a guide to align the MDS configuration for the remaining partitions. We use a new variable, **cum-mds**, that will be growing as long as new partitions are used. Before adding a new MDS configuration, it is aligned and, after that, added.

- Define **cum-mds** equals to **MDS(1)** and start iterating until the last partition is reached.

- Given a step k, $1 < k \leq p$, partitions k and *k-1* are joint, i.e, $\mathbf{X_k} \cup \mathbf{X_{k-1}}$. MDS is calculated on this union, obtaining $\mathbf{MDS_{k,k-1}}$. In order to add the rows of the *k-th* partition to **cum-mds**, the following steps are performed:

  - Take the rows of the partition *k-1* from $\mathbf{MDS_{k,k-1}}$: $\mathbf{MDS_{k,k-1}}\Big|_{k-1}$.

– Take the rows of the partition *k-1* from **cum-mds**: $\textbf{cum-mds}\Big|_{k-1}$.

– Apply Procrustes to align both solutions. It means that a scalar number s, a vector $\mathbf{t}$ and an orthogonal matrix $\mathbf{T}$ are obtained so that

$$\textbf{cum-mds}\Big|_{k-1} \approx s\mathbf{MDS_{k,k-1}}\Big|_{k-1}\mathbf{T} + \mathbf{1t'}.$$

– Take the rows of the partition k from $\mathbf{MDS_{k,k-1}} : \mathbf{MDS_{k,k-1}}\Big|_{k}$.

– Use the previous Procrustes parameters to add the rows of $\mathbf{MDS_{k,k-1}}\Big|_{k}$ to **cum-mds**:

$$\textbf{cum-mds}_\mathbf{k} := s\mathbf{MDS_{k,k-1}}\Big|_{k}\mathbf{T} + \mathbf{1t'}.$$

– Add the previous dataset to **cum-mds**, i.e:

$$\textbf{cum-mds} = \textbf{cum-mds} \cup \textbf{cum-mds}_\mathbf{k}$$

As we have seen, the algorithm depends on p, the number of partitions. How many of them are needed? To answer this question, let $l \times l$ be the size of the largest matrix that allows to run MDS efficiently, i.e, in a reasonable amount of time. If n is the number of rows of $\mathbf{X}$, then p is $\frac{2n}{l}$.

Note that if the number of rows of the original dataset, $\mathbf{X}$, is such that allows to run classical MDS over $\mathbf{X}$ without partitioning it, then $p = 1$ and *Divide and Conquer MDS* is just classical MDS.

2.2.2 Some indicators about the performance of the algorithm

The aim of this section is to show some indicators about the performance of the algorithm. A deeper analysis is done in Chapter 3, where more details are provided.

We generate a matrix $\mathbf{X}$ with 3 independent *Normal* distributions ($\mu = 0$ and $\sigma = 1$) and 10^3 rows with l equals to 500. Afterwards, we run the algorithm. We require the algorithm to return 3 columns. So, a new matrix with 3 columns and 10^3 rows ($\mathbf{MDS_{Div}}$) has been obtained. Both matrices should be "equal" with an exception of either a rotation, translation or reflection, but not a dilation. We do not allow dilations to see that the distance is preserved.

To align the matrices we perform a Procrustes transformation, but setting the the dilation parameter (s) equals to 1. After that, we compare the three columns (we refer to the columns as *dimensions*). Figure 2.3, Figure 2.4 and Figure 2.5 show the dimension i of $\mathbf{X}$ against the dimension i of $\mathbf{MDS_{Div}}$, $i \in \{1, 2, 3\}$.

As we can see, the algorithm is able to capture the dimensions of the original matrix. We do not show cross-dimensions (i.e dimension i of $\mathbf{X}$ against dimension j of $\mathbf{MDS_{Div}}$ $i \neq j$), but Table 2.1 contains the cross-correlation matrix. The results show that dimension i of $\mathbf{MDS_{Div}}$ captures dimension i of $\mathbf{X}$ and just dimension i. So, it seems that the algorithm, for this particular case, has a good performance in terms of quality.

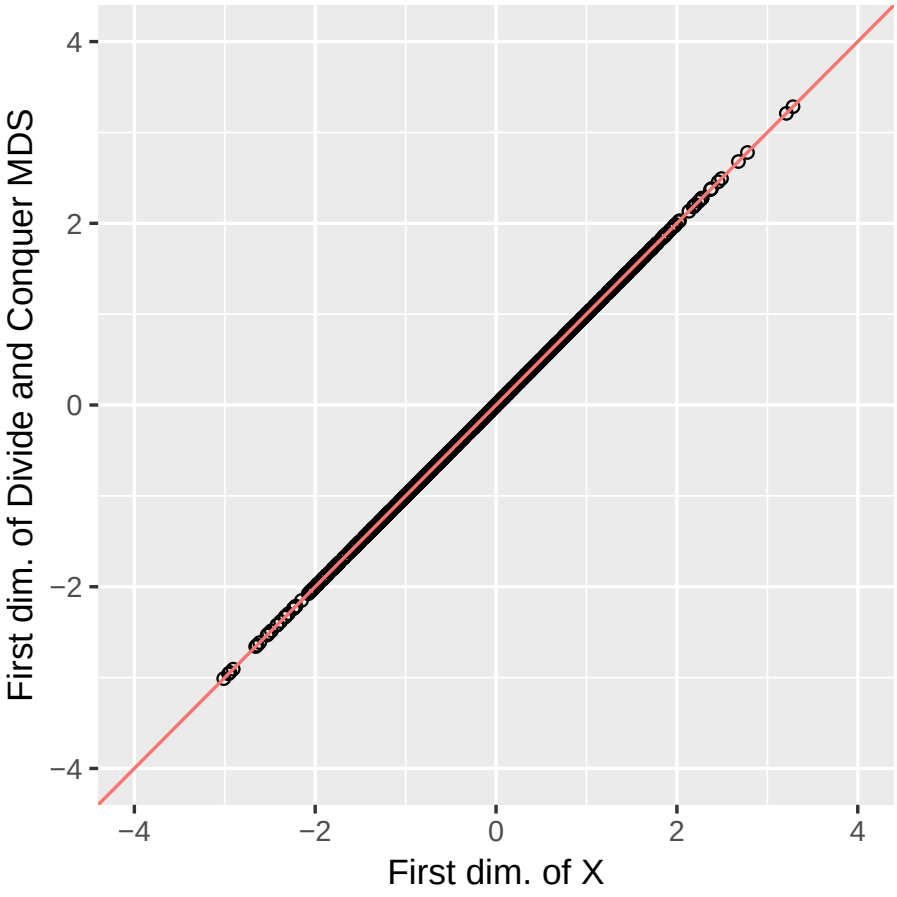


Figure 2.3: Dimension 1 $\mathbf{X}$ against dimension 1 of $\mathbf{MDS_{Div}}$. In red, the line $x = y$.

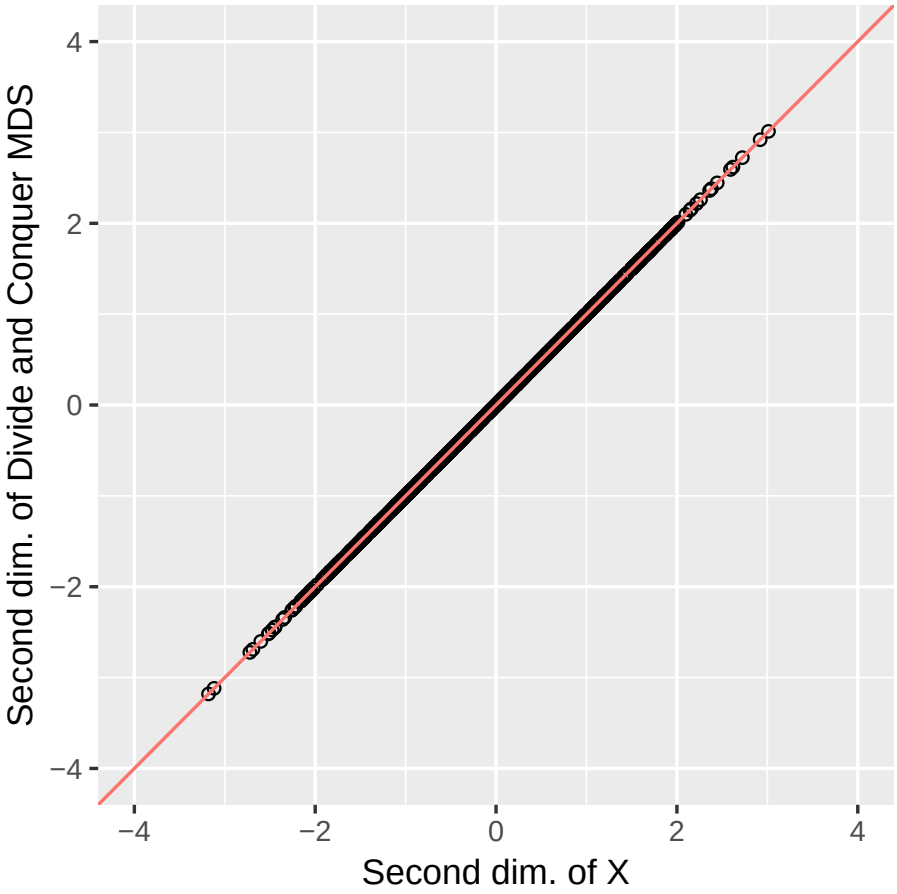


Figure 2.4: Dimension 2 $\mathbf{X}$ against dimension 2 of $\mathbf{MDS_{Div}}$. In red, the line $x = y$.

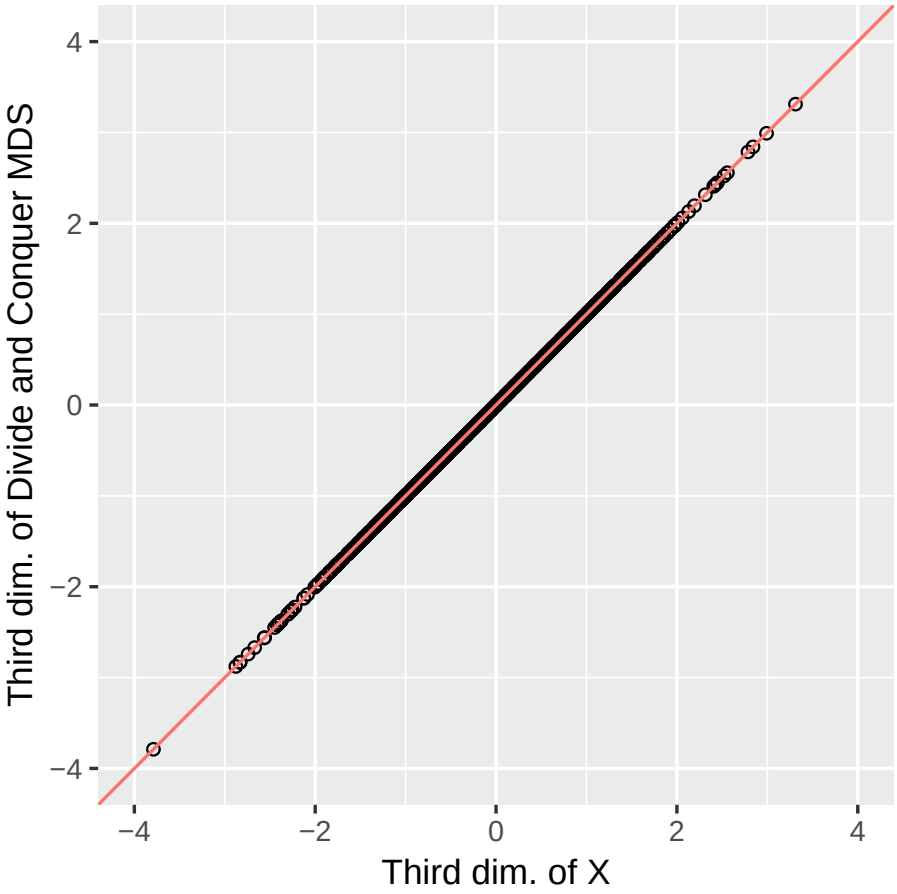


Figure 2.5: Dimension 3 of $\mathbf{X}$ against dimension 3 of $\mathbf{MDS_{Div}}$. In red, the line $x = y$.

	X_1	X_2	X_3
MDS_{Div1}	1	0.02	-0.04
MDS_{Div2}	0.02	1	0.02
MDS_{Div3}	-0.04	0.02	1

Table 2.1: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Div}}$.

2.3 Fast MDS

During the process of gathering information about work previously done around MDS with large datasets, we found that Tynia, Jinze, Leonard, and Wei (2006) already proposed an algorithm, they named it *Fast Multidimensional Scaling*.

2.3.1 Algorithm

- Divide $\mathbf{X}$ into $\mathbf{X_1}, \ldots, \mathbf{X_p}$. Each of them of the same dimensions.
- Obtain MDS configuration in the following way:
  - If $\mathbf{X}_i$ has such dimensions that allow to run classical MDS in a reasonable amount of time, compute MDS for each $\mathbf{X_i}$: $\mathbf{MDS_1}, \ldots, \mathbf{MDS_p}$.
  - Otherwise, for each $\mathbf{X_i}$ call recursively *Fast MDS* (at the end of this section we explain detailedly how the stop condition for the recursion is found).
- These individuals MDS solutions are stitched together by sampling s points (rows) from each submatrix $\mathbf{X_i}$ and putting them into an alignment matrix $\mathbf{M_{align}}$ of size $sp \times sp$. In principle, s should be at least 1 plus the estimated dimensionality of the dataset. In practice, it is oversampled by a factor of 2 or more.
- MDS is run on $\mathbf{M_{align}}$. After this, it is obtained **mMDS**. Given a sampled point, there are two solutions of MDS: one from $\mathbf{X_i}$ and another one from $\mathbf{M_{align}}$.
- The next step is to compute Procrustes transformation to line these two sets of solutions up in a common coordinate system:

  $$\mathbf{mMDS_i} = s_i \mathbf{dMDS_i T_i} + \mathbf{1 t_i}'$$

  where:
  - $\mathbf{dMDS_i}$ is $\mathbf{MDS_i}$ but taking into account just the subset of the sample points that belongs to partition i.
  - $\mathbf{mMDS_i}$ is **mMDS** but taking into account just the subset of the sample points that belongs to partition i.
- After doing the previous part, we obtain a set of p Procrustes parameters $(s_i, \mathbf{T_i}, \mathbf{t_i})$. So, the next step is to apply this set of parameters to each $\mathbf{MDS_i}$, i.e,

  $$\mathbf{MDS_i}^a := s_i \mathbf{MDS_i T_i} + \mathbf{1 t_i'}.$$

- The last step is to join $\mathbf{MDS_1}^a, \ldots, \mathbf{MDS_p}^a$ all together, i.e,

$$\mathbf{MDS_X} := \mathbf{MDS_1}^a \cup \cdots \cup \mathbf{MDS_p}^a.$$

The process of splitting the dataset is applied recursively until the size of $\mathbf{X_i}$ is optimal to run MDS on. The stop condition is found as follows: let $l \times l$ be the size of the largest matrix that allow MDS to be executed efficiently, i.e, in a reasonable amount of time. There are two issues that impact the performance of the algorithm: the size of each submatrix after subdivision and the number of submatrices, p, that are stitched together at each step. Ideally, the size of each submatrix after division should be as large as possible without exceeding l. By the same token, the size of $\mathbf{M_{align}}$ should be bounded by l. The number of submatrices to be stitched together, p, should be the largest number such that $sp \leq l$.

As in the previous algorithm, if the number of rows of the original dataset, $\mathbf{X}$, is such that allows to run classical MDS over $\mathbf{X}$ without partitioning it, then $p = 1$ and *Fast MDS* is just the classical MDS.

2.3.2 Some indicators about the performance of the algorithm

As we have done in Section 2.2.2, we present some visual results of this algorithm. The data used are the same as in Section 2.2.2. We call $\mathbf{MDS_{Fast}}$ the result that provides the previous algorithm.

Figure 2.6, Figure 2.7 and Figure 2.8 show that, for this particular case, the algorithm captures quite well the dimensions of the original data, providing a good performance. In addition, dimension i of $\mathbf{MDS_{Fast}}$ fits perfectly the same dimensions i of $\mathbf{X}$ and just this one, as we can see in Table 2.2.

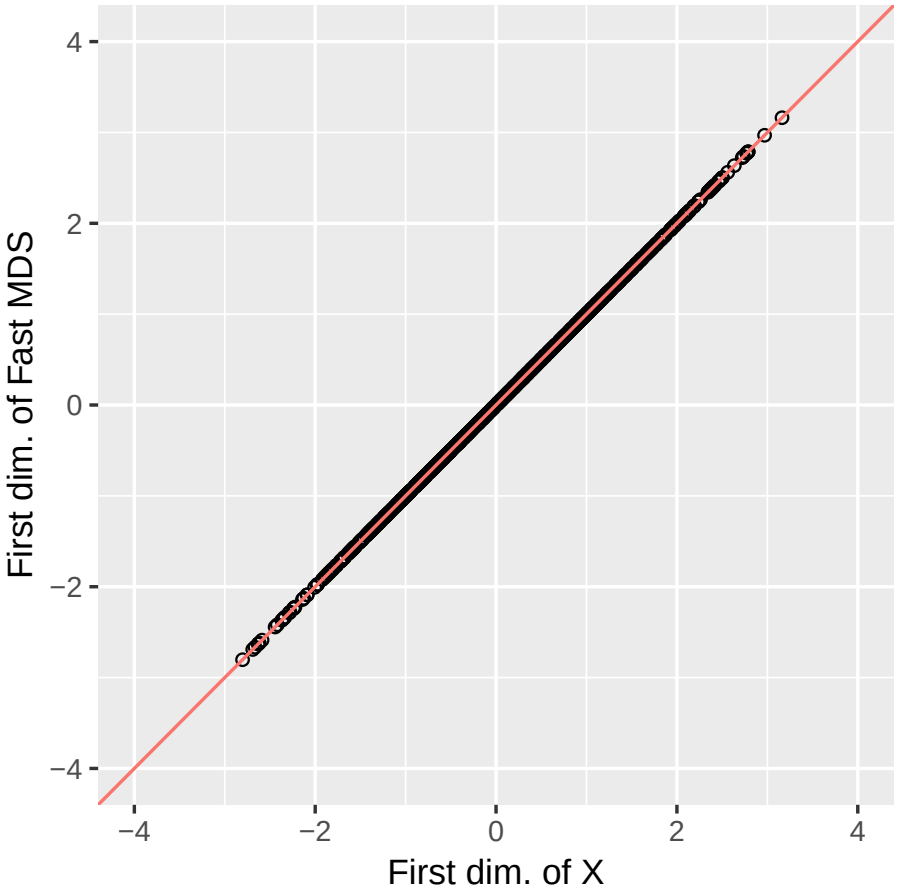


Figure 2.6: Dimension 1 of $\mathbf{X}$ against dimensions 1 of $\mathbf{MDS_{Fast}}$. In red, the line $x = y$.

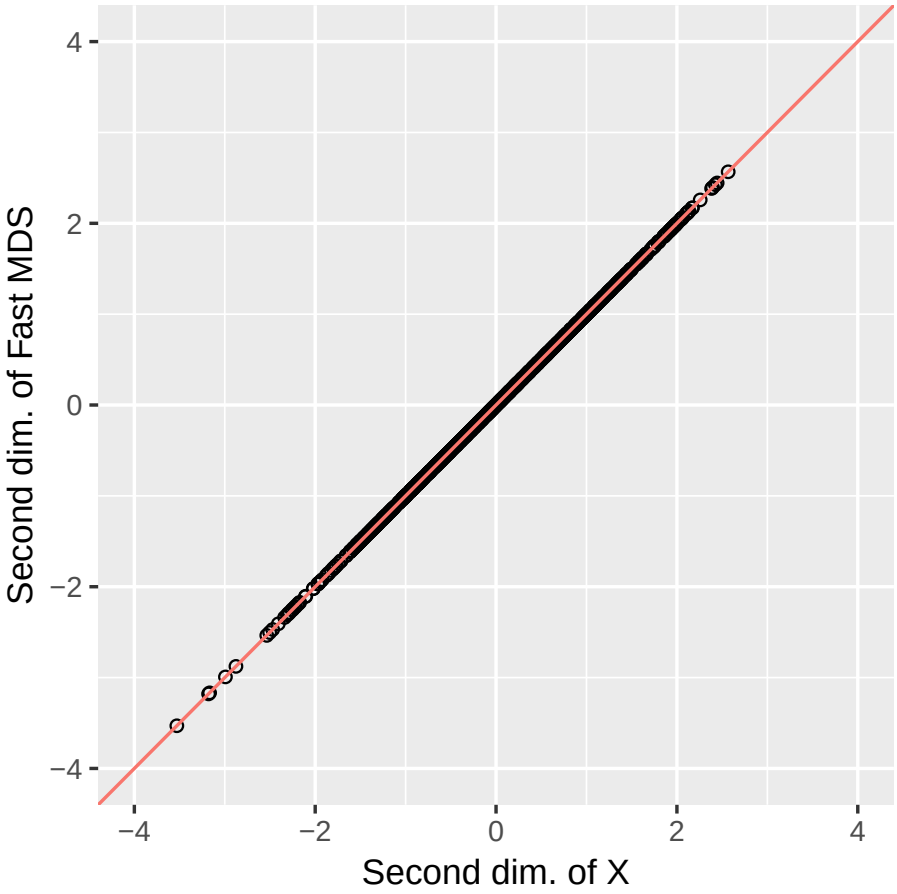


Figure 2.7: Dimension 2 of $\mathbf{X}$ against dimensions 2 of $\mathbf{MDS_{Fast}}$. In red, the line $x = y$.

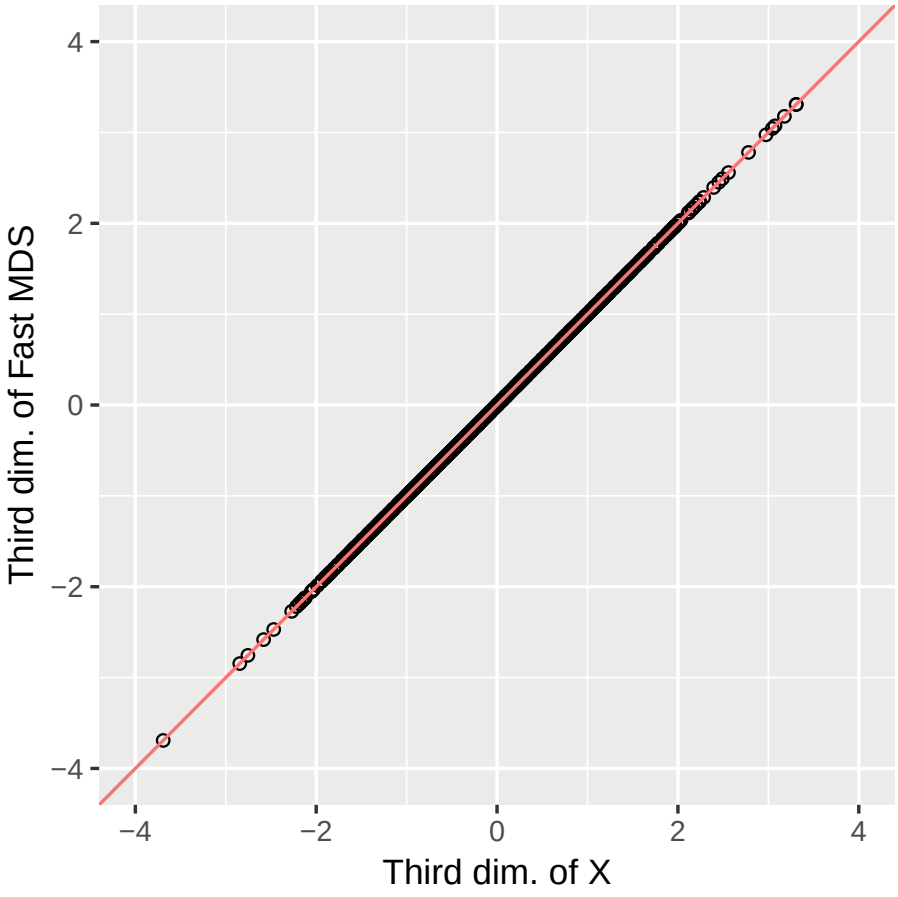


Figure 2.8: Dimension 3 of $\mathbf{X}$ against dimensions 3 of $\mathbf{MDS_{Fast}}$. In red, the line $x = y$.

	X_1	X_2	X_3
MDS_{Fast1}	1	0.02	0
MDS_{Fast2}	0.02	1	0.02
MDS_{Fast3}	0	0.02	1

Table 2.2: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Fast}}$.

2.4 MDS based on Gower interpolation

Gower interpolation formula (see Appendix of (Gower and Hand 1995)) allows to add a new set of points to a given MDS configuration. Given a matrix $\mathbf{X}$ $n \times p$, a MDS configuration for this matrix of dimension $n \times c$ and a matrix $\mathbf{X_{new}}$ $m \times p$, one wants to add these new m rows to the existing MDS configuration. So, after adding this new rows, the MDS configuration will have $n+m$ rows and c columns. We briefly summarise how to do so:

- Obtain $\mathbf{J} = \mathbf{I_n} - \frac{1}{n}\mathbf{1}\mathbf{1}'$, where $\mathbf{I_n}$ is the identity matrix $n \times n$.
- Given the distance matrix $\mathbf{D}$ of the rows of $\mathbf{X}$, calculate $\mathbf{\Delta} = (\delta_{ij}^2)$. Note that $\mathbf{\Delta}$ is of size $n \times n$.
- Calculate $\mathbf{G} = -\frac{1}{2}\mathbf{J}\mathbf{\Delta}\mathbf{J}'$.
- Let $\mathbf{g}$ be the diagonal of $\mathbf{G}$, i.e, $\mathbf{g} = \text{diag}(\mathbf{G})$. We treat $\mathbf{g}$ as a vector.
- Let $\mathbf{A}$ be the distance matrix between the rows of $\mathbf{X}$ and the rows of $\mathbf{X_{new}}$. $\mathbf{A}$ has dimensions $m \times n$. Let $\mathbf{A}^2$ be the matrix of the square elements of $\mathbf{A}$, i.e, $\mathbf{A}^2 = (a_{ij}^2)$.
- Let $\mathbf{M}$ and $\mathbf{S}$ be the MDS for $\mathbf{X}$ and the variance-covariance matrix of the c columns of $\mathbf{M}$.
- The interpolated coordinates for the new m observations are given by

$$\frac{1}{2n}(\mathbf{1}\mathbf{g}' - \mathbf{A}^2)\mathbf{M}\mathbf{S}^{-1}. \tag{2.1}$$

The resulting MDS for the m observations of $\mathbf{X_{new}}$ is in the same coordinate system as $\mathbf{M}$. So, here it is not needed to do any Procrustes transformation.

2.4.1 Algorithm

- Divide $\mathbf{X}$ into p partitions $\mathbf{X_1}, \ldots, \mathbf{X_p}$ of the same dimensions.
- If $p = 1$, classical MDS is run over $\mathbf{X}$.
- Otherwise, we use the procedure explained above, being $\mathbf{X} := \mathbf{X_1}$ and $\mathbf{X_{new}} := \mathbf{X_k}$, $k \in \{2, \ldots, p\}$.
- Calculate $\mathbf{J}$, $\mathbf{\Delta}$, $\mathbf{G}$, $\mathbf{g}$, $\mathbf{A}$, $\mathbf{M}$ and $\mathbf{S}$ according to the above formulas.
- Obtain MDS for the first partition $\mathbf{X_1}$.

- Given a partition $1 < k \leq p$, do the following steps to get the related MDS:
  - Calculate the distance matrix between the rows of $\mathbf{X_1}$ and $\mathbf{X_k}$ and calculate the square of each element of this matrix. Let $\mathbf{A}^2$ be this matrix (same as above).
  - Use Gower interpolation formula (2.1) to obtain MDS for partition k.
  - Accumulate this solution.

As in the previous two algorithms, there is a key parameter to choose: p, the number of partitions. For this algorithm, p is set in the following way. Let $l \times l$ be the size of the largest distance matrix that a computer can calculate efficiently, i.e, in a reasonable amount of time. The value of p is set as n/p.

2.4.2 Some indicators about the performance of the algorithm

We repeat the same as in Section 2.2.2. Figure 2.9, Figure 2.10, Figure 2.11 and Table 2.3 show that the algorithm, for this particular case, captures quite well the dimensions of the original data, providing a good performance.

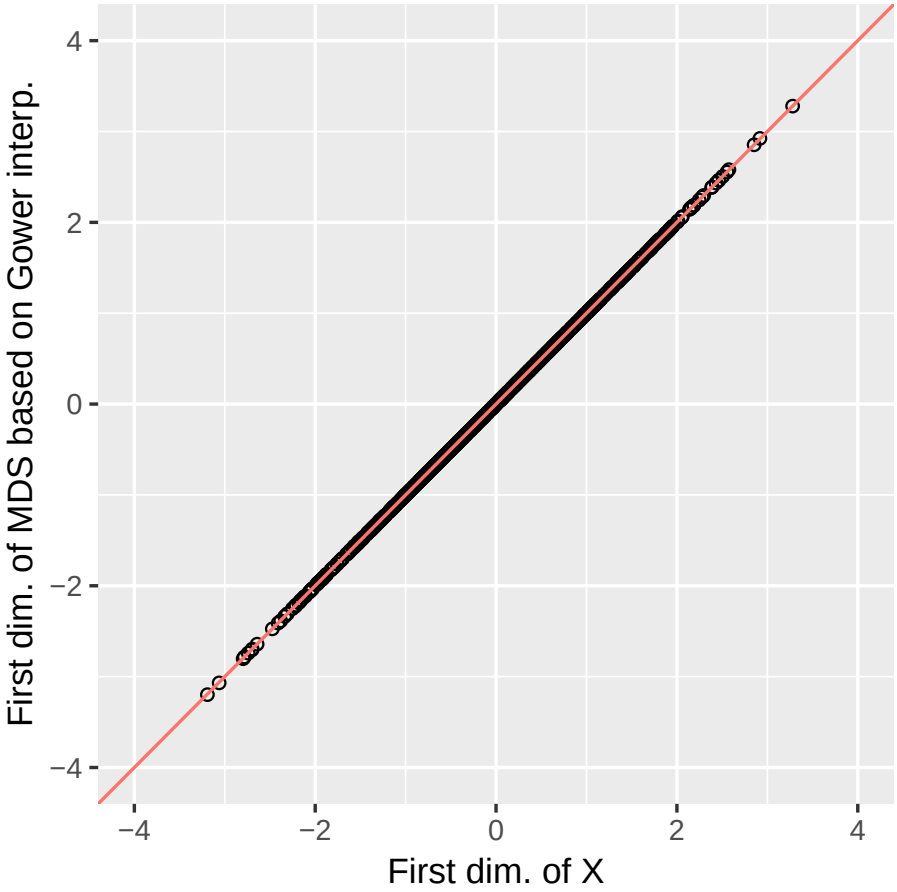


Figure 2.9: Dimension 1 of $\mathbf{X}$ against dimension 1 of $\mathbf{MDS_{Gower}}$. In red, the line $x = y$.

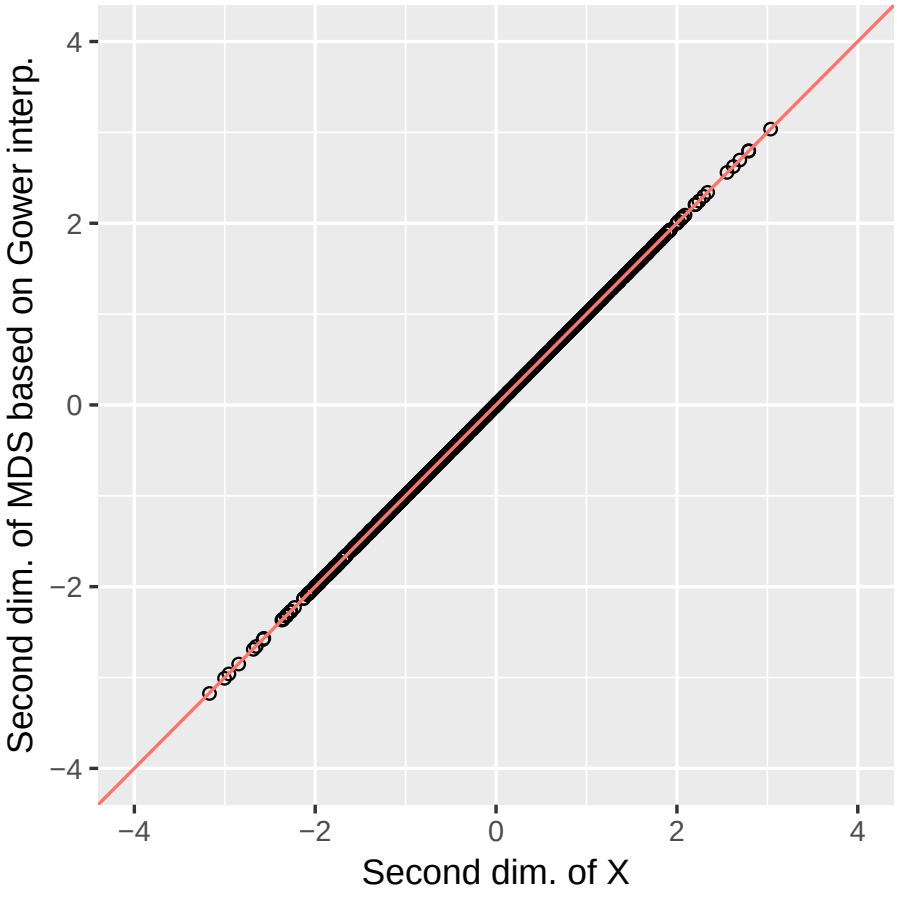


Figure 2.10: Dimension 2 of $\mathbf{X}$ against dimension 2 of $\mathbf{MDS_{Gower}}$. In red, the line $x = y$.

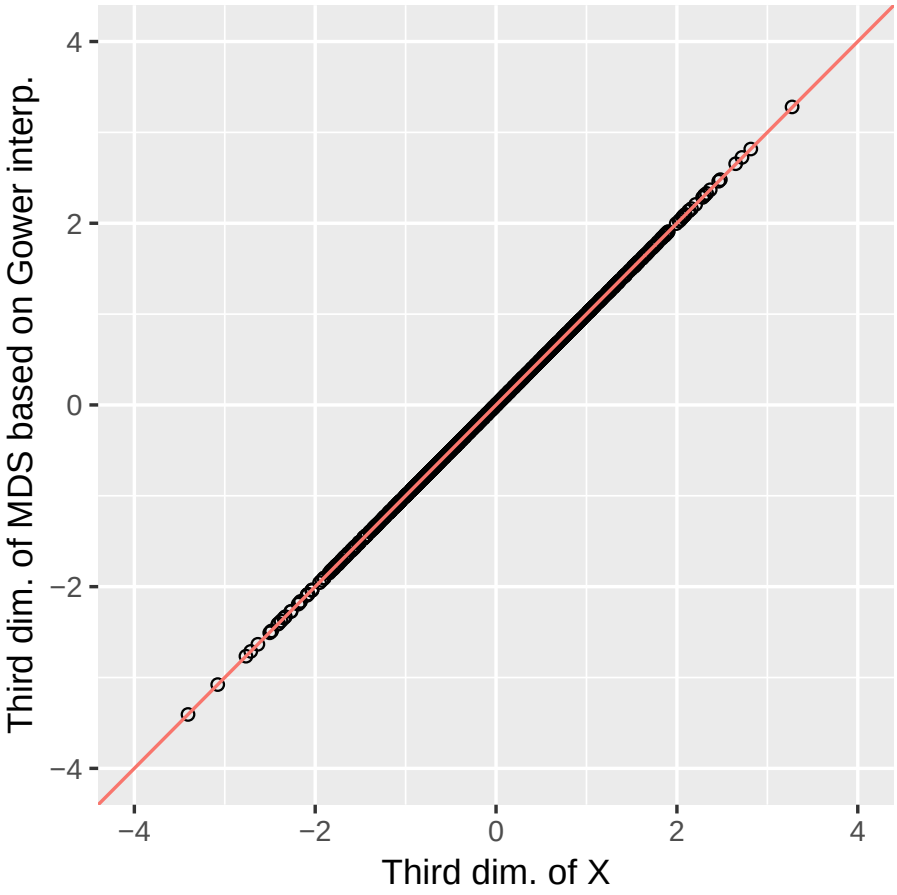


Figure 2.11: Dimension 3 of $\mathbf{X}$ against dimension 3 of $\mathbf{MDS_{Gower}}$. In red, the line $x = y$.

	X_1	X_2	X_3
MDS_{Gower1}	1	0	-0.04
MDS_{Gower2}	0	1	-0.0
MDS_{Gower3}	-0.04	-0.03	1

Table 2.3: Cross-correlation of $\mathbf{X}$ and $\mathbf{MDS_{Gower}}$.

2.5 Output of the algorithms

The three algorithms have the same type of output. It consists on a list of two parameters.

The first parameter is the MDS configuration calculated by the algorithm. It is a matrix of n rows and c columns, where n is the number of rows of the input data and c is the number of dimensions the user has required.

The second parameter is a list of eigenvalues. This list is built as follows:

- All the algorithms divide the initial data into a set of p partitions.
- Given a partition i, a distance matrix of dimensions $m_i \times m_i$ is calculated: $\mathbf{D_i}$.
- Over $\mathbf{D_i}$ a singular value decomposition is performed, providing a list of length m_i that contains all the eigenvalues of the previous decomposition: list_i.
- Let $\text{norm\_eigenvalues}_i$ be list_i/m_i, i.e, each eigenvalue is divided by the number of rows of $\mathbf{D_i}$.
- The algorithms return $\text{norm\_eigenvalues}_1 \cup \cdots \cup \text{norm\_eigenvalues}_p$. We refer to this union as the *normalized eigenvalues*.

2.6 Comparison of the algorithms

The three previous algorithms share the same goal: obtaining a MDS configuration for a given large dataset. However, there are some differences between the approaches that impact the performance of the algorithms. The main differences between them are:

- *Divide and Conquer MDS* uses a guide (the first subset, $\mathbf{X_1}$) to align the solutions as well as it uses the whole partition $\mathbf{X_i}$ to find Procrustes parameters. However, *Fast MDS* does not use a guide an it uses a set of subsamples to find Procrustes parameters.
- *Fast MDS* is based on recursive programming. It divides until a manageable dimensionality is found. However, *Divide and Conquer MDS* finds the number of partitions without applying recursive programming.
- *MDS based on Gower interpolation* does not need any Procrustes transformation.

The fact that we found three algorithms to compute MDS possesses some questions that need to be answered:

- Are these algorithms able to capture the data dimensionality as good as classical MDS does?

- Which is the fastest method?
- Can they deal with large datasets in a reasonable amount of time?
- How are they performing when dealing with large data sets?

All these questions are answered in Chapter 3.

Chapter 3

Simulation study

3.1 Design of the simulation

Given the three algorithms, we would like to explore their performance. There are two issues to study:

- Performance in terms of results quality: are they able to capture the right data dimensionality?
- Performance in terms of time: are they " fast" enough? Which one is the fastest?

To test the algorithms under different conditions, a simulation study has been carried out. The scenarios are obtained as combinations of:

- *Sample sizes*: we use different sample sizes, combining small datasets and large ones. A total of six sample sizes are used, which are:
  - Small sample sizes: $10^3, 3 \cdot 10^3, 5 \cdot 10^3$ and 10^4.
  - Large sample sizes: 10^5 and 10^6.
- *Data dimensions*: we generate a matrix with two different number of columns: 10 and 100.
- *Main dimensions*: given $\mathbf{X}$ $n \times k$, where $n \in \{10^3, 3 \cdot 10^3, 5 \cdot 10^3, 10^4, 10^5, 10^6\}$ and $k \in \{10, 100\}$, it is postmultiplied by a diagonal matrix that contains k values, $\lambda_1, \ldots, \lambda_k$. The first values are much higher than the rest. The idea of this is to see if the algorithms are able to capture the main dimensions of the original dataset, i.e, the columns with the highest variance. We set 5 combinations for this variable, which are:
  - All the columns with the same values of λ: $\lambda_1 = \cdots = \lambda_k = 1$.
  - One main dimension with $\lambda_1 = 15$ and $\lambda_2 = \cdots = \lambda_k = 1$.
  - Two main dimensions of the same value λ: $\lambda_1 = \lambda_2 = 15$ and $\lambda_3 = \cdots = \lambda_k = 1$.
  - Two main dimensions of different values λ: $\lambda_1 = 15$, $\lambda_2 = 10$ and $\lambda_3 = \cdots = \lambda_k = 1$.

  - Four main dimensions of the same value λ: $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 15$ and $\lambda_5 = \cdots = \lambda_k = 1$.

- As a probabilistic model, we use a Normal distribution with $\mu = 0$ and $\sigma = 1$. With this distribution, we generate a matrix of n observations and k columns, being the k columns independent. After generating the dataset $\mathbf{X}$, it is postmultiplied by the diagonal matrix that contains the values of λ's.

There is a total of 60 scenarios to simulate. Given a scenario, it is replicated 100 times. For every simulation, it is generated a dataset (according to the scenario), and all the algorithms are run using this dataset. So, a total of 6000 simulations are carried out.

Tables 3.1 and 3.2 show the configuration of each scenario, being Table 3.1 the configurations for small sample sizes and Table 3.2 the configurations for for large sample sizes. Given a scenario, *scenario_id* identifies it. We refer to a scenario by its *scenario_id*.

scenario_id	sample_size	n_dimensions	value_primary_dimensions
1	10^3	10	-
2	10^3	100	-
3	10^3	10	15
4	10^3	100	15
5	10^3	10	15, 15
6	10^3	100	15, 15
7	10^3	10	15, 10
8	10^3	100	15, 10
9	10^3	10	15, 15, 15, 15
10	10^3	100	15, 15, 15, 15
11	$3 \cdot 10^3$	10	-
12	$3 \cdot 10^3$	100	-
13	$3 \cdot 10^3$	10	15
14	$3 \cdot 10^3$	100	15
15	$3 \cdot 10^3$	10	15, 15
16	$3 \cdot 10^3$	100	15, 15
17	$3 \cdot 10^3$	10	15, 10
18	$3 \cdot 10^3$	100	15, 10
19	$3 \cdot 10^3$	10	15, 15, 15, 15
20	$3 \cdot 10^3$	100	15, 15, 15, 15
21	$5 \cdot 10^3$	10	-
22	$5 \cdot 10^3$	100	-
23	$5 \cdot 10^3$	10	15
24	$5 \cdot 10^3$	100	15
25	$5 \cdot 10^3$	10	15, 15
26	$5 \cdot 10^3$	100	15, 15
27	$5 \cdot 10^3$	10	15, 10
28	$5 \cdot 10^3$	100	15, 10
29	$5 \cdot 10^3$	10	15, 15, 15, 15
30	$5 \cdot 10^3$	100	15, 15, 15, 15
31	10^4	10	-
32	10^4	100	-
33	10^4	10	15
34	10^4	100	15
35	10^4	10	15, 15
36	10^4	100	15, 15
37	10^4	10	15, 10
38	10^4	100	15, 10
39	10^4	10	15, 15, 15, 15
40	10^4	100	15, 15, 15, 15

Table 3.1: Scenarios simulated for small sample sizes.

scenario_id	sample_size	n_dimensions	value_primary_dimensions
41	10^5	10	-
42	10^5	100	-
43	10^5	10	15
44	10^5	100	15
45	10^5	10	15, 15
46	10^5	100	15, 15
47	10^5	10	15, 10
48	10^5	100	15, 10
49	10^5	10	15, 15, 15, 15
50	10^5	100	15, 15, 15, 15
51	10^6	10	-
52	10^6	100	-
53	10^6	10	15
54	10^6	100	15
55	10^6	10	15, 15
56	10^6	100	15, 15
57	10^6	10	15, 10
58	10^6	100	15, 10
59	10^6	10	15, 15, 15, 15
60	10^6	100	15, 15, 15, 15

Table 3.2: Scenarios simulated for large sample sizes.

Note that scenarios 1, 2, 11, 12, 21, 22, 31, 32, 41, 42, 51, 52 are pure noise. We refer to them as *noisy scenarios*.

Given a scenario, the steps done to calculate and to store all the data needed are:

1. Generate the dataset $\mathbf{X}$ according to the scenario.
2. For each algorithm, we do the following steps:
   (a) Run the algorithm and get MDS configuration for the algorithm ($\mathbf{MDS_{alg}}$).
   (b) Get the elapsed time to compute MDS configuration and store it.
   (c) Get *normalized eigenvalues* and store them.
   (d) Align $\mathbf{MDS_{alg}}$ and $\mathbf{X}$ using Procrustes.
   (e) Get the correlation coefficients between the main dimensions of $\mathbf{MDS_{alg}}$ and $\mathbf{X}$ and store them.

There are some important details that affect the results of the simulations, which are:

- When running the algorithms, we ask for as many columns as the original data has, i.e, k. Therefore, the low-dimensional space has the same dimension as the original dataset.
- For the *normalized eigenvalues*, we just store 6 eigenvalues instead of the full list of eigenvalues (otherwise we would store n eigenvalues, which is memory consuming).
- For Procrustes we dot not allow dilations, otherwise distance could not be preserved. In addition, we do not use all the columns to do the alignment, we select the main dimensions. If there is not any main dimension, i.e it is one of the *noisy scenarios*, we just select 4 columns.
- To avoid memory problems with the alignment when n is greater or equal to 10^5, Procrustes is done in the following way:
  1. Create p partitions of $\mathbf{X}$ and the result of a given MDS algorithm ($\mathbf{MDS_{alg}}$). Both sets of partitions contain exactly the same observations.
  2. For each partition get Procrustes parameters without dilations.
  3. Accumulate the parameters iteration after iteration. So, at the end, we obtain $\mathbf{R} = \sum_{i=1}^{p} \mathbf{R_i}/p$ and $\mathbf{t} = \sum_{i=1}^{p} \mathbf{t_i}/p$.
  4. Apply these parameters to $\mathbf{MDS_{alg}}$ so that $\mathbf{X}$ and $\mathbf{MDS_{alg}}$ are in the same coordinate system and they can be compared, i.e

$$\mathbf{X_{Procrustes}} = \mathbf{XR} + \mathbf{1t}'.$$

Note that the original dataset, $\mathbf{X}$, is always available and it is already the MDS configuration, since we simulate independent columns with mean value equals to 0. Therefore, even though n is so large that MDS can not be calculated using classical methods, we always have the solution that we would obtain if running classical MDS were possible. Therefore, we can always compare the MDS provided by the algorithms ($\mathbf{MDS_{alg}}$) with the original dataset ($\mathbf{X}$).

In order to test the results quality of the algorithms as well as the time needed to compute the MDS configuration, some metrics are calculated. These metrics are the following ones:

- Performance of results quality: two metrics are calculated, which are:
  - Correlation between the main dimensions of the data and the main dimensions after applying the algorithms. We get the diagonal of the correlation matrix, i.e, the correlation between dimension i of the data and the dimension i of the algorithm.
  - *Normalized eigenvalues* as an approximation of the standard deviation of the variables of $\mathbf{X}$.
- Elapsed Time to get the MDS configuration: Given an algorithm, we compute and store the elapsed time to get the corresponding MDS configuration.

We do it in this way because we want to check some hypothesis. We expect the three algorithms to behave "correctly". By "correctly" we mean that the behavior should be similar to those obtained as if classical MDS were run. Therefore, we expect that the correlation between the main dimensions of the data and the main dimensions of the MDS of each algorithm is close to 1.

In addition, the variance of the original data should be captured. So, given the highest *normalized eigenvalues*, we expect that its square root is approximately 15 or 10 when the scenarios are not the *noisy scenarios*.

For the time of the algorithms, we have done some analysis and it seems that *MDS based on Gower interpolation* is the fastest one. So, proper tests will be done.

The algorithms have as input values a set of variables. The input matrix is already explained, but there is another parameter that has been used in the description of the algorithms (see Chapter 2): l. The meaning of l is a little bit different in each algorithm, but for simplicity we set this value equals to 500.

Fast MDS has an extra parameter: the amplification parameter. Tynia, Jinze, Leonard, and Wei (2006) use a value of 3 to test the algorithm. We use the same value. So, for each partition, it is taken 30 (when the original matrix has 10 columns) or 300 (when the original matrix has 100 columns) points for every partition to build $\mathbf{M_{align}}$.

Since a total of 6000 simulations are performed and some of them include large datasets, we use *Amazon Web Services* (AWS) to carry out the simulations. 10 servers of the same type are used: *c5n.4xlarge*. It has 16 cores and 42 GB of RAM memory.

3.2 Correlation coefficients

In this section we provide the simulations results for the correlation coefficients. Given a scenario and its dataset $\mathbf{X}$ of size $n \times k$, the correlation matrix between the main dimensions of $\mathbf{X}$ and the main dimensions of $\mathbf{MDS_{alg}}$ is computed. We are interested in the diagonal of the correlation matrix, expecting that the values are close to 1.

The length of the diagonal correlation matrix depends on the scenarios. It can be:

- 1, when $\lambda_1 = 15$ and $\lambda_2 = \cdots = \lambda_k = 1$.

- 2, when$\lambda_1 = \lambda_2 = 15$ and $\lambda_3 = \cdots = \lambda_k = 1$ or $\lambda_1 = 15$, $\lambda_2 = 10$ and $\lambda_3 = \cdots \lambda_k = 1$.
- 4, when $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 15$ and $\lambda_5 = \cdots = \lambda_k = 1$.
- 0, when $\lambda_1 = \cdots = \lambda_k = 1$, i.e, *noisy scenarios*.

When the scenario is not a *noisy one*, the correlation obtained is greater than 0.9999, which indicates that the algorithms are able to capture the main dimensions of the data.

To provide a visual result, Figure 3.1 shows a boxplot of the correlation coefficients between the four main dimensions of the original dataset $\mathbf{X}$ and the four main dimensions of the MDS obtained with *Divide and Conquer MDS*, $\mathbf{MDS_{Div}}$, for *scenario_id = 60*.[1] As we can see, all the values are close to 1. What shows Figure 3.1 happens for all the *not-noisy scenarios* and all the algorithms.

Note that before calculating the correlation matrix, Procrustes transformation is performed (dilations are not allowed) so that both coordinate systems are the same.

For the *noisy scenarios*, Figure 3.2 shows the boxplot for *scenario_id* $= 51$[2] and *scenario_id* $= 52$[3]. These Figures have been generated using *Divide and Conquer MDS*. Since these scenarios are pure noise, the correlation is low.

We do not observe any negative value since Procrustes alignment is done before calculating the correlation coefficients. Therefore, $\mathbf{X}$ and $\mathbf{MDS_{Div}}$ have the same orientation.

Again, what shows Figure 3.2 also happens for the remaining *noisy scenarios* and for the other two algorithms.

[1] *scenario_id* $= 60$ has the following configuration: $n = 10^6$, 100 columns and 4 main dimensions with $\lambda_i = 15$, $i \in \{1, 2, 3, 4\}$.

[2] *scenario_id* $= 51$ has the following configuration: $n = 10^6$, 10 columns and without any main dimensions.

[3] *scenario_id* $= 52$ has the following configuration: $n = 10^6$, 100 columns and without any main dimensions.

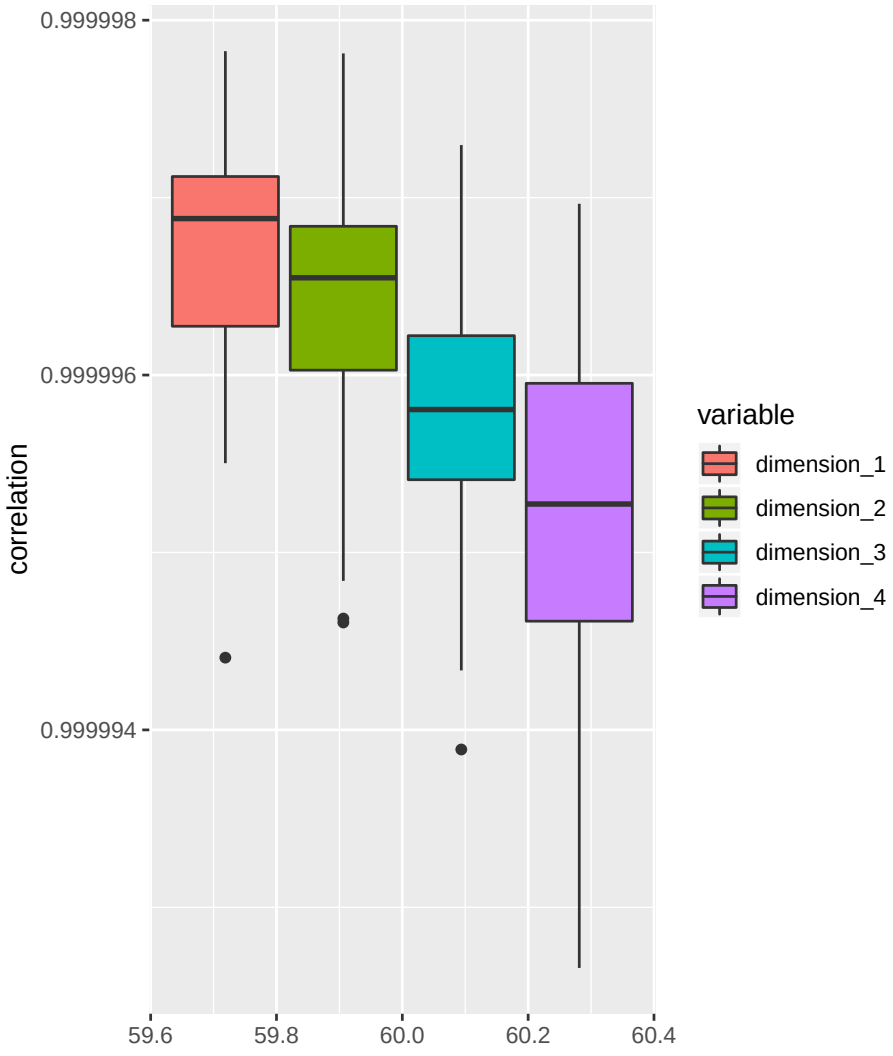


Figure 3.1: Boxplot for correlation coefficients between the main dimensions of $\mathbf{X}$ and the main dimensions of $\mathbf{MDS_{Div}}$ for $scenario\_id = 60$[1].
The correlation is close to 1, indicating that the algorithm captures the main dimensions of the original dataset $\mathbf{X}$.

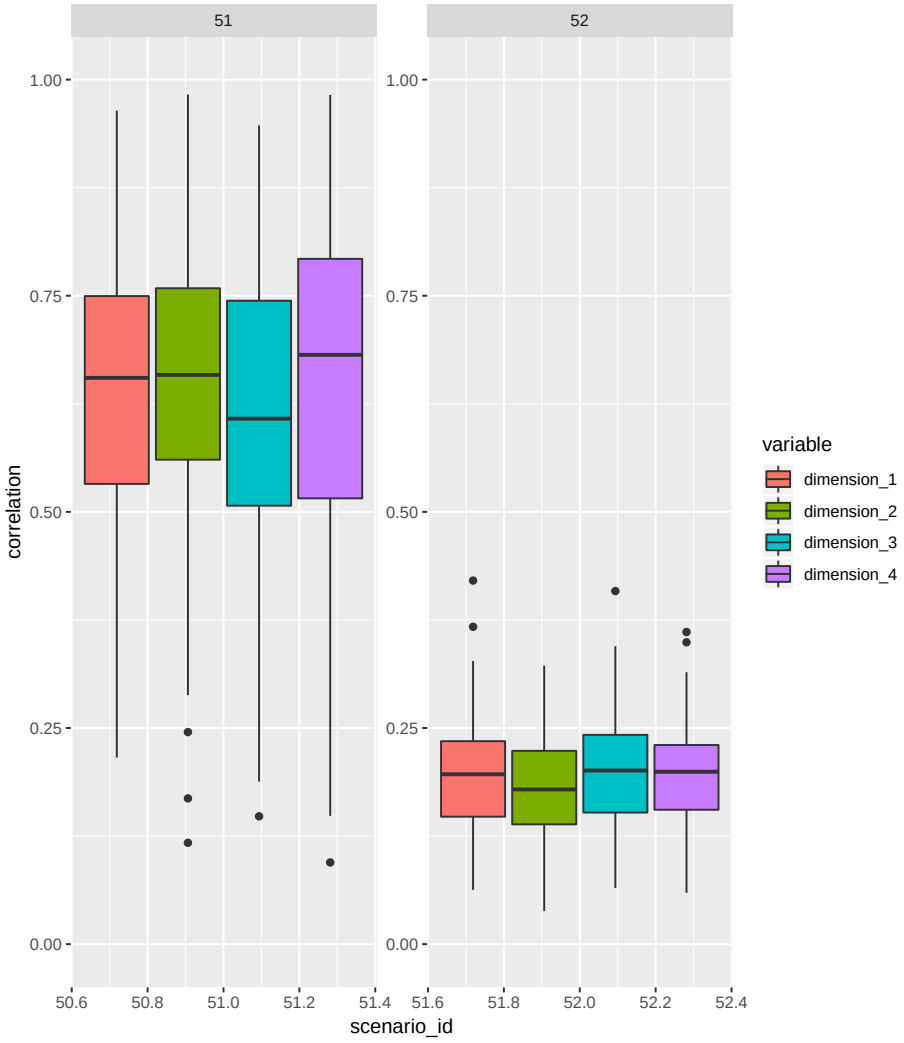


Figure 3.2: Boxplot for correlation coefficients between $\mathbf{X}$ and $\mathbf{MDS_{Div}}$ for two different scenarios: *scenario_id* $= 51^2$ and *scenario_id* $= 52^3$.
Since all of them are noise, the correlation is low.
scenario_id is on the top of each boxplot.

3.3 Eigenvalues

In this section we analyse how the eigenvalues approximate the standard deviation of the original variables.

Since the original dataset, $\mathbf{X}$, is postmultiplied by a diagonal matrix $k \times k$ that contains $\lambda_1, \ldots, \lambda_k$, then $\text{var}(X_i) = \lambda_i^2$ and $\text{sd}(X_i) = \lambda_i$, where X_i is the column i from $\mathbf{X}$.

MDS should be able to capture the variance of the main dimensions through the eigenvalues. Let $\phi_1, \ldots, \phi_t$ be the *normalized eigenvalues* of the MDS configuration such that $\phi_1 > \phi_2 > \cdots > \phi_t$. The first highest *normalized eigenvalues* have to verify $\sqrt{\phi_j} \approx \lambda_j$.

To check how the algorithms approximate the variance of the original data, we compute the bias and the Mean Square Error (MSE) for each scenario. We do not include the noisy ones. Remember that bias and MSE are calculated as follows:

$$\widehat{\text{bias}} = \frac{1}{m}\sum_{i=1}^{m}\sqrt{\phi_{ij}} - \lambda_j = \overline{\sqrt{\phi_j}} - \lambda_j,$$

$$\widehat{\text{MSE}} = \frac{1}{m}\sum_{i=1}^{m}(\lambda_j - \sqrt{\phi_{ij}})^2.$$

Since we perform 100 simulations, $m = 100$. Depending on the scenario, there can be 1, 2 or 4 estimators.

Table 3.3 shows the $\widehat{\text{bias}}$ and the $\widehat{\text{MSE}}$ for *Divide and Conquer MDS* and scenarios with one main dimension $\lambda = 15$. As we can see, the $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ are "low" for these scenarios and this algorithm.

The remaining cases are in Appendix A.1. As long as number of dimensions increases, the $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ do the same. However, they seem to be in an acceptable range.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}}_1$	$\widehat{\text{MSE}}_1$
3	14.98	-0.02	0.03
4	15.03	0.03	0.11
13	15.00	-0.00	0.00
14	14.96	-0.04	0.16
23	14.99	-0.01	0.02
24	14.99	-0.01	0.01
33	14.99	-0.01	0.01
34	14.99	-0.01	0.00
43	14.99	-0.01	0.01
44	14.99	-0.01	0.01
53	14.98	-0.02	0.03
54	14.99	-0.01	0.01

Table 3.3: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with one main dimension $\lambda_1 = 15$ for *Divide and Conquer MDS*.

Table 3.4 shows the $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for *Fast MDS* and scenarios with one main dimension $\lambda = 15$. For this algorithm, the $\widehat{\text{MSE}}$ is higher than for *Divide and Conquer*,

even in some cases (for instance *scenario_id* = 33 [4]) $\widehat{\text{MSE}}$ is high compared with the previous algorithm.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$
3	14.85	-0.15	2.27
4	15.01	0.01	0.01
13	14.91	-0.09	0.76
14	15.10	0.10	0.93
23	14.96	-0.04	0.14
24	15.03	0.03	0.07
33	14.33	-0.67	44.82
34	15.09	0.09	0.76
43	15.00	-0.00	0.00
44	15.00	0.00	0.00
53	14.86	-0.14	1.88
54	14.90	-0.10	1.02

Table 3.4: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with one main dimension $\lambda_1 = 15$ for *Fast MDS*.

One possible reason is the fact that the number of points used to do the Procrustes alignment are not enough. For this case, *scenario_id* = 33, there are 30 points per partition, being 17 partitions in total. So, the alignment is done with 510 points out of 10^4 (i.e, 5% of the points). We have repeated the simulations for *scenario_id* = 33 with 60 points per partition and the $\widehat{\text{MSE}}$ is lower (0.10).

The remaining cases for *Fast MDS* are in Appendix A.2. Again, some of the scenarios have a high $\widehat{\text{MSE}}$ compared with *Divide and Conquer*.

Although *Fast MDS* have higher $\widehat{\text{MSE}}$ values than the previous algorithm, we consider that they are acceptable.

Table 3.5 shows the $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for *MDS based on Gower interpolation* and scenarios with one main dimension $\lambda = 15$. The remaining ones are in Appendix A.2. The comments given to *Divide and Conquer MDS* apply also to these cases.

The algorithm that has the lowest error is the *Divide and Conquer MDS*. Even though the other ones have higher errors, especially *Fast MDS*, we consider that they are good enough.

Note that, we do not consider the *noisy scenarios*, since all the directions have the same variance.

[4] *scenario_id* = 33 has the following configuration: $n = 10^6$, 10 columns and 1 main dimension with $\lambda_1 = 15$.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$
3	15.05	0.05	0.22
4	15.02	0.02	0.04
13	14.94	-0.06	0.36
14	15.04	0.04	0.20
23	14.98	-0.02	0.04
24	15.02	0.02	0.05
33	14.99	-0.01	0.01
34	15.06	0.06	0.31
43	15.04	0.04	0.19
44	14.97	-0.03	0.07
53	14.98	-0.02	0.06
54	14.90	-0.10	1.07

Table 3.5: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with one main dimension $\lambda_1 = 15$ for *MDS based on Gower interpolation*.

3.4 Time to compute MDS

In this section we investigate if there exists an algorithm that is faster than the other ones. Given the results of Table 3.6, it seems that *MDS based on Gower interpolation* has the lowest time. Table 3.6 provides a rank between the methods: it seems that *MDS based on Gower interpolation* is the fastest one. In second position it would be *Fast MDS* and finally *Divide and Conquer*.

sample_size	n_dimensions	mean_divide_conquer	mean_fast	mean_gower
10^3	10	0.27	0.14	0.10
10^3	100	0.78	0.69	0.28
$3 \cdot 10^3$	10	0.78	0.32	0.16
$3 \cdot 10^3$	100	2.50	3.14	0.52
$5 \cdot 10^3$	10	1.37	0.54	0.20
$5 \cdot 10^3$	100	4.25	5.69	0.84
10^4	10	2.60	1.81	0.31
10^4	100	8.85	11.79	1.37
10^5	10	28.10	11.46	2.44
10^5	100	106.30	116.46	18.02
10^6	10	420.29	106.59	53.15
10^6	100	2365.46	1070.19	813.15

Table 3.6: Mean of elapsed time (in seconds) to compute each algorithm.

We do an ANOVA test using three factors: the sample size (which has 6 levels), the number of dimensions (which has 2 levels) and the algorithm (which has 3 levels). Instead of using the *elapsed time* variable, we use its logarithm.

Given the results of the ANOVA test, which are in Table 3.7, we can reject the null hypothesis that the three algorithms have the same expected elapsed time.

We fit a linear regression with these variables and see the magnitude of the coefficients. In addition, we plot the distribution of log(elapsed_time) for all the algorithms.

Table 3.8 contains the value of the coefficients. As long as either the sample size

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
algorithm	2	9283.73	4641.86	32143.99	$< 2e-16$
sample_size	5	108572.93	21714.59	150369.26	$< 2e-16$
n_dimensions	1	12868.36	12868.36	89110.86	$< 2e-16$
Residuals	17991	2598.05	0.14		

Table 3.7: Results for ANOVA test for differences in log(elapsed_time) using algorithms, sample size and num. dimensions as factors.

or the data dimensions increase the coefficients do the same (and so the time needed). Looking at the values for the *algorithm* variable, it seems that *MDS base on Gower interpolation* is the fastest. On the other hand, *Divide and Conquer* is the slowest one.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-1.4058	0.0085	-165.44	$< 2e-16$
algorithmfast	-0.4313	0.0069	-62.17	$< 2e-16$
algorithmgower	-1.6926	0.0069	-243.96	$< 2e-16$
sample_size3000	0.9473	0.0098	96.54	$< 2e-16$
sample_size5000	1.4434	0.0098	147.10	$< 2e-16$
sample_size10000	2.1505	0.0098	219.17	$< 2e-16$
sample_size1e+05	4.4286	0.0098	451.35	$< 2e-16$
sample_size1e+06	7.2782	0.0098	741.78	$< 2e-16$
n_dimensions100	1.6910	0.0057	298.51	$< 2e-16$

Table 3.8: Linear model for response log(elapsed_time).

Figure 3.3 and Figure 3.4 show the estimated density of the elapsed time for each algorithm and each *scenario_id* for $n = 10^3$. As we can see, *MDS based on Gower interpolation* is the fastest algorithm, especially when 100 dimensions are required. The remaining figures, i.e $n > 10^3$, are in Appendix B.

One interesting thing that we can observe from Appendix B is the fact that the elapsed time grows as long as the sample size does, especially form *Divide and Conquer MDS*. However, *MDS based on Gower interpolation* and *Fast MDS* provide a really good time even though the sample size is large, being *MDS based on Gower interpolation* the fastest one. So, we can consider both algorithms efficient, since they are able to compute MDS is a reasonable amount of time.

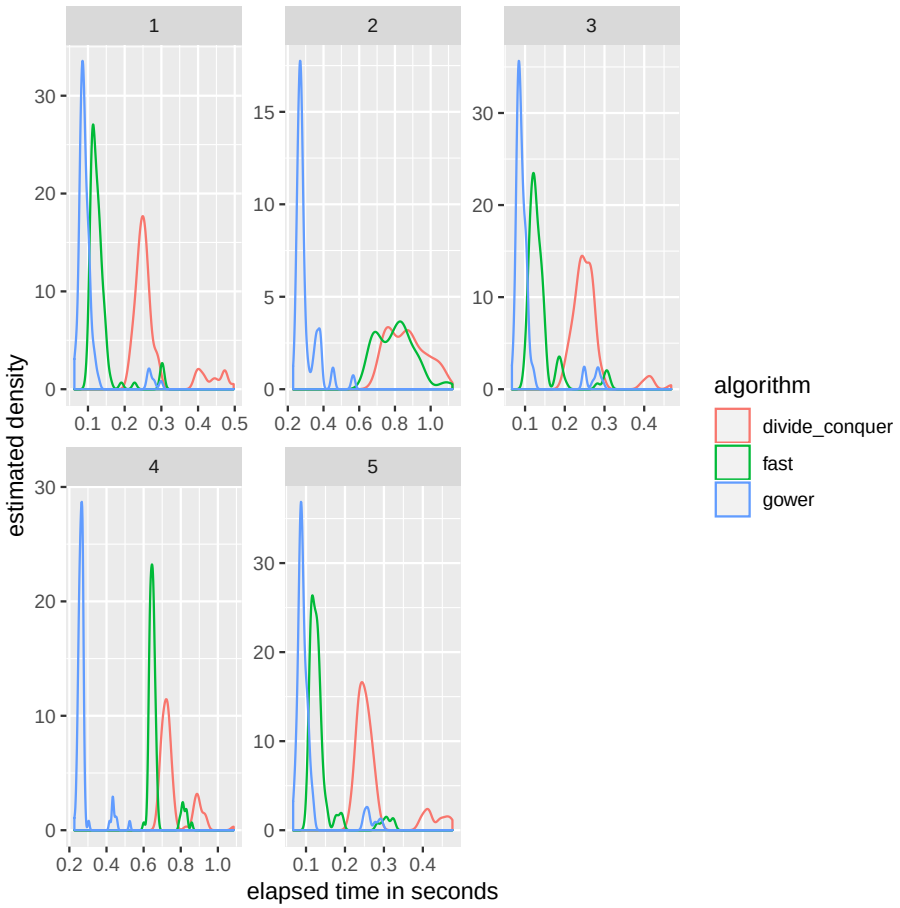


Figure 3.3: Estimated density of elapsed time (in sec.) for each algorithm and the first five *scenario_id* of $n = 10^3$.
scenario_id is on the top of each plot.

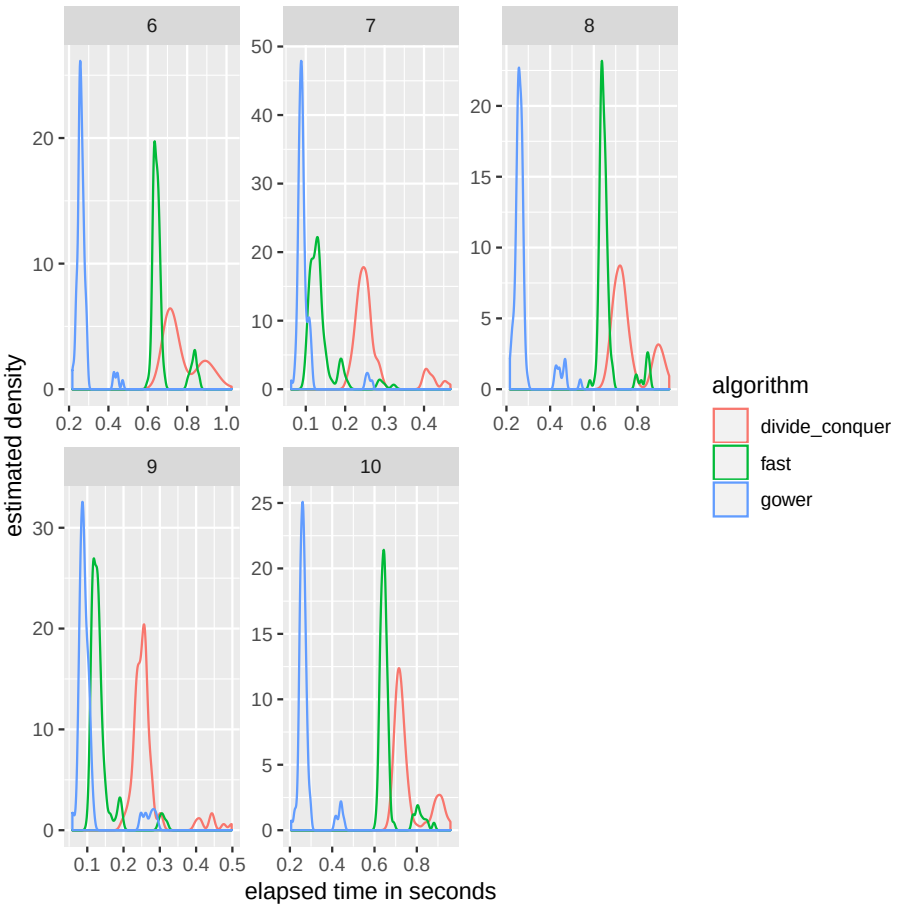


Figure 3.4: Estimated density of elapsed time (in sec.) for each algorithm and the last five *scenario_id* of $n = 10^3$.
scenario_id is on the top of each plot.

Chapter 4

Conclusions

The goal of this master's thesis was to find an algorithm able to compute a MDS configuration when dealing with large datasets in an "efficient" way, i.e, such an algorithm should be fast enough to get the configuration.

Even though our first method, *Divide and Conquer MDS*, is the slowest one, it is able to obtain a low-dimensional configuration. In addition, it has a really good property that the other two algorithms do not have: it is able to capture the variance of the original data quite well.

Fast MDS provides an improvement of timing, being faster than *Divide and Conquer MDS*. However, it is based on recursive programming. The problem with these kind of algorithms is that they can consume a lot of memory.

In addition, a carefully selection of the number of points to perform Procrustes alignment has to be done, since the $\widehat{\text{MSE}}$ can be high, as it happen with our simulation study.

A really good algorithm is *MDS based on Gower interpolation*, since it provides a MDS configuration in a short amount of time with low errors and its implementation is easy. Apart from this, it does not need to do any Procrustes transformation, which save time and memory. Therefore, this is the algorithm to obtain MDS with large datasets.

Observe that this algorithm does essentially what classical Statistics advises: when your population is too large, take a sample of it.

Problem encountered during the development of the thesis

Basically we have faced two kind of problems, which are:

- Computational problems: when working with large datasets, we suffered from consuming all RAM of the servers. Especially when doing Procrustes for aligning the original data and the MDS for n large. The solution to it was to partition the process into pieces that the servers could manage without consuming all RAM.
- Procrustes packages: even though R has a lot of packages that allows to compute Procrustes, they did not fulfilled our goals. The reasons are either because the output is not well specified or because some of the transformations (mainly dilations or translations) were not included. We recommend to use MCMCpack package.

Future research

The algorithms that we have developed are implemented in R, which is a good language to do prototypes. However, this is not the best programming language to be used.

So, we recommend to implement them in a robust programming language such as C, C++ or Java.

Tuning of parameters: as we have seen in *Fast MDS*, the $\widehat{\text{MSE}}$ depends on the number of points to perform Procrustes alignment. So, a (simulation) study on this parameter should be done in order to obtain the "optimal" value for which the $\widehat{\text{MSE}}$ is as good as the one for *Divide and Conquer*.

On the other hand, it might be that *Divide and Conquer* uses more than necessary points to perform Procrustes alignment. Reducing the number of points would preserve the $\widehat{\text{MSE}}$ while improving the time needed to get the MDS configuration.

Given that the world is talking about Big Data, it is a good opportunity to challenge the algorithms and use them with Spark/Hadoop. Actually, the initial idea of this thesis was to use them with a Spark DB that contains millions of chess games. Due to lack of time, we could not do so.

In the same line of Spark/Hadoop, an implementation based on *map-reduce* would speed up the algorithms, reducing the time needed to get the low-dimensional configuration.

Acknowledgments

I would like to express my gratitude to *Pedro Delicado*, since he helped me to do this thesis. There have been some difficult issues in which his help allowed me to move on and solve them. Also, I would like to thank *Roger Devesa*, since he has helped me with reviewing recursive programming and with implementing *Fast MDS*.

Bibliography

Borg, I. and P. Groenen (2005). *Modern Multidimensional Scaling: Theory and Applications*. Springer.

Gower, J. C. and D. J. Hand (1995). *Biplots*, Volume 54. CRC Press.

Peña, D. (2002). *Análisis de datos multivariantes*. Madrid, Spain: McGraw Hill.

Tynia, Y., L. Jinze, M. Leonard, and W. Wei (2006). A fast approximation to multidimensional scaling.

Appendix A

Bias and MSE for eigenvalues

A.1 Divide and Conquer MDS

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
5	15.41	0.41	17.12	14.59	0.41	16.99
6	15.41	0.41	17.08	14.54	0.41	20.88
15	15.40	0.40	15.99	14.55	0.40	19.89
16	15.39	0.39	15.03	14.54	0.39	21.59
25	15.41	0.41	16.71	14.55	0.41	20.01
26	15.41	0.41	16.94	14.57	0.41	18.79
35	15.39	0.39	15.57	14.54	0.39	21.13
36	15.42	0.42	17.60	14.57	0.42	18.22
45	15.40	0.40	15.77	14.56	0.40	19.42
46	15.41	0.41	16.85	14.56	0.41	19.15
55	15.40	0.40	16.00	14.56	0.40	19.52
56	15.41	0.41	16.73	14.56	0.41	18.96

Table A.1: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *Divide and Conquer MDS*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
7	14.96	-0.04	0.15	9.95	-0.05	0.27
8	14.98	-0.02	0.02	9.92	-0.08	0.57
17	15.01	0.01	0.02	9.98	-0.02	0.05
18	15.03	0.03	0.08	9.98	-0.02	0.04
27	15.00	0.00	0.00	9.99	-0.01	0.02
28	15.01	0.01	0.01	9.97	-0.03	0.11
37	15.01	0.01	0.01	9.97	-0.03	0.06
38	15.01	0.01	0.00	10.00	-0.00	0.00
47	15.00	0.00	0.00	9.98	-0.02	0.05
48	15.01	0.01	0.00	9.98	-0.02	0.04
57	15.00	-0.00	0.00	9.97	-0.03	0.08
58	15.00	0.00	0.00	9.98	-0.02	0.03

Table A.2: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *Divide and Conquer MDS*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$	$\overline{\sqrt{\phi_3}}$	$\widehat{\text{bias}_3}$	$\widehat{\text{MSE}_3}$	$\overline{\sqrt{\phi_4}}$	$\widehat{\text{bias}_4}$	$\widehat{\text{MSE}_4}$
9	15.88	0.88	77.09	15.26	0.26	6.94	14.71	-0.29	8.40	14.08	-0.92	84.94
10	15.89	0.89	80.08	15.26	0.26	6.52	14.70	-0.30	8.82	14.09	-0.91	83.59
19	15.90	0.90	80.98	15.26	0.26	6.87	14.68	-0.32	10.37	14.04	-0.96	91.72
20	15.88	0.88	76.61	15.25	0.25	6.02	14.70	-0.30	9.07	14.04	-0.96	91.55
29	15.87	0.87	76.05	15.26	0.26	6.58	14.68	-0.32	9.99	14.06	-0.94	87.55
30	15.87	0.87	75.86	15.24	0.24	5.83	14.68	-0.32	10.21	14.07	-0.93	86.76
39	15.89	0.89	79.01	15.25	0.25	6.07	14.69	-0.31	9.80	14.06	-0.94	87.95
40	15.91	0.91	82.54	15.25	0.25	6.32	14.69	-0.31	9.64	14.06	-0.94	87.92
49	15.89	0.89	78.58	15.25	0.25	6.10	14.68	-0.32	9.98	14.06	-0.94	87.77
50	15.89	0.89	79.78	15.25	0.25	6.30	14.69	-0.31	9.69	14.07	-0.93	86.87
59	15.89	0.89	78.65	15.25	0.25	6.09	14.68	-0.32	9.95	14.06	-0.94	88.05
60	15.89	0.89	79.73	15.25	0.25	6.40	14.69	-0.31	9.57	14.07	-0.93	86.68

Table A.3: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Divide and Conquer MDS*.

A.2 Fast MDS

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
5	16.06	1.06	112.12	13.79	-1.21	145.86
6	15.41	0.41	17.21	14.56	-0.44	19.44
15	15.71	0.71	49.84	14.31	-0.69	46.93
16	15.40	0.40	15.93	14.39	-0.61	37.06
25	15.50	0.50	24.80	14.43	-0.57	32.32
26	15.45	0.45	20.39	14.46	-0.54	29.07
35	16.52	1.52	231.64	13.35	-1.65	271.33
36	15.46	0.46	21.30	14.49	-0.51	25.90
45	15.45	0.45	20.62	14.37	-0.63	40.08
46	15.41	0.41	17.03	14.55	-0.45	20.34
55	15.67	0.67	45.00	14.27	-0.73	52.78
56	15.35	0.35	12.11	14.57	-0.43	18.11

Table A.4: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *Fast MDS*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
7	15.00	0.00	0.00	9.82	-0.18	3.22
8	14.96	-0.04	0.17	9.93	-0.07	0.56
17	15.14	0.14	2.00	9.88	-0.12	1.37
18	14.99	-0.01	0.00	9.98	-0.02	0.04
27	14.91	-0.09	0.75	9.95	-0.05	0.21
28	14.89	-0.11	1.29	9.90	-0.10	0.92
37	15.10	0.10	0.93	9.64	-0.36	13.12
38	14.97	-0.03	0.10	9.96	-0.04	0.12
47	15.02	0.02	0.05	9.93	-0.07	0.47
48	14.99	-0.01	0.02	10.02	0.02	0.03
57	14.88	-0.12	1.38	9.91	-0.09	0.74
58	15.03	0.03	0.09	9.99	-0.01	0.02

Table A.5: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *Fast MDS*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$	$\overline{\sqrt{\phi_3}}$	$\widehat{\text{bias}_3}$	$\widehat{\text{MSE}_3}$	$\overline{\sqrt{\phi_4}}$	$\widehat{\text{bias}_4}$	$\widehat{\text{MSE}_4}$
9	17.43	2.43	590.46	15.58	0.58	33.32	13.74	-1.26	158.12	11.99	-3.01	903.29
10	15.90	0.90	80.54	15.25	0.25	6.49	14.70	-0.30	8.79	14.08	-0.92	84.90
19	16.40	1.40	195.78	15.27	0.27	7.32	14.31	-0.69	47.00	13.33	-1.67	280.08
20	16.01	1.01	102.73	15.32	0.32	10.08	14.65	-0.35	11.97	13.87	-1.13	127.87
29	16.14	1.14	130.55	15.29	0.29	8.64	14.53	-0.47	22.40	13.81	-1.19	141.75
30	16.02	1.02	104.48	15.32	0.32	10.27	14.63	-0.37	13.72	14.00	-1.00	99.58
39	17.96	2.96	875.12	15.53	0.53	27.71	13.70	-1.30	169.28	11.42	-3.58	1281.54
40	16.01	1.01	102.49	15.29	0.29	8.26	14.69	-0.31	9.35	13.93	-1.07	114.86
49	16.14	1.14	129.03	15.39	0.39	15.54	14.63	-0.37	13.90	13.94	-1.06	112.16
50	15.98	0.98	96.04	15.25	0.25	6.28	14.61	-0.39	15.18	13.97	-1.03	106.71
59	16.29	1.29	165.16	15.22	0.22	4.73	14.37	-0.63	39.55	13.44	-1.56	243.94
60	15.98	0.98	95.75	15.33	0.33	10.87	14.70	-0.30	9.21	14.06	-0.94	87.68

Table A.6: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Fast MDS*.

A.3 MDS based on Gower interpolation

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
5	15.45	0.45	20.49	14.63	-0.37	13.52
6	15.41	0.41	17.04	14.56	-0.44	19.36
15	15.42	0.42	17.23	14.59	-0.41	16.71
16	15.35	0.35	12.08	14.47	-0.53	28.25
25	15.45	0.45	19.82	14.55	-0.45	19.91
26	15.40	0.40	16.06	14.52	-0.48	23.04
35	15.36	0.36	13.23	14.53	-0.47	22.56
36	15.40	0.40	15.75	14.53	-0.47	22.29
45	15.36	0.36	12.93	14.47	-0.53	27.88
46	15.38	0.38	14.27	14.60	-0.40	16.20
55	15.37	0.37	13.86	14.52	-0.48	22.97
56	15.33	0.33	11.15	14.57	-0.43	18.74

Table A.7: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 15$ for *MDS based on Gower interpolation*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$
7	14.95	-0.05	0.25	9.95	-0.05	0.24
8	14.96	-0.04	0.18	9.93	-0.07	0.51
17	15.05	0.05	0.27	9.98	-0.02	0.06
18	15.02	0.02	0.04	9.99	-0.01	0.01
27	14.98	-0.02	0.05	9.98	-0.02	0.03
28	14.92	-0.08	0.63	9.90	-0.10	1.05
37	14.98	-0.02	0.02	9.98	-0.02	0.04
38	14.97	-0.03	0.07	9.97	-0.03	0.11
47	15.05	0.05	0.28	9.97	-0.03	0.07
48	15.00	-0.00	0.00	10.00	-0.00	0.00
57	14.89	-0.11	1.19	9.98	-0.02	0.05
58	15.03	0.03	0.09	9.98	-0.02	0.06

Table A.8: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with two main dimensions $\lambda_1 = 15$ and $\lambda_2 = 10$ for *MDS based on Gower interpolation*.

scenario_id	$\overline{\sqrt{\phi_1}}$	$\widehat{\text{bias}_1}$	$\widehat{\text{MSE}_1}$	$\overline{\sqrt{\phi_2}}$	$\widehat{\text{bias}_2}$	$\widehat{\text{MSE}_2}$	$\overline{\sqrt{\phi_3}}$	$\widehat{\text{bias}_3}$	$\widehat{\text{MSE}_3}$	$\overline{\sqrt{\phi_4}}$	$\widehat{\text{bias}_4}$	$\widehat{\text{MSE}_4}$
9	15.89	0.89	79.83	15.28	0.28	7.68	14.73	-0.27	7.38	14.08	-0.92	84.17
10	15.90	0.90	80.72	15.26	0.26	7.00	14.70	-0.30	9.07	14.10	-0.90	81.70
19	15.83	0.83	68.94	15.20	0.20	4.00	14.63	-0.37	13.55	14.03	-0.97	94.05
20	15.86	0.86	73.48	15.24	0.24	5.66	14.67	-0.33	10.78	14.02	-0.98	96.87
29	15.88	0.88	76.87	15.24	0.24	5.91	14.66	-0.34	11.64	14.08	-0.92	84.24
30	15.93	0.93	86.05	15.27	0.27	7.03	14.67	-0.33	10.97	14.12	-0.88	77.32
39	15.87	0.87	75.93	15.23	0.23	5.30	14.67	-0.33	10.59	14.05	-0.95	91.13
40	15.90	0.90	80.39	15.26	0.26	6.61	14.68	-0.32	10.22	14.07	-0.93	86.34
49	15.97	0.97	94.74	15.30	0.30	9.03	14.70	-0.30	8.94	14.13	-0.87	74.89
50	15.88	0.88	77.34	15.22	0.22	5.03	14.66	-0.34	11.33	14.10	-0.90	81.27
59	15.88	0.88	77.43	15.20	0.20	4.12	14.63	-0.37	13.67	14.03	-0.97	94.66
60	15.97	0.97	94.19	15.33	0.33	10.85	14.70	-0.30	9.29	14.07	-0.93	86.59

Table A.9: Estimator, $\widehat{\text{bias}}$ and $\widehat{\text{MSE}}$ for scenarios with four main dimensions $\lambda_i = 15$ $i \in \{1, 2, 3, 4\}$ for *Fast MDS*.

Appendix B

Time required to compute MDS configuration

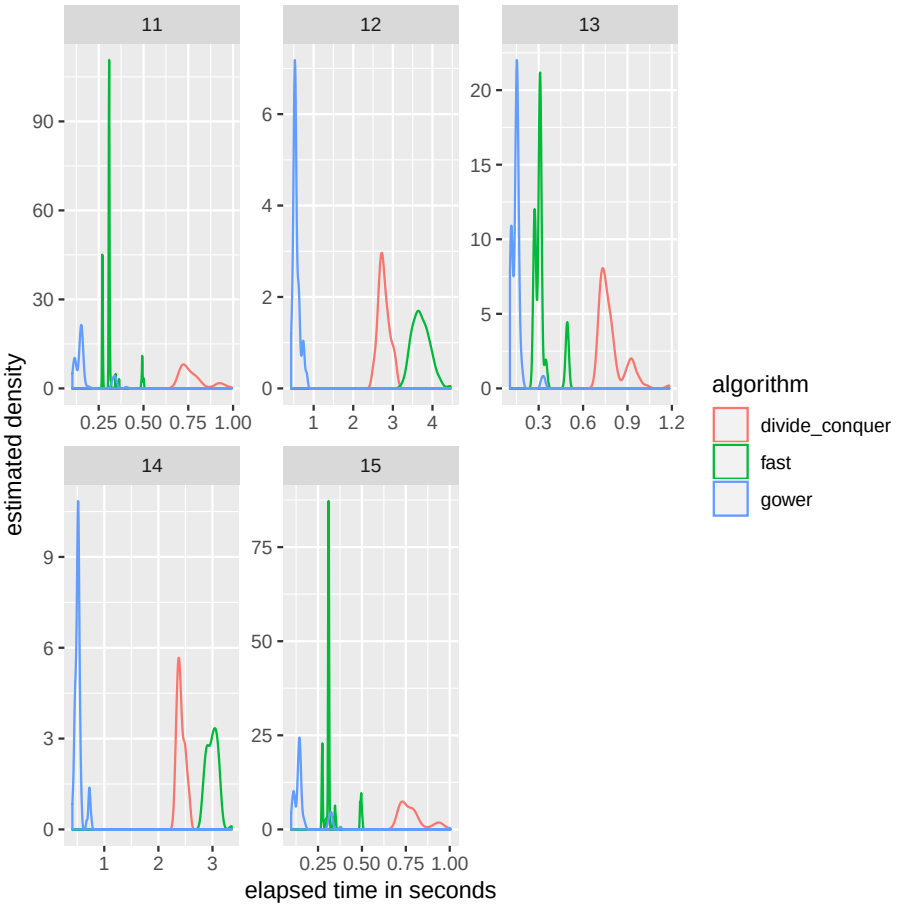


Figure B.1: Estimated density of elapsed time (in sec.) for each algorithm and the five first *scenario_id* of $n = 3 \cdot 10^3$.
scenario_id is on the top of each plot.

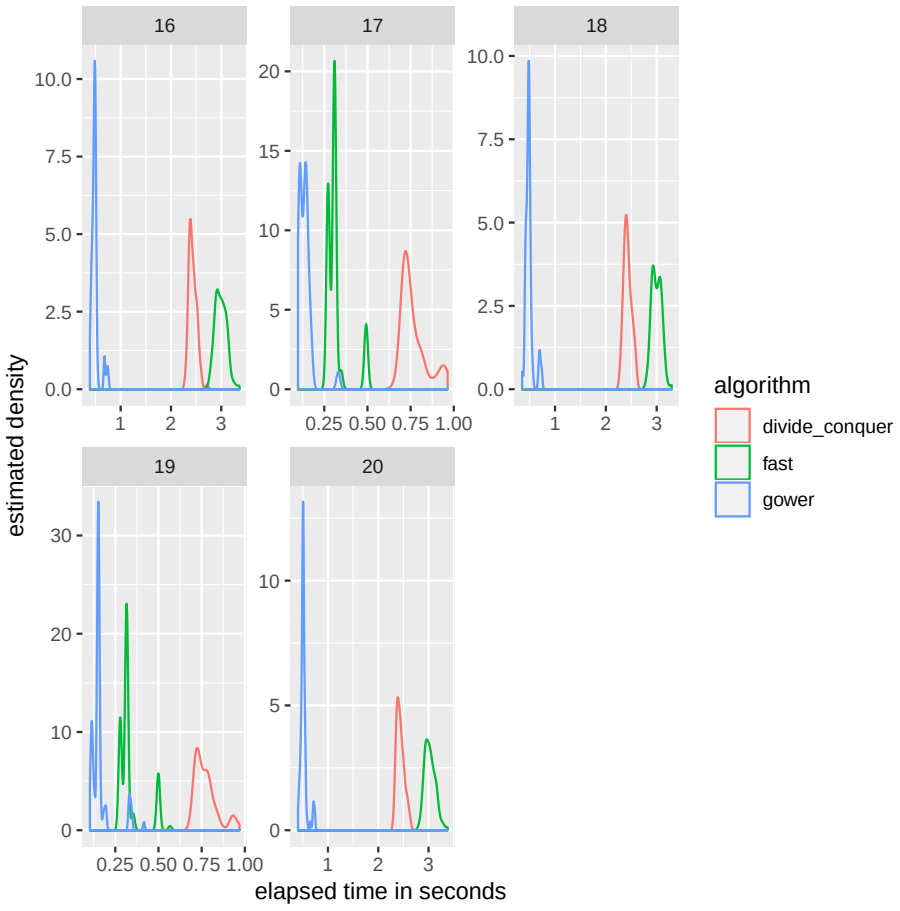


Figure B.2: Estimated density of elapsed time (in sec.) for each algorithm and the five last *scenario_id* of $n = 3 \cdot 10^3$.
scenario_id is on the top of each plot.

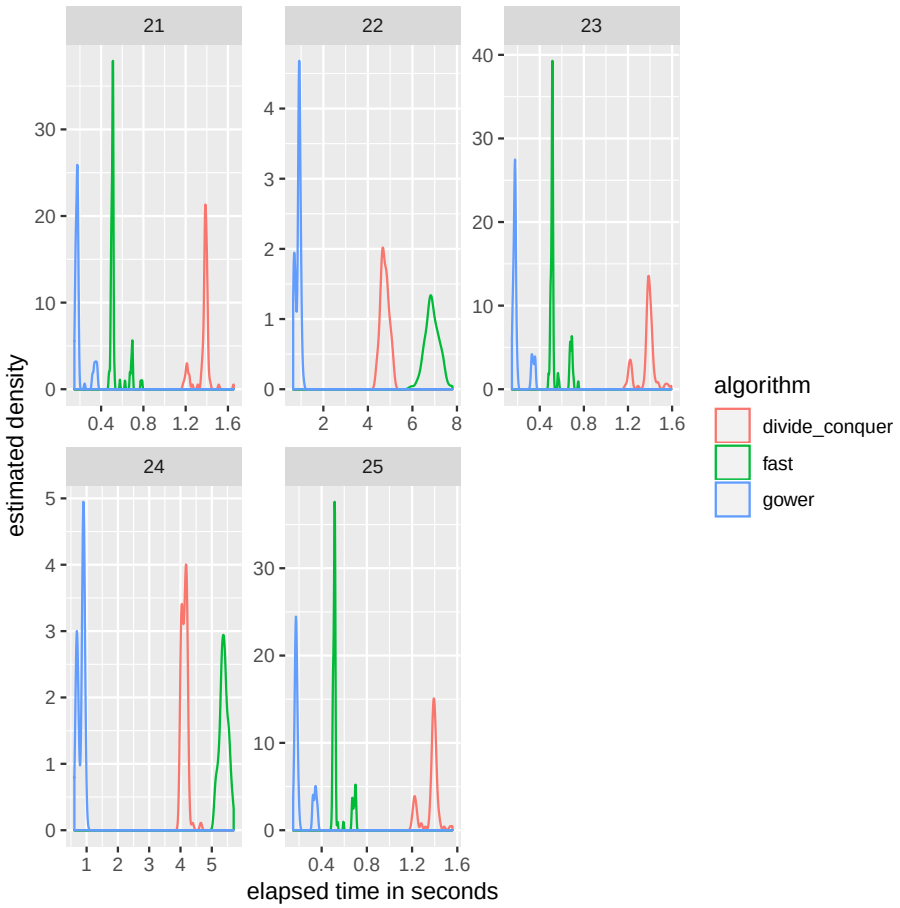


Figure B.3: Estimated density of elapsed time (in sec.) for each algorithm and the five first *scenario_id* of $n = 5 \cdot 10^3$.
scenario_id is on the top of each plot.

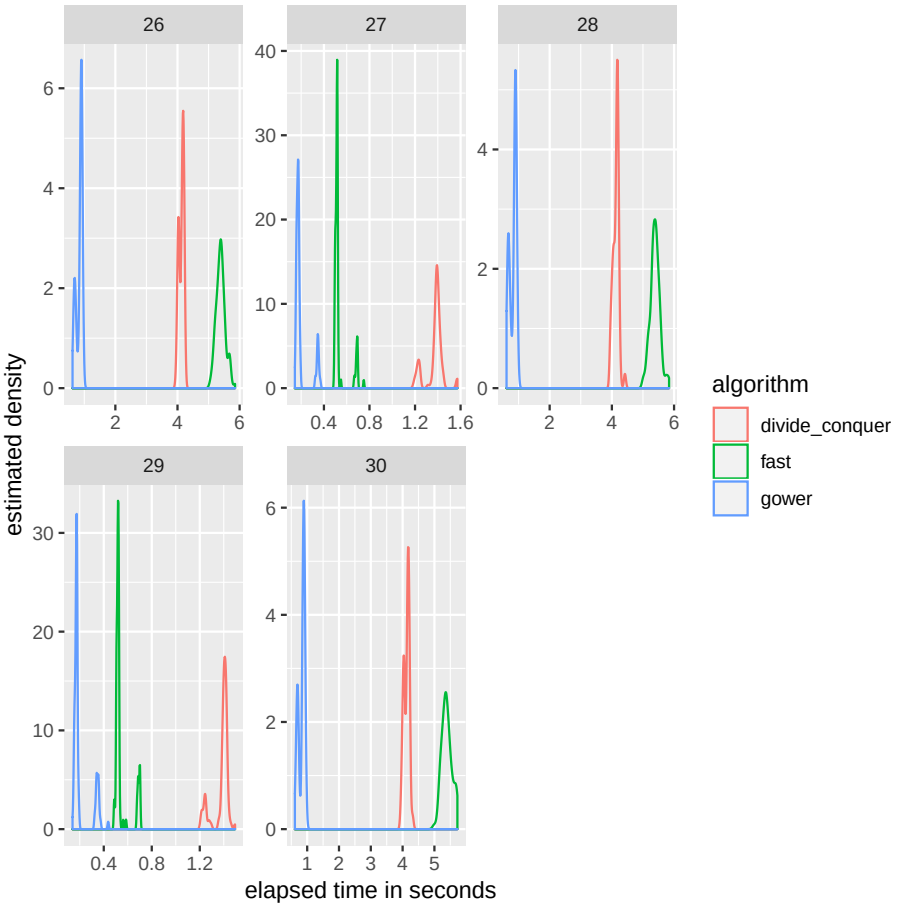


Figure B.4: Estimated density of elapsed time (in sec.) for each algorithm and the five last *scenario_id* of $n = 5 \cdot 10^3$.
scenario_id is on the top of each plot.

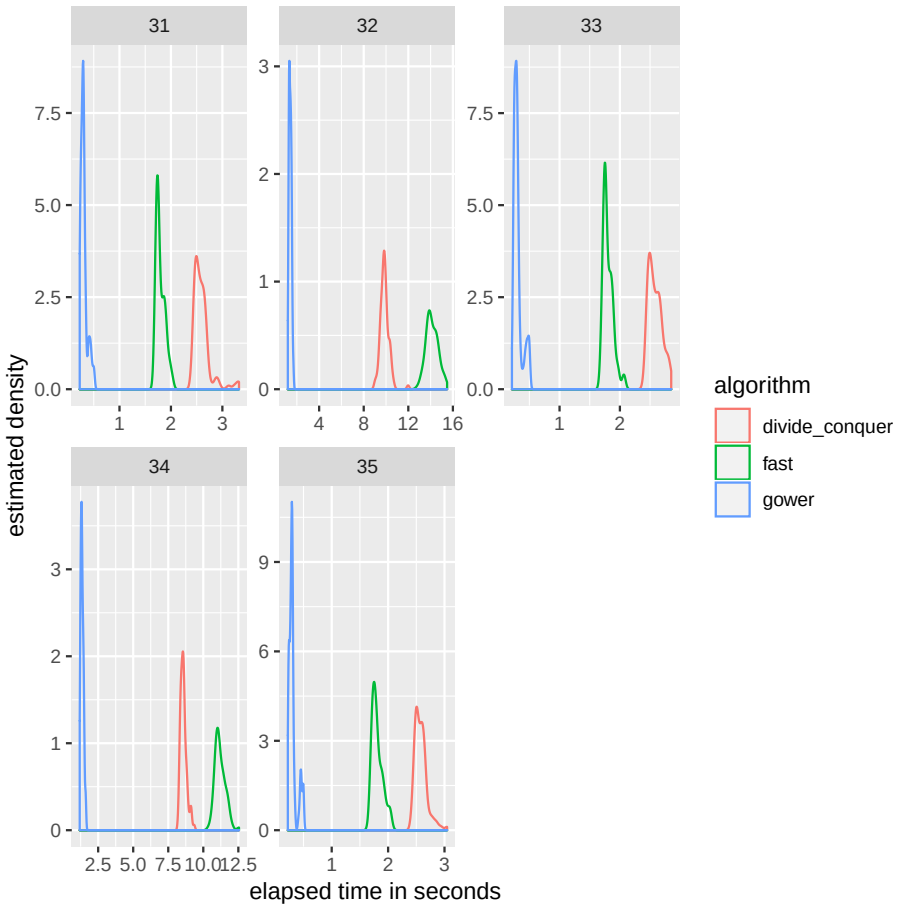


Figure B.5: Estimated density of elapsed time (in sec.) for each algorithm and the five first *scenario_id* of $n = 10^4$.
scenario_id is on the top of each plot.

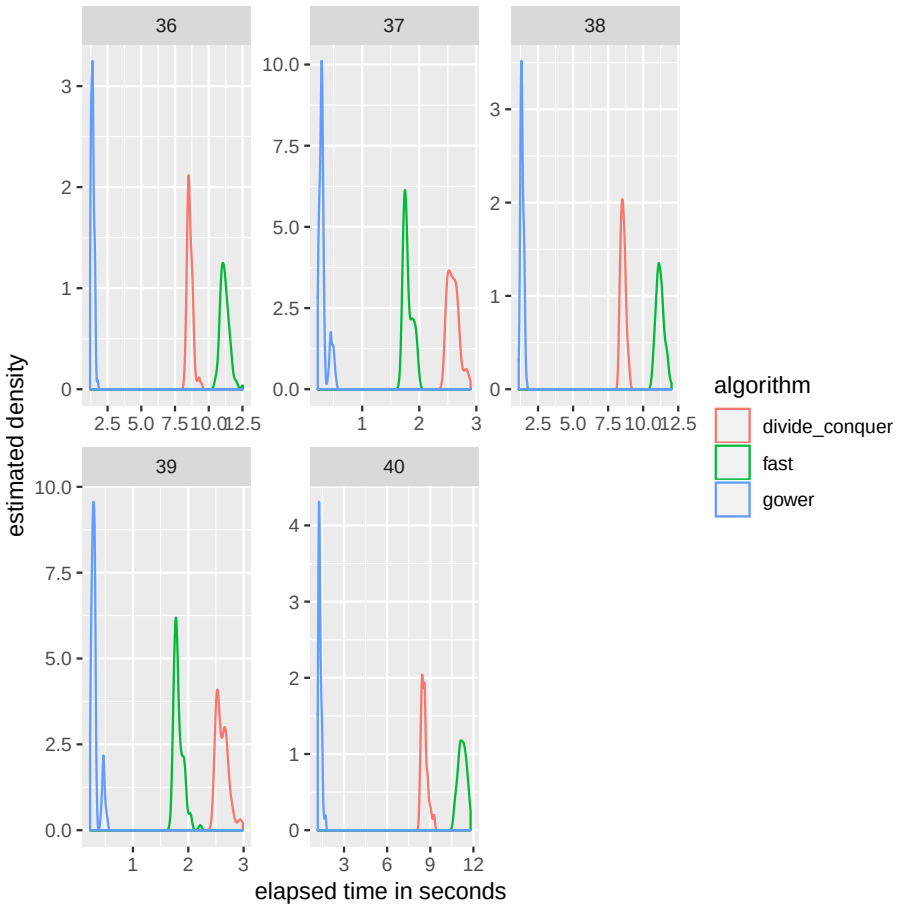


Figure B.6: Estimated density of elapsed time (in sec.) for each algorithm and the five last *scenario_id* of $n = 10^4$.
scenario_id is on the top of each plot.

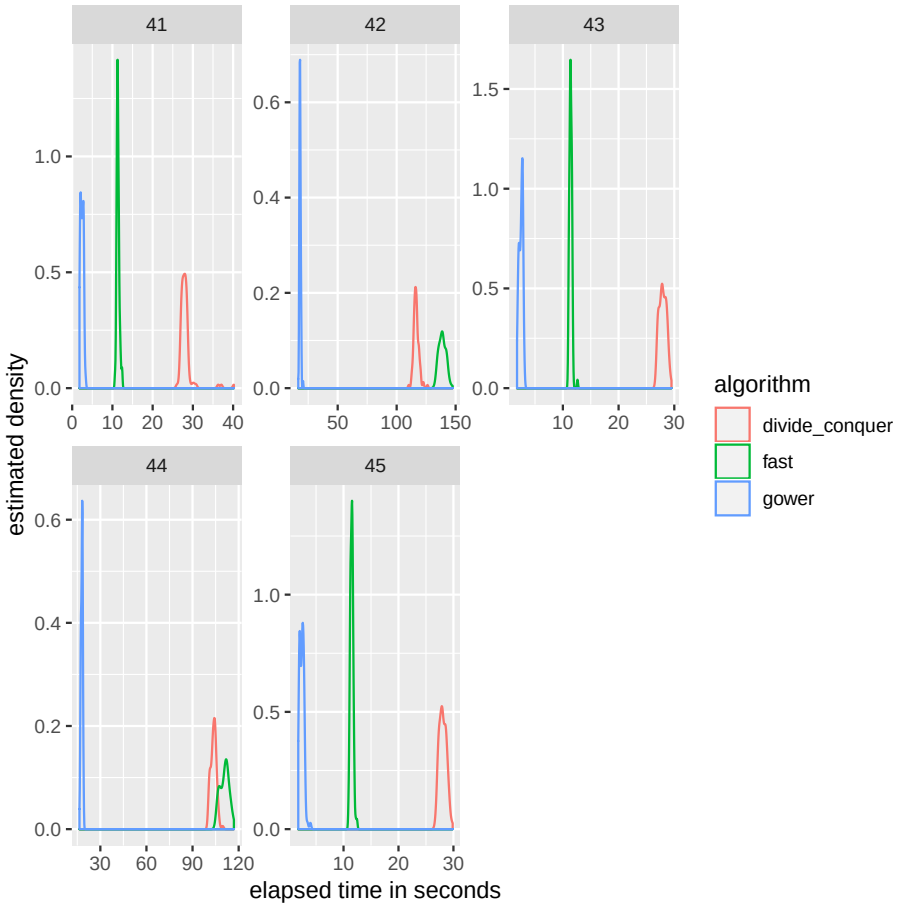


Figure B.7: Estimated density of elapsed time (in sec.) for each algorithm and the five first *scenario_id* of $n = 10^5$.
scenario_id is on the top of each plot.

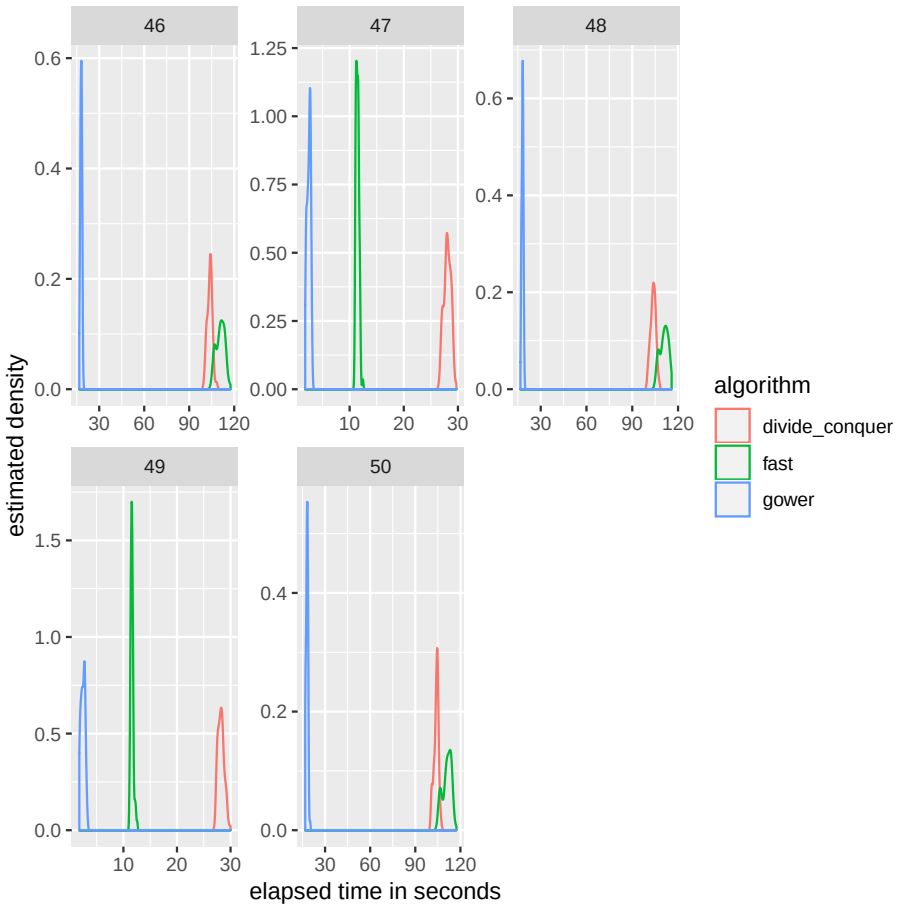


Figure B.8: Estimated density of elapsed time (in sec.) for each algorithm and the five last *scenario_id* of $n = 10^5$.
scenario_id is on the top of each plot.

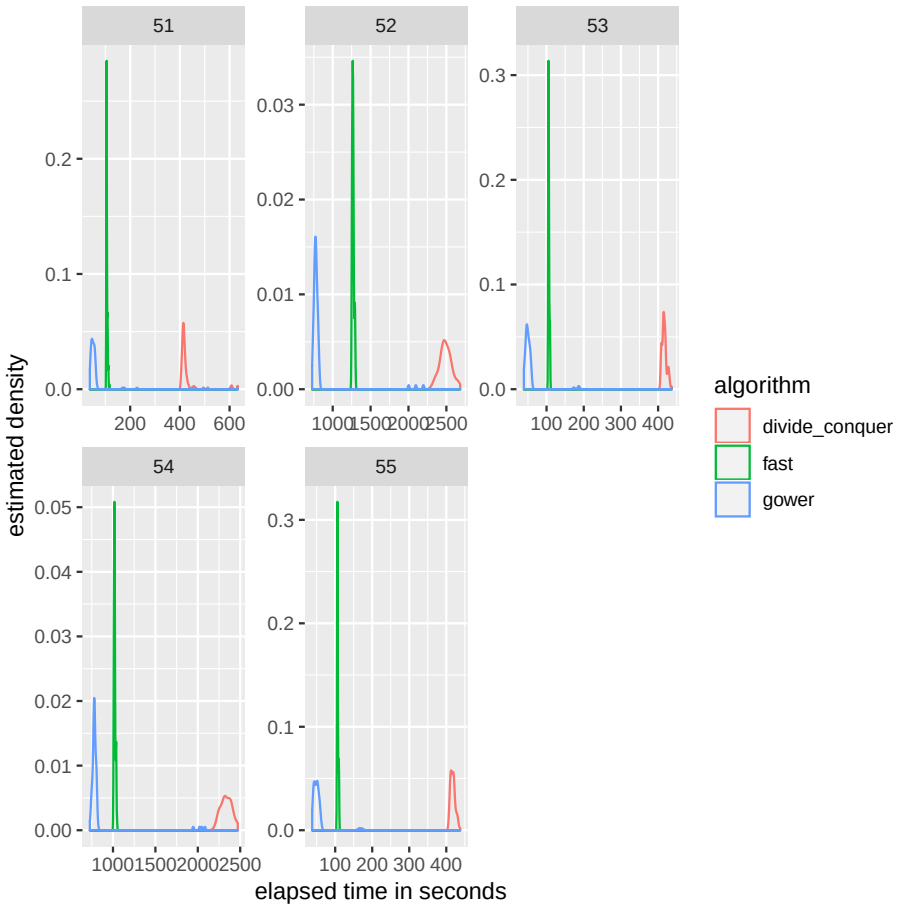


Figure B.9: Estimated density of elapsed time (in sec.) for each algorithm and the five first *scenario_id* of $n = 10^6$.
scenario_id is on the top of each plot.

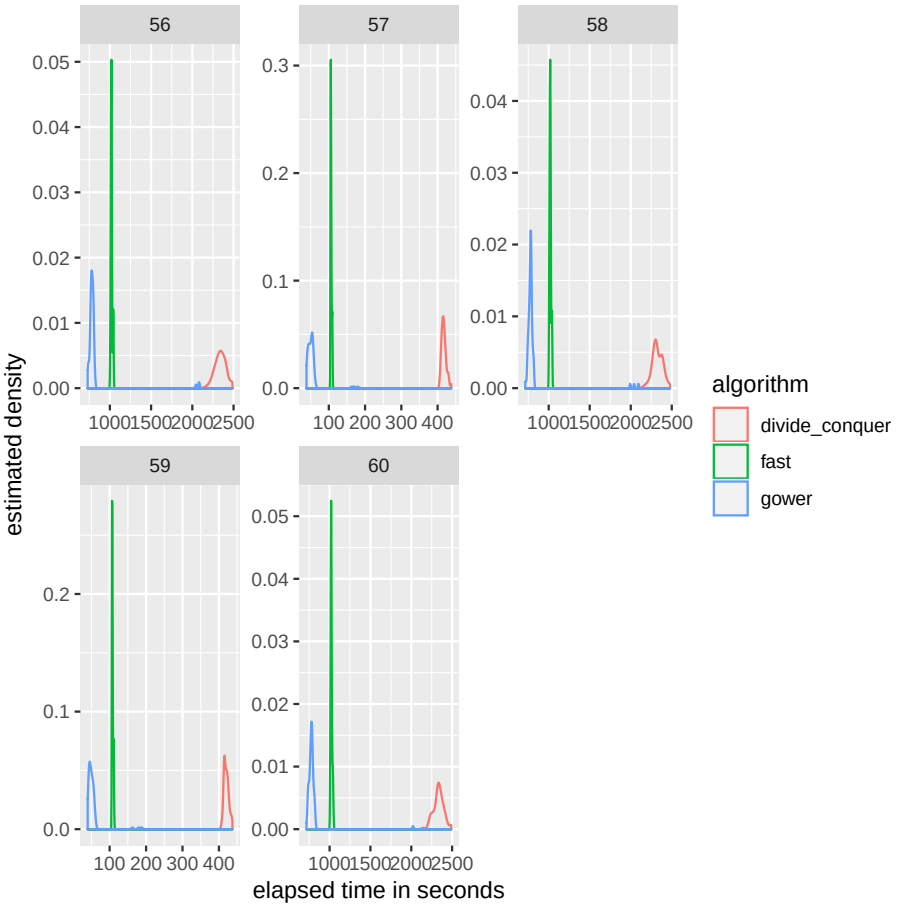


Figure B.10: Estimated density of elapsed time (in sec.) for each algorithm and the five last *scenario_id* of $n = 10^6$.
scenario_id is on the top of each plot.

Appendix C

Code

C.1 Divide and Conquer MDS

```
divide_conquer.mds ← function(
  x,
  l,
  s,
  metric
){

  # List positions
  ls_positions = list()
  list_eigenvalues = list()
  i_eigen = 1

  # Initial parameters
  p = ceiling(2*nrow(x)/l)
  groups = sample(x = p, size = nrow(x), replace = TRUE)
  groups = sort(groups)
  unique_group = unique(groups)
  total_groups = length(unique_group)

  for(k in 1:total_groups){
    # Getting the group that is being processed
    current_group = unique_group[k]
    x_positions_current_group = which(groups == current_group)
    ls_positions[[k]] = x_positions_current_group

    # Take the data in the following way:
    #   If it is the first iteration, take the data from first
       group
    #   else, take the data from k-1 and k groups
    if(k == 1){
      filter_rows_by_position = x_positions_current_group
      rows_processed = x_positions_current_group
    }else{

      rows_processed = c(
        rows_processed,
        x_positions_current_group
      )
```

```
38        previous_group = unique_group[k-1]
39        x_positions_previous_group =  which(groups == previous_group
              )
40
41        # Rows to be filtered
42        filter_rows_by_position = c(
43          x_positions_previous_group,
44          x_positions_current_group
45        )
46
47      }
48
49      # Matrix to apply MDS
50      submatrix_data = x[filter_rows_by_position, ]
51
52      # Calculate distance
53      distance_matrix = cluster::daisy(
54        x = submatrix_data,
55        metric = metric
56      )
57
58      # Applying MDS to the submatrix of data
59      cmd_eig = stats::cmdscale(
60        d = distance_matrix,
61        k = s,
62        eig = TRUE
63      )
64
65      mds_iteration = cmd_eig$points
66      if(p%%2 == 0){
67        if(k%%2 == 0){
68          list_eigenvalues[[i_eigen]] = cmd_eig$eig/nrow(
                submatrix_data)
69          i_eigen = i_eigen + 1
70        }
71      }else{
72        if(k %% 2 == 1){
73          list_eigenvalues[[i_eigen]] = cmd_eig$eig/nrow(
                submatrix_data)
74          i_eigen = i_eigen + 1
75        }
76      }
77
78          row.names(mds_iteration) = row.names(submatrix_data)
79
80      if(k == 1){
81        # Define cum-MDS as MDS(1)
82        cum_mds = mds_iteration
83
84      }else{
85        # Take the result of MDS(k-1) obtained with k-1 and k
86        rn_prev = row.names(x)[x_positions_previous_group]
87        rn_cur = row.names(x)[x_positions_current_group]
88        pos_previous_group_current_mds = row.names(mds_iteration) %
              in% rn_prev
```

```
89        pos_current_group_current_mds = row.names(mds_iteration) %in
              % rn_cur
90        mds_previous = mds_iteration[pos_previous_group_current_mds
              ,]
91        mds_current = mds_iteration[pos_current_group_current_mds,]
92
93        # From cum-MDS take the result of group k-1
94        positions_cum_sum_previous = which(row.names(cum_mds) %in%
              rn_prev
95        cum_mds_previous = cum_mds[positions_cum_sum_previous, ]
96
97
98        # Apply Procrustes transformation
99        procrustes_result =  MCMCpack::procrustes(
100         X = mds_previous, #The matrix to be transformed
101         Xstar = cum_mds_previous, # target matrix
102         translation = TRUE,
103         dilation = TRUE
104       )
105
106       rotation_matrix = procrustes_result$R
107       dilation = procrustes_result$s
108       translation = procrustes_result$tt
109       ones_vector = rep(1, nrow(mds_current))
110       translation_matrix = ones_vector %*% t(translation)
111
112       # Transform the data for the k-th group
113       cum_mds_current = dilation * mds_current %*% rotation_matrix
              +
114         translation_matrix
115
116       cum_mds = rbind(
117         cum_mds,
118         cum_mds_current
119       )
120
121     }
122
123   }
124
125   # Reordering
126   reording_permutation = match(1:nrow(x), rows_processed)
127   cum_mds = cum_mds[reording_permutation, ]
128
129
130   return(
131     list(
132       points = cum_mds,
133       eig = list_eigenvalues
134     )
135   )
136 }
```

C.2 Fast MDS

```
fast.mds ← function(
  x,
  l,
  s,
  k,
  metric
){
  # Initial Parametres
  list_matrix = list()
  list_index = list()
  list_mds = list()
  list_mds_align = list()
  list_number_dimensions = list()
  list_eigenvalues = list()

  sub_sample_size = k * s
  n = nrow(x)

  # Partition into p matrices
  # When doing the partitions it can happen that there are so many matrices
  # that s*k < nrow(x_i). In this case, we do a sampling again, otherwise
  # cmdscale have problems

  p = ceiling(l/sub_sample_size)
  observations_division = sample(x = p, size = nrow(x), replace = TRUE)
  observations_division = sort(observations_division)
  min_sample_size = min(table(observations_division))

  while( min_sample_size < sub_sample_size && p > 1){
    p = p - 1
    observations_division = sample(x = p, size = nrow(x), replace = TRUE)
    observations_division = sort(observations_division)
    min_sample_size = min(table(observations_division))
  }

  # Partition into p submatrices
  for(i_group in 1:p){
    ind = which(observations_division == i_group)
    list_matrix[[i_group]] = x[ind, ]
  }

  able_to_do_mds = n/p ≤ l | p == 1


  # We can do MDS
  if(able_to_do_mds == TRUE){
    for (i_group in 1:p) {
      matrix_filter = list_matrix[[i_group]]

```

```
50        # MDS for each submatrix
51        distance_matrix = daisy(
52          x = matrix_filter,
53          metric = metric
54        )
55
56        cmd_eig = stats::cmdscale(
57          d = distance_matrix,
58          k = s,
59          eig = TRUE
60        )
61
62        # Storing the eigenvalues
63        list_mds[[i_group]] = cmd_eig$points
64        list_eigenvalues[[i_group]] = cmd_eig$eig
65
66        # Subsample
67        sample_size = sub_sample_size
68        if(sample_size > length( row.names(matrix_filter ) ) ){
69          sample_size = length( row.names(matrix_filter ) )
70        }
71
72
73        list_index[[i_group]] = sample(
74          x = row.names(matrix_filter),
75          size = sample_size,
76          replace = FALSE
77        )
78
79
80        # Building x_M_align
81        ind_M = which(row.names(x) %in% list_index[[i_group]])
82        if(i_group == 1){
83          x_M_align = x[ind_M, ]
84        }else{
85          x_M_align = rbind(
86            x_M_align,
87            x[ind_M, ]
88          )
89        }
90
91      }
92
93      # M_align: MDS over x_M_align
94      distance_matrix_M = distance_matrix = daisy(
95        x = x_M_align,
96        metric = metric
97      )
98
99      M_align = stats::cmdscale(
100       d = distance_matrix_M,
101       k = s
102     )
103
104     # Global alignment
```

```
105     for(i_group in 1:p){
106       row_names = list_index[[i_group]]
107 
108       ind_M = which(row.names(M_align) %in% row_names)
109       M_align_filter =  M_align[ind_M, ]
110 
111       di = list_mds[[i_group]]
112       ind_mds = which(row.names( di ) %in% row_names)
113       di_filter = di[ind_mds, ]
114 
115       # Alignment
116       procrustes_result =  MCMCpack::procrustes(
117         X = di_filter, #The matrix to be transformed
118         Xstar = M_align_filter, # target matrix
119         translation = TRUE,
120         dilation = TRUE
121       )
122 
123       rotation_matrix = procrustes_result$R
124       dilation = procrustes_result$s
125       translation = procrustes_result$tt
126       ones_vector = rep(1, nrow(di))
127       translation_matrix = ones_vector %*% t(translation)
128 
129 
130       tranformation_di = dilation * di %*% rotation_matrix +
            translation_matrix
131 
132 
133       # Append
134       if(i_group == 1){
135         Z = tranformation_di
136       } else{
137         Z = rbind(
138           Z,
139           tranformation_di
140         )
141 
142       }
143     }
144 
145     row.names(Z) = row.names(x)
146 
147   }else{
148     list_zi <- list()
149     list_index <- list()
150     list_number_dimensions = list()
151     list_eigenvalues = list()
152     for(i_group in 1:p){
153 
154       # Apply the algorithm
155       cmd_eig = fast.mds(
156         x = list_matrix[[i_group]],
157         l = l,
158         s = s,
```

```
159           k = k,
160           metric = metric
161         )
162
163         # Storing MDS and eigenvalues
164         list_zi[[i_group]] = cmd_eig$points
165         list_eigenvalues[[i_group]] = cmd_eig$eig
166
167         # Take a subsample
168         list_index[[i_group]] = sample(
169           x = row.names( list_zi[[i_group]] ),
170           size = k * s,
171           replace = FALSE
172         )
173
174         ind = which( row.names( list_zi[[i_group]] ) %in% list_index
                [[i_group]])
175         submatrix = list_matrix[[i_group]][ind, ]
176
177
178         if(i_group == 1){
179           x_M_align = submatrix
180         } else{
181           x_M_align = rbind(
182             x_M_align,
183             submatrix
184           )
185         }
186       }
187
188
189       distance_matrix_M  = daisy(
190         x = x_M_align,
191         metric = metric
192       )
193
194       M_align = stats::cmdscale(
195         d = distance_matrix_M,
196         k = s
197       )
198
199
200       # Global alignment
201       for(i_group in 1:p){
202         row_names = list_index[[i_group]]
203
204         ind_M = which(row.names(x_M_align) %in% row_names)
205         M_align_filter =  M_align[ind_M, ]
206
207         di = list_zi[[i_group]]
208         ind_mds = which(row.names( di ) %in% row_names)
209         di_filter = di[ind_mds, ]
210
211         # Alignment
212         procrustes_result =  MCMCpack::procrustes(
```

```
213           X = di_filter, #The matrix to be transformed
214           Xstar = M_align_filter, # target matrix
215           translation = TRUE,
216           dilation = TRUE
217         )
218
219         rotation_matrix = procrustes_result$R
220         dilation = procrustes_result$s
221         translation = procrustes_result$tt
222         ones_vector = rep(1, nrow(di))
223         translation_matrix = ones_vector %*% t(translation)
224
225
226         tranformation_di = dilation * di %*% rotation_matrix +
              translation_matrix
227
228
229         # Append
230         if(i_group == 1){
231           Z = tranformation_di
232         } else{
233           Z = rbind(
234             Z,
235             tranformation_di
236           )
237
238         }
239       }
240
241       row.names(Z) = row.names(x)
242
243     }
244
245     return(
246       list(
247         points = Z,
248         eig_normal = list_eigenvalues
249       )
250     )
251   }
```

C.3 MDS based on Gower interpolation

```
1   gower.interpolation.mds ← function(
2     x,
3     l,
4     s
5   ){
6
7     nrow_x = nrow(x)
8     p = ceiling(nrow_x/l)
9     if(p<1) p = 1
10
11    if( p>1 ){
12      # Do MDS with the first group and then use the Gower
```

```
    interpolation formula
sample_distribution = sample(x = p, size = nrow_x, replace =
    TRUE)
sample_distribution = sort(sample_distribution)

# Get the first group
ind_1 = which(sample_distribution == 1)
n_1 = length(ind_1)

# Do MDS with the first group
submatrix_data = x[ind_1, ]

distance_matrix = cluster::daisy(
  x = submatrix_data,
  metric = "euclidean"
)

distance_matrix = as.matrix(distance_matrix)

# MDS for the first group
cmd_eig = stats::cmdscale(
  d = distance_matrix,
  k = s,
  eig = TRUE
)

M = cmd_eig$points
eigen = cmd_eig$eig/nrow(M)
cum_mds = M

# Calculations needed to do Gower interpolation
delta_matrix = distance_matrix^2
In = diag(n_1)
ones_vector = rep(1, n_1)
J = In - 1/n_1*ones_vector %*% t(ones_vector)
G = -1/2 * J %*% delta_matrix %*% t(J)
g_vector = diag(G)
# S = cov(M)
S = 1/(nrow(M)-1)*t(M) %*% M
S_inv = solve(S)

# For the rest of the groups, do the interpolation
for(i_group in 2:p){
  # Filtering the data
  ind_i_group = which(sample_distribution == i_group)
  submatrix_data = x[ind_i_group, ]


  # A matrix
  distance_matrix_filter = pdist::pdist(
    X = submatrix_data,
    Y = x[ind_1, ]
  )

  distance_matrix_filter = as.matrix(distance_matrix_filter)
```

```
66        A = distance_matrix_filter^2
67        ones_vector = rep(1, length(ind_i_group))
68        MDS_i_group = 1/(2*n_1)*(ones_vector %*%t(g_vector) - A) %*%
              M %*% S_inv
69        cum_mds = rbind(
70          cum_mds,
71          MDS_i_group
72        )
73      }
74    }else{
75      # It is possible to run MDS directly
76      distance_matrix = cluster::daisy(
77        x = x,
78        metric = "euclidean"
79      )
80
81      distance_matrix = as.matrix(distance_matrix)
82
83      # MDS for the first groups
84      cmd_eig = stats::cmdscale(
85        d = distance_matrix,
86        k = s,
87        eig = TRUE
88      )
89
90      cum_mds = cmd_eig$points
91      eigen = cmd_eig$eig/nrow_x
92    }
93
94    return(
95      list(
96        points = cum_mds,
97        eig = eigen
98      )
99    )
100 }
```