A Toolbox for Probability Calculus and Optimization

Pedro Melgueira

INESC-ID Tec. Rep. 9/2012 Reference Number 8139
Acknowledgements

I would like to thank INESC-ID Lisboa for providing this opportunity and in particular I would like to thank Professor Dr. Miguel Barão for orienting me in this project and receiving me at his office in Évora on a weekly basis.
# Contents

1 Introduction 4

2 Development 5
  2.1 The BLAS and CBLAS Libraries 5
    2.1.1 Matrix Conventions in BLAS and CBLAS 6
    2.1.2 Other BLAS and CBLAS Conventions 6
  2.2 Tools Used For Development 7

3 Multidimensional Matrices and Probabilities 9
  3.1 Multidimensional Matrices (MD Matrices) 9
  3.2 Probability Distributions and Conditional Probability Functions 11
  3.3 Representing a Probability Distributions as a Multidimensional Matrix 12
  3.4 Using MD Matrices With the C Programming Language 14
  3.5 Vectorization of a MD Matrix into a Single Dimension 15
  3.6 The MDVar Data Structure 16
  3.7 The MDMatrix Data Structure 18

4 Interaction Between MD Matrices 20
  4.1 Non Probability Distribution MD Matrices 20
  4.2 Cross MD Matrix Index Calculation 22
  4.3 MD Matrix Calculation Properties 24
    4.3.1 Addition 25
    4.3.2 Subtraction 25
    4.3.3 Product 26
    4.3.4 Division 27
  4.4 Non Probability Distribution Operation 30

5 Multidimensional Operations 31
  5.1 Scalar Operations 31
    5.1.1 Multiplication 31
    5.1.2 Division 32
    5.1.3 Exponent 32
5.2 Exponential and Logarithm operations .......................... 32
  5.2.1 Exponential ............................................. 33
  5.2.2 Logarithm ............................................. 33
5.3 Euclidean Norm and Dot Product ............................... 34
  5.3.1 Euclidean Norm ......................................... 34
  5.3.2 Dot Product ............................................ 34
  5.3.3 Sum All .................................................. 34
5.4 Probability MD Matrix Marginal ................................ 34
5.5 Sub MD Matrix ................................................ 36
5.6 Cross MD Matrix OP ........................................... 37
  5.6.1 MD Matrix Addition ..................................... 37
  5.6.2 MD Matrix Subtraction ................................ 37
  5.6.3 MD Matrix Product ..................................... 38
  5.6.4 MDMatrix Division ..................................... 38
5.7 Normalization .................................................. 38
5.8 Mixtures ....................................................... 39
  5.8.1 Mixture .................................................. 39
  5.8.2 Exponential Mixture ................................... 40
5.9 Other Operations ............................................... 40
  5.9.1 Array Size ............................................. 40
  5.9.2 MD Matrix Marginal Size ............................... 41
  5.9.3 Get Maximum ........................................... 41
  5.9.4 Get Minimum ........................................... 41
5.10 Information Theory Measures .................................. 41
  5.10.1 Entropy and Joint Entropy .............................. 41
  5.10.2 Conditional Entropy .................................. 43
  5.10.3 Cross Entropy ......................................... 43
  5.10.4 Mutual Information ................................... 44
  5.10.5 Kullback-Leibler Divergence ........................... 45
6 Algorithms .......................................................... 47
  6.1 Bayes Theorem ............................................... 47
  6.2 The Blahut-Arimoto Algorithm ............................... 48
6.3 Standard and Natural Gradient ................................ 49
  6.4 The Gradient Method Using the Natural Gradient ............. 49
  6.5 Minimization of the Kullback Leibler Divergence ............ 51
  6.6 Markov Process ............................................ 52
  6.7 Hidden Markov Model ....................................... 52
7 Final Remarks ........................................................ 54
Chapter 1

Introduction

This project was done in the scope of an integration to investigation grant made available by the Control of Dynamic Systems Group - INESC-ID Lisboa. The project was developed under the supervision of Professor Miguel José Simões Barão.

The project consisted in using the C programming language to build a library to work with multi dimensional matrices that represented probability distributions. Those probability distributions were then used to perform calculations related to information theory like calculating the entropy of a distribution, the Kullback-Leibler divergence between two distributions and implementing the Blahut-Arimoto algorithm to calculate the channel capacity.

All of these operations were made so they were as dynamic as possible. Because of that the operations in this library work with probability distributions with $n \in \mathbb{N}$ random variables. Making this possible presented a challenge that had to be redressed more than one time.

This project also used a library called BLAS to make some scalar-vector calculations. Due to the multi dimensional nature of the operations the BLAS library wasn’t used outside of the scalar-vector operations.
Chapter 2
Development

This project was develop with the C programming language using the default standard, which is C89. For some functions, the BLAS library was used to accelerate certain algebra operations.

2.1 The BLAS and CBLAS Libraries

The BLAS library, which stands for Basic Linear Algebra Subprograms, is a library written initially in Fortran to perform linear algebra calculations such as dot products, matrix multiplications, vector rotations, among others. BLAS as been used to develop many projects including other libraries for other programming languages besides Fortran. To work with BLAS in a C program an interface called CBLAS is used.

Both BLAS and CBLAS libraries are maintained and distributed by Netlib, a software repository for scientific computing. Their source code along with their documentation are at [6].

The functions in the BLAS library work with one dimensional arrays that represent vectors and matrices. The reason why matrices are represented in one dimensional arrays is explained in the next subsection along with an explanation of how to vectorize a matrix.

Most functions in this library are prepared to work with more than one data type. This is because some arrays may be declared of type float or double. There is also two more data types, Scomplex and Complex. These data types represent single and double precision complex types, respectively. The naming convention for the functions in the BLAS library is related to what data type it is written for. For example, the functions that use arrays of floats have their name starting with s, like in scopy() and sswap(), and the functions that use arrays of doubles start with d, like in dcopy() and dswap().

The CBLAS library uses the function from the BLAS library and adds the prefix cblas_ to their name. So dcopy() is cblas_dcopy() in CBLAS.
BLAS also as enumerated types to represent transpose matrices and triangular matrices among others.

This project works with multilinear algebra, so representing matrices according to the BLAS convention is not directly used in this project, however, the convention to vectorize matrices is relevant for a better understanding of the data structure MDMatrix defined at 3.7. This project also doesn’t use complex data types. For these reasons, this report will not discuss these matters in further detail.

2.1.1 Matrix Conventions in BLAS and CBLAS

BLAS only works with arrays, so any matrix needs to be vectorize. Doing this for two dimensional matrices to work with BLAS is actually very simple. Take the following matrix to be vectorized,

\[ A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}. \]

Matrix \( A \) can be vectorize in two ways. They are row major and column major. What vectorization means is taking the values of a matrix and writing them in a vector. First exploring the row major way, this means the values of the matrix will be inserted row be row. The result of the vectorization will be

\[ V = (1, 2, 3, 4, 5, 6). \]  \hspace{1cm} (2.1)

By vectorizing the matrix in a column major way, the values will be inserted column by column. The result will be:

\[ V = (1, 4, 2, 5, 3, 6). \]

The way a multidimensional matrix is represented with a MDMatrix data structure is very similar to this, but a little bit more complicated since the indexes of the matrix also have to be represented. Further details will be in the respective chapter.

2.1.2 Other BLAS and CBLAS Conventions

To start understanding the conventions in BLAS and CBLAS first take the example of a copy of an array to another array.

\[ x \leftarrow y. \]

```c
void cblas_dcopy(int n, double *x, int incx, double *y, int incy);
```
The above function is the mathematical operation that consists in the copy of the contents of array \( y \) to array \( x \). This operation is usually called assignment.

It is observable that the function takes one integer argument, \( n \), that represents the size of the arrays \( x \) and \( y \). It is also observable two integer variables, \( incx \), and \( incy \).

When assigning the values from an array with size \( n \) to another array of the same size, there is no problem as to where to put the values from the first array to the second. However, there are some operations that require that only a few elements are allocated from one array to the other. Such operations are, for example, if there is the need to extract a row from a matrix. Taking the example from 2.1.1, there is the vectorized matrix \( A \) from

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}.
\]

to

\[
V = (1, 4, 2, 5, 3, 6).
\]

If there is the need to extract the first row of the matrix from the vectorized matrix to another array, the elements of the target array in the \( k \)th position will be equal to those in the \( 2k \)th of the source array. This operation would be like in the following C code:

```c
double y[6] = {1, 2, 3, 4, 5, 6};
double x[3];
for (i = 0; i < 3; i++)
  x[i] = y[i*2]
```

So when iterating element by element the increment in each iteration for the array \( y \) will be 1 while in the increment for the \( x \) will be 2. The arguments \( incx \) and \( incy \) of the function \( scopy \) represent that increment. In this specific example the value of \( incx \) would be 1 and \( incy \) would be 2.

This type of increment idea is used in almost every operation in the BLAS library. It is this idea that makes it possible for BLAS to be very efficient at performing vector and matrix operations.

This idea of vectorization and index incrementation served as the starting point for the creation of the data structure MDMatrix.

### 2.2 Tools Used For Development

This project was completely programed under the Ubuntu operating system which is a Linux distribution based on Debian. Due to the fact that the project is somewhat low level programming when comparing to other projects that use more complicated and advanced libraries and resources,
the default text editor at the time of development for Ubuntu, Gedit, was used instead of some other more advanced IDE.

The documentation of this library was written using LaTeX. The IDE used to develop LaTeX documents was Kile.
Chapter 3

Multidimensional Matrices and Probabilities

Throughout this article probabilities are calculated and used in the form of mathematical vectors and matrices. This means that the common symbol $P(X)$ doesn’t represent just the value of a probability where $X$ is equal to an $x$ belonging to the space state of $X$, but represents all of the values $X$ can take from its state space and represents it in a matrix.

This section clarifies how to represent a probability distribution function in a matrix, so it can be used in this package, and explains what are multidimensional matrices and how to use them.

3.1 Multidimensional Matrices (MD Matrices)

Multidimensional matrices, or MD matrices, are a generalization of what a matrix and a vector are.

A vector has many mathematical representations, one of them is

$$V = (v_i) = (v_1, v_2, \ldots, v_n), \ n \in \mathbb{N}, \ i \leq i \leq n.$$  

From the representation above, it is understood that the vector $V$ has $n$ elements which are $(v_1, v_2, \ldots, v_n)$. Likewise, matrices have a similar representations:

$$A = [a_{i,j}] = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}.$$  

$m, \ n \in \mathbb{N}, \ 1 \leq i \leq m, \ 1 \leq j \leq n.$
Just like the vectors example, from above it is understood that the matrix $A$ has $(m, n)$ elements.

Until now there are vectors with one index, therefore one dimension, and matrices with two indexes, two dimensions. Supposing now that a third dimension is represented. That means there are three indexes. Usually, such structure is represented like

$$A = \begin{bmatrix}
a_{i,j,k}
\end{bmatrix}$$

Let $k = 1$

$$A = \begin{bmatrix}
a_{1,1,1} & a_{1,2,1} & \cdots & a_{1,n,1} \\
a_{2,1,1} & a_{2,2,1} & \cdots & a_{2,n,1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1,1} & a_{m,2,1} & \cdots & a_{m,n,1}
\end{bmatrix}$$

Let $k = 2$

$$A = \begin{bmatrix}
a_{1,1,2} & a_{1,2,2} & \cdots & a_{1,n,2} \\
a_{2,1,2} & a_{2,2,2} & \cdots & a_{2,n,2} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1,2} & a_{m,2,2} & \cdots & a_{m,n,2}
\end{bmatrix}$$

... 

Let $k = w$

$$A = \begin{bmatrix}
a_{1,1,w} & a_{1,2,w} & \cdots & a_{1,n,w} \\
a_{2,1,w} & a_{2,2,w} & \cdots & a_{2,n,w} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1,w} & a_{m,2,w} & \cdots & a_{m,n,w}
\end{bmatrix}.$$

$m, n, w \in \mathbb{N}$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq k \leq w$.

The structure above has three indexes $(i, j, k)$ and each element is represented like $a_{i,j,k}$. This structure is often called a matrix array because it is basically a set of $w$ matrices with indices $(i, j)$ in an array (or vector) with $w$ elements indexed by $k$.

This idea can be further generalized to include more then only three indexes. A structure that follows the same idea of a vector, matrix or matrix array, but has more then three indexes is therefore called a multidimensional matrix, or MD matrix for short. By consequence, any vector or matrix can also be called a multidimensional matrix because the general idea of representing a collection of elements by their indexes is the same and there is no need to pay attention to a correct designation based on the number of indexes. A multidimensional matrix may be hard to represent on paper with all its elements if it has more then three indexes, but a general representation without its elements is:

$$A = \left[ a_{i_1,i_2,\ldots,i_n} \right].$$
Above, the MD matrix $A$ has $n$ indexes, therefore, $n$ dimensions. Note that each dimension may also have different sizes.

3.2 Probability Distributions and Conditional Probability Functions

A probability distributions is a function that describes the probability of a random variable taking certain values. Generally these functions are written like

$$P(X = x).$$

Where $X$ is a random variable, $x$ is a value belonging to the state space of $X$, and $P(X = x)$ the probability of $(X = x)$.

Mathematically, a random variable may be either discrete or continuous. Discrete variables may also take values that are not integer number. One such example of a discrete random variable is the probability of a door of some building being open or close at some point of the day. It can be agreed that a variable $X$ is equal to $True$ if the door is open, and equals to $False$ if the door is close. So the state space for this variable is a binary space, $\Omega = \{True, False\}$.

Discrete random variables may, therefore, take other state spaces that are not a group of number but are statements of common language. The example of the door could be a little more specific and instead of having just a $True$ or $False$, the state space could be something like $\Omega = \{“Door is open”, “Door is closed but unlocked”, “Door is closed and locked”\}$.

To represent a probability distribution of a discrete random variable in this package, the state space must be numbered starting from 0. So the example of the door will look like

$$\Omega = \{0, 1, 2\}.$$

Where 0, 1 and 2 indicate the states “Door is open”, “Door is closed but unlocked” and “Door is closed and locked” respectively. The same is true when representing a joint probability function or a conditional probability function with $n$ variables ($n \in \mathbb{N}$).

Numbering the events in the state space will be necessary because the probability distribution functions and conditional probability functions are going to be represented as a multidimensional matrices, and the space states of those matrices will be its indexes. For a more details explanation, see section 3.3.
3.3 Representing a Probability Distributions as a Multidimensional Matrix

Returning to the example of the door in 3.1,

\[ \Omega = \{ \text{“Door is open”}, \text{“Door is closed but unlocked”}, \text{“Door is closed and locked”} \}. \]

Which was represented as

\[ \Omega = \{ 0, 1, 2 \}. \]

It was needed to represent the probability of a certain door at some point of the day taking one of the values from the state space \( \Omega \). This probability can be written like \( P(X = x) \), where \( x \in \Omega \). For this example the probabilities below will be assumed:

\[
\begin{align*}
P(X = 0) &= 0.2. \\
P(X = 1) &= 0.75. \\
P(X = 2) &= 0.05. 
\end{align*}
\]

To represent this function in an MD matrix, the states in \( \Omega \) will serve as the indexes for the matrix, and the values its elements. Therefore the MD matrix will be:

\[
P(X = x) = \begin{pmatrix}
0 & 1 & 2 \\
0.2 & 0.75 & 0.05
\end{pmatrix}.
\]

Suppose now that there is a joint probability distribution with random variables \( X \) and \( Y \). Suppose the following probabilities

\[
\begin{align*}
P(X = 0, Y = 0) &= 0.19. \\
P(X = 1, Y = 0) &= 0.11. \\
P(X = 2, Y = 0) &= 0.14. \\
P(X = 0, Y = 1) &= 0.25. \\
P(X = 1, Y = 1) &= 0.12. \\
P(X = 2, Y = 1) &= 0.19. 
\end{align*}
\]

Above we observe that the variable \( X = x, x \in \{0, 1, 2\} \), therefore \( X \) has a size of 3. Likewise, the variable \( Y = y, y \in \{0, 1\} \) has a size of 2. Like before, the space state of each variable has been numbered starting from 0. Writing this function in an MD matrix the result is:
\[
P(X = x, \ Y = y) = \begin{pmatrix}
\begin{array}{ccc}
0 & 1 & 2 \\
0.19 & 0.11 & 0.14 \\
0.25 & 0.12 & 0.19 
\end{array}
\end{pmatrix}
\]

The process is similar when representing a conditional probability function. Suppose the MD matrix representing \( P(X = x, \ Y = y) = P(X = x \mid Y = y) \). Its values would be approximately:

\[
P(X = 0 \mid Y = 0) = 0.43.
P(X = 1 \mid Y = 0) = 0.25.
P(X = 2 \mid Y = 0) = 0.32.
P(X = 0 \mid Y = 1) = 0.45.
P(X = 1 \mid Y = 1) = 0.21.
P(X = 2 \mid Y = 1) = 0.34.
\]

And its representation in an MD matrix.

\[
P(X = x \mid Y = y) = \begin{pmatrix}
\begin{array}{ccc}
0 & 1 & 2 \\
0.43 & 0.25 & 0.32 \\
0.45 & 0.21 & 0.34 
\end{array}
\end{pmatrix}
\]

For a last example in this section, suppose a probability distribution with four variables: \( X, \ Y, \ Z, \ W \) having a size of 4, 3, 2, 2 respectively. For the sake of simplicity, real probability values will be ignored and replaced by the symbol \( \circ \). The representation of the distribution is shown below:

\[
x \in \{0, 1, 2, 3\}, \ \text{having a size of 4}.
y \in \{0, 1, 2\}, \ \text{having a size of 3}.
z \in \{0, 1\}, \ \text{having a size of 2}.
w \in \{0, 1\}, \ \text{having a size of 2}.
P(X = x, \ Y = y, \ Z = z, \ W = w) =
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\begin{bmatrix}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{bmatrix} & \begin{bmatrix}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{bmatrix}
\end{array}
\]
Note that in this section, all of the structures, regardless of having one, two or more dimensions, were always called MD matrices. This is, again, just to emphasize the reason why, in this article, a vector or matrix is called an MD matrix.

### 3.4 Using MD Matrices With the C Programming Language

Almost every programming language offers a way to represent arrays of arrays. In the C programming language that is done like so:

```c
double x[3][4][8][2]; // An MD matrix with four dimensions.
double ****p; // A pointer to an MD matrix with four dimensions.
```

This two ways to represent MD matrices, although simple to understand, make calculations not very dynamic, or at the very least confusing to work with. Also, by working with variables like this the calculations wouldn’t be dynamic because by representing matrices this way the functions would be limited to a specific number of dimensions.

The ideal way to represent an MD matrix in a truly dynamic and efficient way is to store the information of said matrix in a form of an array with one dimension. This form is preferable because by having everything displace in an unidimensional way, there is no theoretical limit to how many variables an MD matrix can have.

In a last advantage of this method of representation, integrating the CBLAS library would be inefficient and complicated if matrices were represented like in the example above. This is because the functions in the CBLAS library only work with one dimensional arrays.

It is possible to make a cast from a multidimensional array to a single dimensional array (Example below). But this method is not really conventional and one can’t be sure of the displacement of the values in an array in memory. The method gets more confusing and hard to work with if the number of dimensions an array has is very high and it gets impossible to have any representation of indexes.

```c
double x[2][3] = {{1, 2, 3}, {4, 5, 6}};
double *p = (double *) x;
int i, j;
```
To solve these problems two data structures were developed. They are the MDVar and the MDMatrix data structures. These data structures represent random variables and MD matrices respectively. They make possible to articulate all the information of probability distributions in the calculation of statistical operations, information theory operations and a few algorithms.

3.5 Vectorization of a MD Matrix into a Single Dimension

The idea of the vectorization of a matrix (two dimensions) into a single dimension as been describe in 2.1.1. This section describes how to do the vectorization for a MD matrices.

Suppose the following MD Matrix:

\[
P(X = x, Y = y, Z = z) =
\begin{array}{cc}
Y \backslash X & 0 & 1 \\
Z = 0 & 0.05 & 0.1 \\
 & 0.15 & 0.05 \\
Z = 1 & 0 & 1 \\
 & 0.1 & 0.2 \\
 & 0.05 & 0.25
\end{array}
\]

There are three random variables which are \(X\), \(Y\) and \(Z\), and all of them have a size of two. To represent the information of the matrix in a single array, the vectorization can be done in several ways. The values can be written like below:

\[
[0.05, 0.1, 0.15, 0.05, 0.1, 0.2, 0.05, 0.25].
\]

Or

\[
[0.05, 0.15, 0.1, 0.05, 0.1, 0.05, 0.2, 0.25].
\]

Or

\[
\ldots
\]

The various ways the information can be placed always follow some logical way of placing the values. Observing the first example, there are the first two values which are 0.05 and 0.1. These values correspond to the matrix indexes \(P(X = 0, Y = 0, Z = 0)\) and \(P(X = 1, Y = 0, Z = 0)\), so it
can be said that going from the first element to the second element only the X variable changed while the Y and the Z variable remained the same. Continuing from the second to the third variable, values 0.1 to 0.15, it is observed that the X variable changed again from 1 to 0 but the Y variable changed from 0 to 1 this time. Only while changing from the fourth to the fifth element the Z variable changes.

With this in mind, the variable changes can be represented in a table. In said table, the order of the variables is also important. The table will be called MD matrix vectorization table.

<table>
<thead>
<tr>
<th></th>
<th>0.05</th>
<th>0.1</th>
<th>0.15</th>
<th>0.05</th>
<th>0.1</th>
<th>0.2</th>
<th>0.05</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note the first two elements. Below them it is observed that Y = 0 and Z = 0, but X = 0 only for the first element and X = 1 for the second element.

In the second example displayed above of another vectorization, the values change places because the order of the variables is different. Representing that vectorization in the table the result would be:

<table>
<thead>
<tr>
<th></th>
<th>0.05</th>
<th>0.15</th>
<th>0.1</th>
<th>0.05</th>
<th>0.1</th>
<th>0.05</th>
<th>0.2</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

With the second example it is observed the importance of the variable displacement for this table. Because going from the first to the second element only the Y variable changes, that variable is putted in the first place.

### 3.6 The MDVar Data Structure

```c
typedef struct MDVAR_STRUCT {
    unsigned int id, dim;
    int side;
} MDVar;
```

In this library it is important to store the information of the indexes of an MD matrix. Although these data structures were developed to represent probability distributions, they can also be used to represent generic MD matrices.

Ordinarily, an MD matrix has its indexes identified. Those identifications are usually made with letter from the Latin alphabet. Such examples are matrices with indexes x, y, z, or x1, x2, x3, etc. In this library implementation the variables are represented by an unsigned integer value. The
value for that identification is completely independent in an MD matrix. There may be, for example, two variables one with id 0 and other with id 42. The variable \( id \) in the data structure MDVar is where this identification is stored.

To represent an index of an MD matrix it is also important to store its individual size because each index may have different sizes. The variable \( dim \) stores that information.

With these two variables of the structure it is possible to represent the indexes of a generic MD matrix, that is, of an MD matrix that doesn’t represent a probability distribution. If the objective of a representation if just that, then the variable \( side \) may be disregarded provided that another condition is verified in the data structure MDMatrix. (for details see page 4.1).

If the objective is the representation of a probability distribution, then another property of an index must be taken into account. That is its side. The side of an index is an umbrella term that specifies if a random variable is representing a probability distribution or a conditional distribution. (Keep in mind that the random variables of a probability distribution are the indexes of its MD matrix representation.)

For a more specific example suppose the example below:

\[
A = P(X, Y \mid Z)
\]

The MD matrix \( A \) represents the probability of \( X \) and \( Y \) conditioned by \( Z \). There for the random variables \( X \) and \( Y \) are on the left side and the random variable \( Z \) is in the right side.

To represent these variables in the MDVar data structure the variable \( side \) will be equal to value to 0, or the macro \( LEFT\_SIDE \), for the variables \( X \) and \( Y \) and it will be equal to 1, or the macro \( RIGHT\_SIDE \) for the variables \( Z \).

For a general example suppose the representation of the variables in the MD matrix \( A \). (The dimensions are arbitrary unsigned integers):

```c
MDVar x;
x.id = 0;
x.dim = x_dim;
x.side = LEFT\_SIDE;

MDVar y;
y.id = 1;
y.dim = y_dim;
y.side = LEFT\_SIDE;

MDVar z;
z.id = 2;
z.dim = z_dim;
z.side = RIGHT\_SIDE;
```
3.7 The MDMatrix Data Structure

This data structure is where the information of an MD matrix is stored for calculations. This structure may be used to represent generic MD matrices or probability distributions.

Its declaration is:

```c
typedef struct MDMATRIX_STRUCT {
    double *array;
    unsigned int array_size;
    MDVar *vars;
    unsigned int vars_size;
    int pro_dist;
} MDMatrix;
```

The values of an MD matrix need to be vectorized. The result of that vectorization is stored in an array pointed by *array. The size of that array is `array_size`.

The indexes of the MD matrix are stored in an array pointed by `vars`. Its size `vars_size`.

The order the values are stores in *array and the variables are stored in *vars is important. That order should be in accordance with the MD matrix vectorization table. An example is shown below.

The variable `pro_dist` takes the values of the macros `PRO_DIST`, equaling 1, and `NON_PRO_DIST`, equaling 0. What this means is that if the variable is equal to 1, then the MD matrix represents a probability distribution and therefore the value of `side` in each `MDVar` is important and may be important for some operations. This is important because of situations like the examples below.

\[ P(X | Y)P(Y). \]

\[ P(X)P(Y) = P(X, Y). \]

In the first line the value of `pro_dist` for the `MDMatrix` representing \( P(X|Y) \) would be equal to 1 (`PRO_DIST`), while in the second line it would be 0 (`NON_PRO_DIST`) because dividing the probability of \( X \) by the probability of \( Y \) is only equal to the joint probability if the variables are independent. These operations are further explained in section 4.

Below there is an example of the process of writing a probability distribution as an MD matrix. Suppose the following MD matrix is to be inserted using the data structures `MDMatrix` and `MDVar`:

```c
18
```
\[
P(X = x, Y = y) = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} 0.16 & 0.19 & 0.18 \\ 0.20 & 0.04 & 0.23 \end{pmatrix}
\]

First, the matrix needs to be vectorized using the MD matrix vectorization table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0.19</th>
<th>0.18</th>
<th>0.20</th>
<th>0.04</th>
<th>0.23</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Next, the variables of type MDVar are declared:

```c
MDVar x;
x.id = 0;
x.dim = 3;

MDVar y;
y.id = 1;
y.dim = 2;
```

Last step, the matrix of type MDMatrix are declared:

```c
double values[] = {0.16, 0.19, 0.18, 0.20, 0.04, 0.23};

MDMatrix m;
m.array_size = 6;
m.array = values;

m.vars_size = 2;
m.vars = (MDVar*) malloc(sizeof(MDVar) * m.vars_size);
m.vars[0] = x;
m.vars[1] = y;
```

Note the order the variable \( x \) and \( y \) were inserted. It is the same order of the MD matrix vectorization table. The values in `array` are also the same way as the vectorization table.
Chapter 4

Interaction Between MD Matrices

Interaction between MD matrices was one of the most difficult yet interesting problems to solve in this project. It is the concept of having a somewhat standard way to have two MD matrices interacting to perform calculations that allows this library to work with MD matrices in such a way that there is more than one possible vectorization for the same object. It is also this concept that allows MD matrices and functions that work with them to not be bound by any theoretical limit regarding the number of random variables indexing them.

Because MD matrices in this library are needed to represent probability distributions, certain properties must be taken into account. These properties specify what the result between the product of two MD matrices is.

4.1 Non Probability Distribution MD Matrices

The calculations performed in this library always take into account the variables indexing the MD matrices. These calculations are not defined as linear algebra calculations and matrix operations are also not exactly the same as the standard operations in this field of mathematics.

The first thing that needs to be taken into account when doing calculations between MD matrices is if the matrices represent probability distributions or not. This is important because the same operations are used when calculating a matrix product element by element or calculating a joint distribution from a conditional probability and probability distribution.

In section 3.7 it was explained how probability distributions are represented as MD matrices. Up until now, MD matrices are written with the same notation of a probability distribution like:

\[ P(X_1, \ldots, X_n \mid Y_1, \ldots, Y_m) \]
What this means is that the MD matrix is a probability distribution with \( n \) random variables conditioned to \( m \) other random variables.

Sometimes it’s needed to represent MD matrices that don’t necessarily represent a probability distribution. This kind of matrices are sometimes used in calculations. One such example of this kind of calculations is the Bayes Theorem (which is further defined in 6.1). The theorem is:

\[
P(X \mid Y) = \frac{P(Y \mid X)P(X)}{P(Y)}.
\]

Generally, the steps for calculating this are:

\[
\frac{P(Y \mid X)P(X)}{P(Y)} = \frac{P(Y, X)}{P(Y)} = P(X \mid Y).
\]

First there is the product between \( P(Y \mid X) \) and \( P(X) \) which is equal to \( P(Y, X) \), and then there is the division of that distribution by \( P(Y) \) (In the implementation of this library, this is the way the theorem is applied). This way, every MD matrix is a probability distribution, because there is no operation done that doesn’t generate an MD matrix.

But the same calculation can be done using different steps:

\[
\frac{P(Y \mid X)P(X)}{P(Y)} = P(Y \mid X)P'(Y, X) = P'(Y, X)
\]

Note that the first operation made was the division between \( P(Y) \) and \( P(X) \). What happens in this operation is that every element from the first distribution is divided by every element from the second distribution. The result \( P'(Y, X) \) will be equal to \( P(X \mid Y) \) The result is a MD matrix indexed by random Variables \( Y \) and \( X \). Mathematically, this can be represented by:

\[
P'(Y, X) = \begin{pmatrix}
P(Y=0) & \ldots & P(Y=0) \\
P(X=0) & \ddots & \vdots \\
\vdots & \ddots & \vdots \\
P(Y=n) & \ldots & P(Y=n)
\end{pmatrix}.
\]

This is not a joint or conditional probability distribution. It is just an MD matrix containing the result of the division indexed by the random variables that generated it. This may be called a non probability distribution MD matrix, and it is represented by adding the symbol ‘: \( P'(...) \). Note that when representing this type of MD matrix the side of the variables is disregarded. So of these representations \( P'(X \mid Y) \) and \( P'(Y, X) \) both are equivalent but only the latter is used.

Although the last MD matrix has the same information, the calculations that lead to it contained non probability distribution MD matrices. Hence, the result is a non probability distribution MD matrix. (Note: the algorithm for the Bayes theorem implemented in this library was written according to
the first calculation given in these two examples. So its result is a probability
distribution MD matrix.)

This type for MD matrices are used has an auxiliary data structure for
the more complex calculations.

4.2 Cross MD Matrix Index Calculation

The operations between MD matrices are addition, subtraction, product
and division. The four operations share the same algorithm, but the result
is different in function of the operation. The difference between the results is
not only in the values of the array of the MD matrix. The variables indexing
the resulting MD matrices are also different.

The properties that determine variables indexing the resulting MD ma-
trix are specified in section 4.3. They depend on the operation itself and on
the two MD matrices involved in the operation.

In any of the four operations the resulting MD matrix is calculated by
performing an operation between two elements of the MD matrices involved
in it and storing the result in another MD matrix. To understand this one
needs to know that every cross MD matrix index calculation returns an MD
matrix that has a number of indexes greater or equal to the MD matrices
in the operation with the exception of one type of division, which is further
explained in section 4.3

To start understanding the algorithm suppose the following two MD
matrices A and B. Let ρ be one of the four operations. The first thing to
do is the resulting MD matrix. Supposing the group of variables from A
is vA and the group of variables from B is vB, the variables indexing the
resulting MD matrix R will be:

\[ v_R = v_A \cup v_B. \]

The above rule is true for every operation, even if said operation is an
addition or subtraction, with the exception of one case in the division. The
reason why will be further explain in 4.3.4. Continuing, the algorithm will
now iterate through the resulting MD matrix’s array and for each position
will do the operation:

\[ R_{kr} = A_{ka} \rho B_{kb}. \]

The index kr goes through the MD matrix’s R array. The ka and kb are
two indexes in A and B respectively. The index ka is always equal to

\[ kr \mod l_A. \]

Where lA is the length of the MD matrix’s A array.
The only thing left is to know how to calculate the index $kb$. But that calculation is better explained first with an example. So, suppose the definition of $A$ and $B$ is the following and the operation $\rho$ is a product.

$$
A = \begin{pmatrix}
0 & \begin{pmatrix} 0.3 & 0.1 \\ 0.4 & 0.2 \end{pmatrix} \\
1 & \begin{pmatrix} 0.15 & 0.05 \\ 0.7 & 0.1 \end{pmatrix}
\end{pmatrix},
$$

$$
B = \begin{pmatrix}
0 & \begin{pmatrix} 0.3 & 0.1 \\ 0.4 & 0.2 \end{pmatrix} \\
1 & \begin{pmatrix} 0.15 & 0.05 \\ 0.7 & 0.1 \end{pmatrix}
\end{pmatrix},
$$

vectorized:

<table>
<thead>
<tr>
<th>A</th>
<th>0.3</th>
<th>0.1</th>
<th>0.4</th>
<th>0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ka$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>0.15</th>
<th>0.05</th>
<th>0.7</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$kb$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The group of variables are $vA = \{X, Y\}$ and $vB = \{X, Z\}$, so the resulting is $vR = vA \cup vB = \{X, Y, Z\}$. Below there is a vectorization table representing the indexes of $R$.

<table>
<thead>
<tr>
<th>$R$</th>
<th>0.045</th>
<th>0.005</th>
<th>0.06</th>
<th>0.01</th>
<th>0.21</th>
<th>0.005</th>
<th>0.28</th>
<th>0.02</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$Z$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$kr$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

The index $kb$ is the index of MD matrix $B$ that operates with $A$ so that the result of said operation is stored in $R$. The place of storage is $kr$ and $kb$ is calculated in its function. To know $kr$ one must first calculate the value of the index $X$ and $Z$ from $kb$. To know the value of the index $X$, which will be denominated by $X_i$, in any $kr$ one must simply do:

$$
X_i = kr \mod 2,
$$

the modulo 2 is because the dimension of the variable $X$ is 2. To know $Z_i$ from $kr$ is not as simple. There must be a division of $kr$ first. That division is equal to the number of elements each $Z$ covers by one index (that is, $Z = 0$ covers the index 0, 1, 2, 3, therefore the number of elements is 4). One
can say that \( X_i \) is also divided. It is divided by 1, because the number of elements that \( X \) covers is 1.

\[
Z_i = \frac{kr}{4} \mod 2.
\]

Again, its modulo 2 because the dimension of \( Z \) is also 2. Also, these operations are always integer operations, so \( \frac{3}{4} = 0 \).

So now there is the individual value for \( X_i \) and \( Z_i \), which are the variables in \( R \) that are also in \( B \). To know the value of \( kb \) the operation is:

\[
kb = X_i + 2Z_i.
\]

The value \( Z_i \) is multiplied by 2 because, in the MD matrix \( B \), the variable \( Z \) covers two elements. Likewise, \( X_i \) is multiplied by 1.

With this example in mind, let \( j \) be one of the random variables indexing \( B \). Let \( \text{div}_j \) be the value that needs to be divided from variable \( j \), \( \text{mult}_j \) be the value that needs to be multiplied from variable \( j \). And let \( \text{dim}_j \) be the value of the dimension of variable \( i \). The general formula for calculating \( kb \) in function of \( kr \) with \( n \in \mathbb{N} \) variables in \( B \) is:

\[
kb = \sum_{j=0}^{n} \left( \frac{kr}{\text{div}_j} \mod \text{dim}_j \right) \text{mult}_j.
\]

### 4.3 MD Matrix Calculation Properties

Any of the four operations between MD matrices returns another MD matrix. As it was said before, the variables that are going to index the resulting MD matrix are going to depends on the operation itself and on the two MD matrices involved in the operation. One other thing that depends on the operation and the indexing variables is if the resulting matrix is a probability distribution MD matrix or a non probability distribution MD matrix.

The following section defines the properties that specify the resulting random variables and if the resulting MD matrix is or it’s not a probability distribution for each of the four operations.

Mathematically, let \( P(\ldots) \) and \( Q(\ldots) \) be probability distribution MD matrices. Let \( vP \) and \( vQ \) be the group of random variables indexing \( P(\ldots) \) and \( Q(\ldots) \) respectively. Let also \( P'(\ldots) \) and \( Q'(\ldots) \) be non probability distribution MD matrices and \( vP' \) and \( vQ' \) be the group of random variables indexing them respectively. \( R(\ldots) \) or \( R'(\ldots) \) will represent the resulting MD matrices.
4.3.1 Addition

As it was explain, its not necessary that the two MD matrices involved in this operation to have the same indexing random variables, but the only way an addition returns a probability distribution MD matrix is if the two MD matrices involved in the operation are probability distributions.

The only way to get the result $R(\ldots)$ is if the two MD matrices involved in the operation are probability distributions, $P(\ldots) + Q(\ldots)$, and if $vP = vQ$. If $vP \neq vQ$ then:

$$R'(\ldots) = P(\ldots) + Q(\ldots).$$

If any of the MD matrices is a non probabilities distribution the result will also be a non probability distribution $R'(\ldots)$, regardless of their indexing variables:

$$R'(\ldots) = P'(\ldots) + Q(\ldots),$$

$$R'(\ldots) = P(\ldots) + Q'(\ldots),$$

$$R'(\ldots) = P'(\ldots) + Q'(\ldots).$$

In any of these cases the resulting indexing random variables are always:

$$vR = vP \cup vQ = vP = vQ,$$

$$vR' = vP' \cup vQ,$$

$$vR' = vP \cup vQ',$$

$$vR' = vP' \cup vQ'.$$

4.3.2 Subtraction

The subtraction between MD matrices is basically the same as the Addition regarding these properties. So the following operation:

$$R(\ldots) = P(\ldots) - Q(\ldots).$$

is true if and only if the condition $vQ = vP$ applies. Else:

$$R'(\ldots) = P(\ldots) - Q(\ldots).$$

In the other cases:

$$R'(\ldots) = P'(\ldots) - Q(\ldots),$$

$$R'(\ldots) = P(\ldots) - Q'(\ldots),$$

$$R'(\ldots) = P'(\ldots) - Q'(\ldots).$$
In any of these cases the resulting indexing random variables are always:

\[ v_R = v_P \cup v_Q = v_P = v_Q, \]
\[ v_{R'} = v_{P'} \cup v_Q, \]
\[ v_{R'} = v_P \cup v_Q', \]
\[ v_{R'} = v_{P'} \cup v_Q'. \]

### 4.3.3 Product

Unlike addition and subtraction, the product between MD matrices takes into account the side the variables are in (left or right). This has the objective of performing such calculation has:

\[ P(X \mid Y)P(Y) = P(X, Y). \]

In the example, the side of the variable \( Y \) is right in \( P(X \mid Y) \) and left in \( P(Y) \) and \( P(X, Y) \).

From the calculations above it is possible to take the various properties for the product between two MD matrices. But first, the situations where the product results in a non probability distribution MD matrix. The most simple example, if \( v_P \cap v_Q = \emptyset \) the resulting MD matrix will be a non probability distribution MD matrix, regardless if the two MD matrices involved in the operation are or are not probability distributions. Mathematically, when \( v_P \cap v_Q = \emptyset \)

\[ R'(\ldots) = P(\ldots)Q(\ldots), \]
and
\[ v_{R'} = v_P \cup v_Q. \]

If that is not the case, there is still the situation where one or both MD matrices are non probability distribution MD matrices. Mathematically, when \( v_P \cap v_Q \neq \emptyset \)

\[ R'(\ldots) = P'(\ldots)Q(\ldots), \]
\[ R'(\ldots) = P(\ldots)Q'(\ldots), \]
\[ R'(\ldots) = P'(\ldots)Q'(\ldots). \]
and
\[ v_{R'} = v_{P'} \cup v_Q, \]
\[ v_{R'} = v_P \cup v_Q', \]
\[ v_{R'} = v_{P'} \cup v_Q'. \]
Now there are the situations when both MD matrices involved in the operation are probability distributions and $v_P \cap v_Q \neq \emptyset$. For these situations the side of the variable is going to be important. Like the example at the beginning of the section, by having a conditional probability distribution and a marginal distribution one can calculate the joint probability distribution.

$$P(X \mid Y)P(Y) = P(X, Y).$$

This is, of course, generalized for more variables and the following situations are observed:

$$P(X, Z \mid Y)P(Y) = P(X, Y, Z),$$

Generally, in a product between MD matrices if both are probability distributions and $v_P \cap v_Q \neq \emptyset$, the side of the indexing random variables in the resulting MD matrix will be given by the following properties:

- If the random variable is only in one MD matrix.
  - The resulting variable will be in the same side as it was in the MD matrix involved in the operation.

- If the random variable is in both MD matrices.
  - The resulting variable will be in the right side if both MD matrices involved in the operation have the variable in the right side.
  - The resulting variable will be in the left side if in any of the MD matrices involved in the operation the variable is in the left side.

4.3.4 Division

Division works a lot like the product but it has one particular case discussed in the end of this section. Like in the product, this operation takes the side of the variables into account. This is observed in the operation:

$$\frac{P(X, Y)}{P(Y)} = P(X \mid Y).$$

The properties, mathematically, when $v_P \cap v_Q = \emptyset$, are like in the product:

$$R'(...) = \frac{P(...)}{Q(...)}.$$

Again, the operation above is made regardless if the MD matrices are or are not probability distributions. Like in the product operation, when
\( vP \cap vQ \neq \emptyset \) and if any or both MD matrices aren’t probability distributions the result is a non probability distribution MD matrix.

\[
R'(\ldots) = \frac{P'(\ldots)}{Q(\ldots)},
\]

\[
R'(\ldots) = \frac{P(\ldots)}{Q'(\ldots)},
\]

\[
R'(\ldots) = \frac{P'(\ldots)}{Q'(\ldots)}.
\]

and

\[
vR' = vP' \cup vQ,
\]

\[
vR' = vP \cup vQ',
\]

\[
vR' = vP' \cup vQ'.
\]

Now the situations when both MD matrices involved in the operation are probability distributions and \( vP \cap vQ \neq \emptyset \). Like in the product operation, the side of the variable is going to be important. Like the example at the beginning of this section, the following is observed:

\[
\frac{P(X, Y)}{P(Y)} = P(X \mid Y),
\]

\[
\frac{P(X, Y, Z)}{P(Y)} = P(X, Z \mid Y),
\]

\[
\frac{P(X, Y, Z \mid W)}{P(Y, W)} = P(X, Z \mid Y, W).
\]

Generally, in a division between MD matrices if both are probability distributions and \( vP \cap vQ \neq \emptyset \), the side of the indexing random variables in the resulting MD matrix will be given by the following properties:

- If the random variable is only in one MD matrix.
  - The resulting variable will be in the same side as it was in the MD matrix involved in the operation.

- If the random variable is in both MD matrices.
  - The resulting variable will in the right side regardless of where it was before.
In particular, it is important to notice the following operation:

\[
\frac{P(X, Y)}{P(X \mid Y)} = P(Y).
\]

This situation is the only situation where the resulting MD matrix is smaller than any of the MD matrices involved in the operation. To understand this first note the same operation made with non probability MD matrices:

\[
\frac{P'(X, Y)}{P'(X \mid Y)} = P'(X, Y).
\]

First, the side of the variable \(Y\), is not important in the resulting MD matrix. This is because that MD matrix isn’t a probability distribution. But looking at the first example, it can be developed like:

\[
\frac{P(X, Y)}{P(X \mid Y)} = \frac{P(X \mid Y)P(Y)}{P(X \mid Y)} = P(Y),
\]

so basically, by dividing \(P(X, Y)\) by \(P(X \mid Y)\), if the result was to be indexed by the random variables \(X\) and \(Y\), the values of the MD matrix’s array would be equal like:

\[
(X = 0, \ Y = 0) = (X = 1, \ Y = 0) = \cdots = (X = n, \ Y = 0),
\]

\[
(X = 0, \ Y = 1) = (X = 1, \ Y = 1) = \cdots = (X = n, \ Y = 1),
\]

\[
\cdots,
\]

\[
(X = 0, \ Y = m) = (X = 1, \ Y = m) = \cdots = (X = n, \ Y = m).
\]

To eliminate this repetition the resulting MD matrix will just be indexed by \(Y\) and it will be equal to the array where \(X = 0\).

Of course, this is generalized for more random variables.

\[
\frac{P(X, Y, Z)}{P(X, Z \mid Y)} = P(Y),
\]

\[
\frac{P(X, Y, Z)}{P(X \mid Y, Z)} = P(Y, Z).
\]

The general idea is that if in an MD matrix division the divisor has variables on the right side that are on the left side of the dividend, then this particular case is used and the resulting MD matrix will be indexed by the variables in question.
4.4 Non Probability Distribution Operation

This operation is used when the algorithm for the cross MD matrix index calculation is used but the operation itself is not one of the four basic operations. This happens, for example, when calculating the cross entropy between two probability distributions. The algorithm must relate two elements of different distributions, operate them and store the result in another position of the resulting MD matrix. (Cross entropy defined in section 5.10)
Chapter 5

Multidimensional Operations

This library implements many linear and multilinear algebra operations. They are scalar operations, operations between two MD matrices, among others. This sections details each one.

Notation: \( A \) an MD matrix. \( \alpha \in \mathbb{R} \).

5.1 Scalar Operations

Scalar operations are any calculation that takes a real value and operates that value with every element of an MD matrix.

The scalar operations that are possible to do are the multiplication, division and the exponent. The multiplication and division functions, that are documented below, use the CBLAS library.

5.1.1 Multiplication

\[ A \leftarrow \alpha A. \]

```c
void MDMatrixScalarMultiplication(int array_size, double alpha, double *array);
```

Arguments:

- int array_size - Size of array to be multiplied.
- double alpha - Scalar to multiply by.
- double *array - Pointer to array to be multiplied.

This function receives an array of doubles and multiplies its elements by alpha. This function uses the CBLAS library, using the function cblas_dscal that take the same arguments as MDMatrixScalarMultiplication.
5.1.2 Division

$$A \leftarrow \frac{1}{\alpha} A.$$ 

```c
void MDMatrixScalarDivision(int array_size, double alpha, double *array);
```

Arguments:

- int array_size - Size of array to be divided.
- double alpha - Scalar to divide by.
- double *array - Pointer to array to be divided.

This function has the same arguments as MDMatrixScalarMultiplication and in essence it is just a call of MDMatrixScalarMultiplication with the argument alpha being 1/alpha. So it is equivalent to call:

```c
MDMatrixScalarMultiplication(array_size, 1/alpha, array);
```

5.1.3 Exponent

$$A \leftarrow (a_k^\alpha), a_k \in A.$$ 

```c
void MDMatrixScalarExponent(int array_size, double alpha, double *array);
```

Arguments:

- int array_size - Size of array.
- double alpha - Scalar.
- double *array - Pointer to array.

This function uses the library math.h of the C standard library to use the function pow(). MDMatrixScalarExponent calculates each element in array to the power of alpha.

5.2 Exponential and Logarithm operations

These operations are similar to scalar operations. They are performed in each separate element of an array.
5.2.1 Exponential

\[ A \leftarrow (a_k^a), \ a_k \in A \]

```c
void exponential(int array_size, double *array, double exp_base);
```

Arguments:
- `int array_size` - Size of array.
- `double *array` - Pointer to array.
- `double exp_base` - Integer with the base of the exponential.

This function calculates the `exp_base` to the power of each element in the array. Example:

```c
double arr[] = {2, 4, 6};
exponential(3, arr, 2);
// arr[0] = 4;
// arr[1] = 16;
// arr[2] = 36;
```

This function uses the library math.h of the C standard library to use the function `pow()`.

5.2.2 Logarithm

\[ A \leftarrow \log_a (a_k), \ a_k \in A \]

```c
void logarithm(int array_size, double *array, double log_base);
```

Arguments:
- `int array_size` - Size of array.
- `double *array` - Pointer to array.
- `double exp_base` - Integer with the base of the logarithm.

Like `exponential()`, this function calculates the logarithm of each element in the array. This function uses the library math.h of the C standard library to use the function `log()`. Each element is calculated first with the base 10 logarithm and divided by `log(exp_base)`.
5.3 Euclidean Norm and Dot Product

5.3.1 Euclidean Norm

double euclideanNorm(MDMatrix m);

This function takes an MDMatrix as its argument and calculates its euclidean norm. This function disregards the number of indexes and if \( m \) is a probability distribution or not and deals with every element in \( m.array \) as a single dimension vector.

5.3.2 Dot Product

double dotProduct(MDMatrix m1, MDMatrix m2);

This function calculates the dot product between \( m1 \) and \( m2 \). To do the operation the function uses \textit{probabilityMDMatrixProduct()} to first calculate the multiplication element by element between the two matrices (Hadamard Product). After that, the function uses the \textit{sumAll()} to make the summation of every element in the resulting matrix of the product operation.

Like the Euclidean Norm, this function disregards the number of indexes and if the MD matrix is a probability distribution or not. But both matrices need to have the same variables or it will return -1. For matrices that have more than one index the operation is applied as if it were for a matrix with one index.

5.3.3 Sum All

double sumAll(MDMatrix m);

This function sums every element of \( m.array \). This is used as a step of the function \textit{dotProduct}.

5.4 Probability MD Matrix Marginal

MDMatrix probabilityMDMatrixMarginal(MDMatrix mdmatrix, int *indexes, int indexes_size);

Arguments:

- MDMatrix mdmatrix - Matrix where the marginal is to be calculated.
- int *indexes - Array with designation of the indexes in which the marginal is to be calculated from in mdmatrix.
- int indexes_size - Size of array *indexes.
This function follows the idea of the calculation of the marginal of a probability distribution function. The function receives an MD Matrix as an argument and calculates the marginal of the indexes in `indexes` in `mdmatrix`. Below are a few examples of this function if use.

Even if the MD matrix received by argument is not a probability distribution the operation is applied in the same way.

Marginal of $X$ in $P(X,Y)$:

$$\sum_{y \in Y} P(X,Y) = P(X)$$

```
// mdmatrix = P(X, Y)
// MDVar: X. id == 0 and Y. id == 1
int indexes[] = {0};
MDMatrix res = probabilityMDMatrixMarginal(mdmatrix, indexes, 1);
```

Marginal of $X,Z$ in $P(X,Y,Z)$:

$$\sum_{y \in Y} P(X,Y,Z) = P(X,Z)$$

```
// mdmatrix = P(X, Y, Z)
// MDVar: X. id == 0, Y. id == 1 and Z. id == 2
int indexes[] = {0, 2};
MDMatrix res = probabilityMDMatrixMarginal(mdmatrix, indexes, 2);
```

Marginal of $X,Y$ in $P(X,Y,Z,W)$:

$$\sum_{z \in Z, w \in W} P(X,Y,Z,W) = P(X,Y)$$

```
// mdmatrix = P(X, Y, Z, W)
// MDVar: X. id == 0, Y. id == 1, Z. id == 2 and W. id = 3
int indexes[] = {0, 1};
MDMatrix res = probabilityMDMatrixMarginal(mdmatrix, indexes, 2);
```

Marginal of $X,Z$ in $P(X,Y|Z,W)$:

$$\sum_{y \in Y, w \in W} P(X,Y|Z,W) = P(X|Z)$$

```
// mdmatrix = P(X, Y | Z, W)
// MDVar: X. id == 0, Y. id == 1, Z. id == 2 and W. id = 3
int indexes[] = {0, 2};
MDMatrix res = probabilityMDMatrixMarginal(mdmatrix, indexes, 2);
```
5.5 Sub MD Matrix

```
MDMatrix subMDMatrix(MDMatrix m, int *indexes, int *values, int indexes_size);
```

Arguments:

- MDMatrix mdmatrix - Matrix where the sub matrix is going to be extracted from.
- int *indexes - Array indexes to be removed.
- int *values - Array with the values each index in *indexes is going to take.
- int indexes_size - Size of array *indexes and *values.

This function selects a number of elements where indexes[i] == values[i], places them in an MD matrix and returns it. To understand the objective of the function take the following example. Suppose the matrix:

\[
A = \begin{pmatrix}
Y & 1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix}
\]

Vectorized and placed in a MDMatrix structure it would be written like below (In a simplified way).

```
// m. array == {1, 2, 3, 4, 5, 6};
m.vars[0].id = 1  // Y
m.vars[1].id = 0  // X
```

Know suppose that only the second row in the matrix is needed. This means that the contents of the matrix that are needed all have X = 1. So to call the function subMDMatrix() the code below would be written.

```
\actal{\%\% m, the matrix from above.}
int indexes[] = {0};
int values[] = {1};
MDMatrix res = subMDMatrix(m, indexes, values, 1);
```

This means that res will have the elements where the variable with id 0 is equal to 1.

A more general example, would be.

```
\actal{\%\% s, some mdmatrix with more then two variables}
int indexes[] = {2, 1};
int values[] = {1, 0};
MDMatrix res = subMDMatrix(s, indexes, values, 2);
```

In this example the resulting MD matrix would have been the elements of s where the variable with id 2 is equal to 1 and the variable with id 1 is equal to 0.
5.6 Cross MD Matrix OP

This algorithm implements what was defined in chapter 4. The four basic functions (+, -, *, /) along with few other functions call crossMDMatrixOP() with different values for its last argument, which is defined in the specification below.

\[
\text{MDMatrix crossMDMatrixOP(MDMatrix m1, MDMatrix m2, double (*op)(double, double));}
\]

Arguments:

- MDMatrix m1 - One matrix for the operation.
- MDMatrix m2 - The other matrix for the operation.
- double (*op)(double, double) - Pointer to operation function.

The operation functions are all defined in the files cross_matrix_op.h and cross_matrix_op.c. Those functions are the only ones used in this library. Their objective is to defined what is the operation that needs to be done in the MD matrix interaction. For example, if the operation is addition, the function will add one element of m1 to its respective element of m2, and so forth.

5.6.1 MD Matrix Addition

\[
R(\ldots) = P(\ldots) + Q(\ldots).
\]

\[
\text{MDMatrix MDMatrixAddition(MDMatrix m1, MDMatrix m2);}
\]

Arguments:

- MDMatrix m1 - One matrix for the operation.
- MDMatrix m2 - The other matrix for the operation.

This function calculates the addition between MD matrices m1 and m2. The operation is done in accordance with 4.3

5.6.2 MD Matrix Subtraction

\[
R(\ldots) = P(\ldots) - Q(\ldots).
\]

\[
\text{MDMatrix MDMatrixSubtraction(MDMatrix m1, MDMatrix m2);}
\]

Arguments:
• MDMatrix \( m_1 \) - One matrix for the operation.

• MDMatrix \( m_2 \) - The other matrix for the operation.

This function calculates the subtraction of MD matrix \( m_1 \) by \( m_2 \). The operation is done in accordance with 4.3

5.6.3 MD Matrix Product

\[
R(...)=P(...)Q(...).
\]

MDMatrix probabilityMDMatrixProduct (MDMatrix m1, MDMatrix m2);

Arguments:

• MDMatrix \( m_1 \) - One matrix for the operation.

• MDMatrix \( m_2 \) - The other matrix for the operation.

This function calculates the product between MD matrices \( m_1 \) and \( m_2 \). The operation must be done in accordance with 4.3.

5.6.4 MDMatrix Division

\[
R(...)=\frac{P(...)}{Q(...)}.
\]

MDMatrix probabilityMDMatrixDivision (MDMatrix m1, MDMatrix m2);

Arguments:

• MDMatrix \( m_1 \) - One matrix for the operation.

• MDMatrix \( m_2 \) - The other matrix for the operation.

This function calculates the matrix division of \( m_1 \) by \( m_2 \). The operation must be done in accordance with 4.3.

5.7 Normalization

MDMatrix normalizeMDMatrix (MDMatrix mdmatrix);

When working with probability distributions, it is common that some operation may not return a distribution that sum to 1. What this means is that there may be MD matrices that should represent a probability distribution, but when all of their values are summed the result is different then 1.
To correct this, the distributions need to be normalize. What this means is that each element of the distribution is divided by the sum of all elements. Take the following example of the array of an MD matrix that does not sum 1. The array is represented as an Euclidean vector for a more simple demonstration.

\[ \vec{x} = (0.6, 2.4). \]

\[ x = \sum \vec{x} = 3. \]

\[ \frac{\vec{x}}{x} = \left( \frac{0.6}{3}, \frac{2.4}{3} \right) = (0.2, 0.8). \]

This function disregards if the MD matrix is a probability distribution or not and works with the matrix’s array as a single dimension vector.

## 5.8 Mixtures

It is possible to calculate a mixture distribution between probability distributions. This mixture is parameterized by a real number. In this library there are two mixtures defined.

### 5.8.1 Mixture

The mixture is defined as:

\[ M(P, Q) = \alpha P + (1 - \alpha)Q, \]

\[ 0 \leq \alpha \leq 1. \]

Note that the same operation could be done for more then two distributions. For example, for three distribution the definition would be:

\[ M(P, Q, R) = \alpha P + \beta Q + (1 - \alpha - \beta)R \]

\[ 0 \leq \alpha + \beta \leq 1 \]

In this library there the mixture is implemented for only two distributions. Its specification is:

```
MDMatrix mixture(MDMatrix m1, MDMatrix m2, double alpha);
```

Arguments:

- MDMatrix m1 - One matrix for the operation, the \( P \) in the definition.
• MDMatrix m2 - The other matrix for the operation, the Q in the definition.
• double alpha - The alpha of the definition.

5.8.2 Exponential Mixture

This mixture follows the same idea of the mixture but the $\alpha$ is applied to the power of the distributions and it’s not multiplied. This mixture must be normalized to ensure that the sum of all of its elements is equal to 1.

$$E(P, Q) = \frac{P^\alpha Q^{1-\alpha}}{\Gamma}$$

$$0 \leq \alpha \leq 1$$

The constant $\Gamma$ is needed for the normalization of the operation. There for:

$$\Gamma = \sum P^\alpha Q^{1-\alpha}$$

```c
MDMatrix exponentialMixture(MDMatrix m1, MDMatrix m2, double alpha);
```

Arguments:

• MDMatrix m1 - One matrix for the operation.
• MDMatrix m2 - The other matrix for the operation.
• double alpha - The alpha of the definition.

5.9 Other Operations

5.9.1 Array Size

This functions takes an MD matrix and calculates the size of its array using only the dimension of its variables. This method is used in most operations, such as the MD matrix products and divisions. The reason for this is because these functions build their returning result by first calculating its variables and then, from that information, they calculate the size of the array.

The calculation in itself is just the product of the dimensions of every variable in the MD matrix. There may be the case where a function returns an MD matrix without variables that represents only one value. In this case, that is, when the MD matrix has no variables, this functions returns 1.

```c
int arraySize(MDMatrix md);
```
Arguments:

- MDMatrix m - Matrix for the operation.

5.9.2 MD Matrix Marginal Size

This function is very much like `arraySize()`, but instead of returning the size for the array of the whole MD matrix, it only returns the size for a marginal, which is chosen by the arguments.

To choose the marginal, the arguments are similar to the ones in the function `MDMatrixMarginal()`. The specification for this function is:

```c
int MDMatrixMarginalSize(MDMatrix m, int *indexes, int indexes_size);
```

Arguments:

- MDMatrix m - Matrix where the marginal is to be calculated.
- int *indexes - Array with the indexes for marginal.
- int indexes_size - Size of array *indexes.

5.9.3 Get Maximum

This function takes an MD matrix and returns the smallest value.

```c
double getMaximum(MDMatrix m);
```

Arguments:

- MDMatrix m - Matrix for the operation.

5.9.4 Get Minimum

This function takes an MD matrix and returns the greatest value. This function is analogous to `getMaximum()`.

```c
double getMinimum(MDMatrix m);
```

Arguments:

- MDMatrix m - Matrix for the operation.

5.10 Information Theory Measures

5.10.1 Entropy and Joint Entropy

In information theory, Entropy, or Shannon’s Entropy in honor of its discoverer Claude Shannon, represents the uncertainty associated with a
random variable. The concept of entropy was first introduced by Ludwig Boltzmann, a Austrian physicist, in the XIX century. His definition, known as the H-Theorem, represents the entropy of an ideal gas in an irreversible process.

The information theory’s entropy was defined by Claude Shannon in his paper “A Mathematical Theory of Communication” [3]. It is defined as:

\[ H(X) = -\sum_X P(X) \log P(X) \]

Above the function represents the entropy of the random variable \( X \) following a distribution \( P \). Note that the logarithm can be calculated in any base.

It is possible to calculate the entropy of more then one random variable in a distribution \( P \). That operation is called the joint entropy. The operation in itself is the same but with more variables. Below there is the specification with \( n \) random variables:

\[ H(X_1, \ldots, X_n) = -\sum_{X_1, \ldots, X_n} P(X_1, \ldots, X_n) \log P(X_1, \ldots, X_n) \]

When calculating the entropy for a probability distribution or joint probability distribution the result is a real number. But it is possible to calculate the entropy for a conditional function. Suppose the function \( P(X \mid Y) \). When calculating the entropy for this function, which will be represented like \( H(P(X \mid Y)) \) the result will be a vector like:

\[
H(P) = \begin{bmatrix}
H(X \mid Y = 0) \\
H(X \mid Y = 1) \\
\vdots \\
H(X \mid Y = n)
\end{bmatrix}
\]

This operation is not to be confused with conditional entropy. The result of calculating the conditional entropy of the same function is a real number and not a vector.

By definition, when calculating the entropy \( 0 \log 0 = 0 \), for any base. Although \( \log 0 \) is not defined this convention exists because there may be distributions in which a few events have 0 probability.

Below there is the specification for the entropy in this library.

```c
MDMatrix entropy (MDMatrix m, double log_base);
```

Arguments:

- MDMatrix m - The matrix containing the distribution.
- double log_base - The base of the logarithm in the calculations.
If the result isn’t an MD matrix and it is just a real number, the resulting MD matrix will not have any variables and it will have only one value in its array. The resulting real number is the value in question.

This function is calculated using the Cross Entropy algorithm. The cross entropy algorithm is called in the following way:

crossEntropy(m, m, log_base);

5.10.2 Conditional Entropy

The conditional entropy is defined as:

\[ H(X | Y) = \sum_y P(Y) H(X | Y = y). \]

Note that:

\[ H(X | Y = y) = H(P(X | Y)). \]

Again, calculating \( H(X | Y) \) is different from calculating \( H(P(X | Y)) \). The first is the conditional entropy of random variable \( X \) condition by \( Y \) and the second is the entropy of \( X \) for every \( Y \).

MDMatrix conditionalEntropy(MDMatrix conditional, MDMatrix marginal, double log_base);

Arguments:

- MDMatrix conditional - An MD matrix containing the conditional distribution (The \( P(X | Y) \) of the example).
- MDMatrix marginal - An MD matrix containing marginal distribution (The \( P(Y) \) of the example).
- double log_base - The base of the logarithm in the calculations.

5.10.3 Cross Entropy

The Cross Entropy (also known as Kerridge inaccuracy) is calculated between two distributions of the same variables. Supposing there is a distribution \( P \) and a distribution \( Q \). The Cross Entropy is:

\[ H(P, Q) = - \sum_X P(X) \log Q(X). \]

Note that calculating \( H(P, Q) \) and \( H(Q, P) \) produces different results. Calculating the Cross Entropy for distributions with more then one variable or conditional distributions follows the same rules expressed for the calculation of the Entropy.
Note that the convention $0 \log 0 = 0$ also applies.

Below there is the specification for this library:

```c
MDMatrix crossEntropy(MDMatrix m1, MDMatrix m2, double log_base);
```

Arguments:

- MDMatrix m1 - An MD matrix containing one of the distributions (The $P$ of the definition).
- MDMatrix m2 - An MD matrix containing the other distribution (The $Q$ of the definition).
- double log_base - The base of the logarithm in the calculations.

5.10.4 Mutual Information

The mutual information expresses the quantity of information between two variables. In other words, this function expresses how must a random variable explains the other. It is defined as:

$$I(X ; Y) = -\sum_X \sum_Y P(X, Y) \log \frac{P(X, Y)}{P(X)P(Y)}.$$ 

The mutual information can be used to calculate what is known as the capacity of a channel. The capacity of a channel can be calculated using the Blahut-Arimoto Algorithm that is further explained in chapter 6.1.

This measure can be associated with entropy. Examples of such associations are shown below:

- $I(X ; Y) = H(X) - H(X \mid Y),$
- $I(X ; Y) = H(Y) - H(Y \mid X),$
- $I(X ; Y) = H(X) + H(Y) - H(X, Y).$

The specification for this library:

```c
MDMatrix mutualInformation(MDMatrix joint, int index1, int index2, double log_base);
```

Arguments:

- MDMatrix joint - An MD matrix with a probability distribution.
- int index1 - An id of a random variable in $joint$ (The $X$ of the definition for example).
- int index2 - An other id of a random variable in $joint$ (The $Y$ of the definition for example).
• double log_base - The base of the logarithm in the calculations.

This function receives a probability distribution in joint and calculates the mutual information for two random variables. Note that this calculation is only possible for two variables at the time. It is possible to calculate the mutual information for more than two variables but there isn’t a direct approach to this problem.

This function uses the Kullback-Leibler divergence to for calculation.

\[ I(X ; Y) = D( P(X, Y) \| P(X)P(Y) ) \].

5.10.5 Kullback-Leibler Divergence

The Kullback-Leibler divergence, also known as information gain and relative entropy, expresses the difference between two probability distributions. Its definition is:

\[ D( P \| Q ) = P(X) \log \frac{P(X)}{Q(X)} \].

Where \( P \) and \( Q \) are two probability distributions. If the distributions are equal then \( D( P \| Q ) = 0 \). This divergence can be informally thought as a distance measure, but it is not a true distance in a mathematical sense, \( D( P \| Q ) \neq D( Q \| P ) \).

The specification for this library is:

\[
\text{MDMatrix kullbackLeiblerDivergence} (\text{MDMatrix m1, MDMatrix m2, double log_base});
\]

Arguments:

• MDMatrix m1 - One probability distribution for the calculation (The \( P \) in the definition, for example).

• MDMatrix m2 - The other probability distribution for the calculation (The \( Q \) in the definition, for example).

• double log_base - The base of the logarithm in the calculations.

Note that by calculating the Kullback-Leibler divergence between, for example, the distribution \( P(X \mid Y) \) and \( Q(X \mid Z) \), the result is not a scalar. It is the MD matrix \( P(Y, Z) \). This MD matrix does not represent the joint probability of random variables \( Y \) and \( Z \). It represents the divergence of \( X \) between distribution \( P \) and \( Q \) for a certain \( Y \) and \( Z \).

When the result of the operation is a real number, which happens, for example when calculating the divergence between \( P(X) \) and \( Q(X) \) or \( P(X, Y) \) and \( Q(X, Y) \), the value needs to be extracted from the MD matrix that is returned by the function. This can be done simply by doing:

45
MDMatrix ret = kullbackLeiblerDivergence(m1, m2, 2);

x = ret.array[0];
Chapter 6

Algorithms

6.1 Bayes Theorem

The Bayes theorem, named after Thomas Bayes, is given by

\[ P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}. \]

Given a conditional probability distribution \( P(Y|X) \) and the distribution \( P(X) \), it is possible to calculate the posterior probability distribution \( P(X|Y) \). The distribution \( P(X) \) is called the prior knowledge in Bayesian statistics.

This theorem follows a few steps to perform the calculations from the first two distributions to the result. First, the product between the two arguments

\[ P(Y|X)P(X) = P(Y, X), \]

and secondly, the division of \( P(Y, X) \) by the marginal \( P(Y) \). To obtain the marginal \( P(Y) \) the following operation is made

\[ \sum_X P(Y, X) = P(Y). \]

Finally, for the final step, the division of \( P(Y, X) \) by the marginal \( P(Y) \) is

\[ P(X|Y) = \frac{P(Y, X)}{P(Y)}. \]

This function also applies the Bayes theorem in situations with more than two variables. One such example of this situations is shown below

\[ P(X|Y, Z) = \frac{P(Y, Z|X)P(X)}{P(Y, Z)}. \]
The specification for the Bayes Theorem function:

```c
MDMatrix bayesTheorem (MDMatrix conditional, MDMatrix prior);
```

Arguments:

- MDMatrix conditional - The MD matrix corresponding to the distribution \( P(Y|X) \) in the example.
- MDMatrix prior - The MD matrix corresponding to the distribution \( P(X) \) in the example.

### 6.2 The Blahut-Arimoto Algorithm

The Blahut-Arimoto Algorithm is an iterative algorithm that calculates what is known as the channel capacity. The term channel capacity was first introduced by Claude Shannon and it specifies an asymptotic limit on the maximum rate at which information can be conveyed reliably over a channel. The algorithm was defined by Richard Blahut in his paper “Computation of Channel Capacity and Rate-Distortion Functions” [4].

In information theory a channel is described by a probability transition matrix of the form \( Q(K|J) \). The matrix represents the probability of receiving the output \( k \)th upon the transmission of input \( j \)th. The channel capacity is given in Blahut’s paper by:

\[
C = \max_{p \in P^n} I(p, Q)
\]

where \( P^n \) is the set of all probability distributions on the channel input. Basically, the algorithm finds the optimal distribution \( P^n \) that maximizes the mutual information between random variables \( K \) and \( J \).

The algorithm is iterative because, as it is described in the paper, most analytical solutions can’t be found for a communication channels with large input/output alphabets.

The specification in this library for this algorithm is:

```c
double BlahutArimotoAlgorithm (MDMatrix pd_input, MDMatrix prob_transition_matrix, double epsilon, double explog_base);
```

Arguments:

- MDMatrix pd_input - The initial estimative.
- MDMatrix prob_transition_matrix - The channel’s transition matrix.
- double epsilon - The symbol \( \epsilon \) in the graphic.
- double explog_base - The base of the logarithms.
6.3 Standard and Natural Gradient

In this library, every time a function needs to calculate a gradient it does so numerically. This is because, most of the times, the analytical solution for a gradient of certain functions is too complicated or specific for a certain number of random variables.

To understand the numerical calculation of a gradient, first it is important to understand the numerical calculation of a derivative of a scalar function. Suppose the following function:

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]

The derivative of this function in a point \( a \) of the domain of \( f \) is given by:

\[
\lim_{h \to 0} \frac{f(a + h) - f(a)}{a + h - a} = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}.
\]

The \( h \) in the expression above represents an infinitesimal number close to 0, that is why the expression is written as a limit. So basically, calculating the derivative is the same as calculating the slope of a line passing in the points \((a, f(a))\) and \((a + h, f(a + h))\).

The gradient of a function \( g : \mathbb{R}^k \) to \( \mathbb{R} \) in a point of its domain \( a \) can be written like:

\[ \nabla g(a) = \left( \frac{dg}{dx_1}, \ldots, \frac{dg}{dx_n} \right), \]

where the elements \( x_k \) represent each variable of \( g \).

In this library, the constant \( h \) is a real value, usually taking values like 0.0000001 or lower. The precision depends on the architecture of the machine that the library was compiled for. Every function that calculated a derivative in this library however, takes this value as an argument.

To calculate the natural gradient of a function \( f \) in some point of its domain \( a \), which is defined in [1] and [2], the following expression is used:

\[ \nabla f = a \circ \nabla f - a(a \cdot \nabla f) \]

6.4 The Gradient Method Using the Natural Gradient

The gradient method is an iterative algorithm that calculates one maximum or one minimum of a function. This algorithm is usually used when
methods that include the calculation of derivative functions isn’t an option, usually because they get too complicated.

The method is called gradient descent if the objective is to calculate the minimum of a function and gradient ascent is the objective is to calculate the maximum of a function. The gradient descent is defined as

\[ x_{n+1} = x_n - \alpha \nabla f(x_n) \]

And the gradient ascent as

\[ x_{n+1} = x_n + \alpha \nabla f(x_n) \]

Where \( x_k \) is an arbitrary point in the function’s domain and \( \alpha \) is a real constant referred to as step. The step can change in each iteration.

The algorithm start with an initial estimative \( x_0 \). Each iteration calculates the gradient of the function being optimized in that point and then it moves in that direction. It is important to note that if a function as more than one maximum or minimum, the one that will be found is the one closer to the initial estimative.

The method defined alone can’t be used in the computation of an optimization. This is because the method lacks a way of stopping the algorithm and it also lacks a method to know what the step constant will be in each iteration. Because of this, this method is used with both step calculation and stop conditions adapted to each problem.

In this library the calculations of the step constant and the stop conditions are defined as in [1]. The step must be in the interval:

\[ \left( \frac{1}{\min(\alpha) - x_k \cdot \nabla f}, \frac{1}{\max(\alpha) - x_k \cdot \nabla f} \right) \]

Where:

\[ \alpha = \{0, \nabla f^1, \ldots, \nabla f^n\} \]

For the stop condition, a real value is given by argument. That value is referred to as error. The following operation must be lesser then then the error for the algorithm to stop:

\[ \tilde{\nabla} f \cdot \nabla f \]

In this library, the function that calculates the standard gradient is:

```cpp
MDMatrix standardGradient (double (*fun)(MDMatrix), MDMatrix point, double h);
```

Arguments:

- double (*fun)(MDMatrix) - The functions to calculate the gradient from.
• MDMatrix point - The point in the function where the gradient is to be calculated.
• double h - The precision to numerically calculate the gradient.

The function that calculates the natural gradient is:

```c
MDMatrix naturalGradient(MDMatrix std_grad, MDMatrix point, double h);
```

Arguments:
• MDMatrix std_grad - The value of the standard gradient at point.
• MDMatrix point - The point in the function where the gradient is to be calculated.

6.5 Minimization of the Kullback Leibler Divergence

In [1], the minimization of the Kullback Leibler Divergence is calculated by finding a $P$ that has the smallest possible value in $D(P||Q)$, with $Q$ being a target distribution. The solution for this problem is obvious because first, the value of the K-L divergence is always $\geq 0$ and, second, if $P = Q$, then $D(P||Q) = 0$. However, there may be situations where the distribution $P$ is not completely free. In the example in [1] the distribution in $P$ is $P(X,Y) = P(Y|X)P(X)$.

To minimize the Divergence, the gradient method algorithm is applied, but instead of it being applied to the whole $P$ distribution it is only applied to $P(X)$. Therefore the solution for the minimization will be the distribution $P(X)$ such as the value of $D(P(X)P(Y|X)||Q(X,Y))$ is the smallest possible.

The specification for this library is:

```c
MDMatrix minimizeKullbackLeiblerDivergence(MDMatrix target, MDMatrix transition, MDMatrix point, double h, double error, double log_base);
```

Arguments:
• MDMatrix target - The $Q$ distribution in the above description.
• MDMatrix transition - The $P(Y|X)$ in the above description.
• MDMatrix point - The initial estimative for $P(X)$.
• double h - The precision for the calculation of the gradient.
• double error - The bound that serves as condition to stop the algorithm.
• log_base - The base of the logarithm in the calculations.

In a last note, it is important to note that the algorithm is made to make the same type of calculation for distributions with more than two variables, like \( P(X,Y,Z,W) \) being generated by \( P(X,Y)P(Z,W|X,Y) \).

### 6.6 Markov Process

This function applies the Markov Process using a known distribution and a sequence of observations that can have more than one property. In the end, the function displays the last probability distribution it calculated.

```c
MDMatrix MarkovProcess(MDMatrix knownDist, MDMatrix iniP, int **it_sequence, int it_sequence_size, int it_sequence_size_by_element);
```

**Arguments:**

- MDMatrix knownDist - The known distribution representing the known probabilities.
- MDMatrix iniP - The initial estimative.
- int **it_sequence - The sequence of iterations.
- int it_sequence_size - The size of the sequence.
- int it_sequence_size_by_element - The size of each elements in the sequence.

### 6.7 Hidden Markov Model

This function applies the Hidden Markov Model using a known distribution and a transition matrix. The function iterates through a sequence containing the observations. In the end it displays the last probability distribution it calculated.

```c
MDMatrix HiddenMarkovModel(MDMatrix knownDist, MDMatrix transitionDist, MDMatrix iniP, int **it_sequence, int it_sequence_size, int it_sequence_size_by_element);
```

**Arguments:**

- MDMatrix knownDist - The known distribution representing the known probabilities.
- MDMatrix transitionDist - The transition distribution.
- MDMatrix iniP - The initial estimative.
- int **it_sequence - The sequence of iterations.
- int it_sequence_size - The size of the sequence.
- int it_sequence_size_by_element - The size of each element in the sequence.
Chapter 7

Final Remarks

With this project I learned a lot about information theory which was new to me and now is a subject I find interesting. I was able to understand some of the various concepts and used them in this project.

The most interesting part of this project was to be able to explore a way to solve multidimensional problems using a very dynamic approach. The whole process that eventually lead to what is described in chapter 4 took same time. Most of that time was spend in calculations and trial and error with the objective of figuring out the best way of having a dynamic way of MD matrix interaction.

The programming aspect of this project itself didn’t present such a great challenge because the project was made to be a low level implementation and aside from BLAS and CBLAS, no other libraries were used.

The mathematical aspect of this project, on the other hand, was same what more challenging. Although I have knowledge in the area of multi variable differential calculus and linear algebra, some of the concepts didn’t became clear to me at first sight. Only after a more careful examination was I able to understand some calculations that lead to the implementation of some algorithms.

In a last remark, the integration to investigation grant was an excellent opportunity were I learned a lot. This grant was especially interesting because it allowed me to have a first contact with academical publications and also to use what I learned in those academical publications for this project.
Bibliography


