First-order methods to recover the measurement operators and states of bipartite quantum correlations

by

V.P.H. Goverse

To obtain the degree of Bachelor of Science in Applied Mathematics and Applied Physics at the Delft University of Technology

Student number:	4595351
Supervisors:	Dr. D. de Laat (EEMCS)
	Dr. D. Elkouss Coronas (TNW)
Other committee members:	Dr. N.V. Budko (EEMCS)
	Prof. Dr. G.A. Steele (TNW)

Delft, June 2021

ABSTRACT

In this thesis, we start with giving a mathematical description of bipartite quantum correlations and how they are built up in the Tensor model. This is needed because we want to recover the state and the operators when only the bipartite quantum correlation is known. In the literature, there are see-saw algorithms to recover the state, but they are limited to only lower dimensions [1]. In this thesis we explore an alternative approach, where we directly minimize the function

$$f(\psi, \{E_s^a\}, \{F_t^b\}) = \sum_{a,b,s,t} \left(P(a,b|s,t) - \psi^* (E_s^a \otimes F_t^b)\psi\right)^2. \tag{1}$$

Here, $P(a,b|s,t)$ is the bipartite correlation, ψ is the state vector, and E_s^a and F_t^b are the POVMs. Furthermore, $\otimes$ is the Kronecker product and * indicates the conjugate transpose of a vector. These variables are subject to constraints and some of them can easily be transformed into penalty functions. The matrices E_s^a and F_t^b have to be Hermitian positive semidefinite, for which we parameterize them by their Cholesky decompositions. The gradient of this (now unconstrained) problem can be explicitly determined with the use of Wirtinger calculus. This offers an elegant way to determine the gradient of real-valued functions with complex variables. Also, a total description of Wirtinger calculus is also given, including a proof that the gradient indeed points towards the direction of the steepest incline. We use first-order methods like gradient descent with backtracking line search and momentum-based gradient descent to find a minimum solution of the equation. If the cost function converges towards zero, we assume that the variables converge to a correct state and measurement operators.

These methods can find large correlations of approximate 3000 separate variables in 1.5 hours and are able to find many different other correlations and states. The algorithm had some problems finding the operators and state of a family of correlations that had four inputs and two outputs. For some correlations the algorithms found states and operators of lower dimension than the correlations were build with.

CONTENTS

1

INTRODUCTION

Is it really true that we are still unable to predict how a system behaves if we know everything about it? The first time I encountered the probabilistic nature of quantum mechanics, I felt frustrated. It seems unfair. Nevertheless, until now it appears to be true. It gets even weirder when we start introducing entanglement. It turns out that if two particles are entangled, they can influence each other even faster than the speed of light. This has recently been experimentally shown at the Delft University of Technology. [2]

In this thesis we investigate what happens when we have these entangled particles and start measuring them. When we have a finite set of measurement devices for both particles, we can generate a probability distribution by repeating the measurement enough times. These probability distributions are called bipartite quantum correlations. This thesis we start with these bipartite quantum correlations and try to recover the state vector and measurement operators. The state vector describes if and how the particles are entangled and the measurement operators represent the measurements. The recovery of this state is done with several different gradient descent methods. We do this with a cost function that incorporates the constraints as penalties and Cholesky decomposition.

In Chapter 2 we formally introduce the bipartite quantum correlation, the state, and the different measurement operators (Observable, PVM and POVM). We also look at the oldest XOR game called CHSH and build the bipartite quantum correlation from the operators.

Next, in Chapter 3 we introduce the Wirtinger calculus. The two Wirtinger derivatives turn out to be really useful for calculating the gradient of real-valued functions with complex variables. We also prove that the Wirtinger gradient points towards the direction of the steepest ascent.

After that, in Chapter 4 we show that it is possible to write the problem of finding P(O)VMs and the state as a constraint optimization problem. Then we turn the constraints into penalty functions that are added to the cost function. We calculate the gradient of the cost function with the penalty terms, using the Wirtinger derivatives.

Subsequently, in Chapter 5 we give several different first-order methods to solve the optimization problem. We also look into second-order algorithms that could be benefi-

cial for the computation time. These are however not implemented.

Finally, in Chapter 6 we implement the algorithms to find the P(O)VM and the state of several bipartite quantum correlations. The bipartite quantum correlations that are examined are CHSH, synchronous with four inputs and two outputs, and correlation-based. Also, all different first-order algorithms in Chapter 5 are compared on a specific case.

This thesis is written as the final part of the double bachelor's degree in Applied Mathematics and Applied Physics at the Delft University of Technology.

Enjoy reading this thesis. I hope you have as much fun reading it, as I had when combining the newly learned mathematics and physics that were required for this thesis.

2

BIPARTITE QUANTUM CORRELATIONS

In this chapter, we introduce bipartite quantum correlations. This is done by first explaining what a bipartite correlation is, followed by an explanation of the quantum part.

First, in Section 2.1 we introduce the bipartite correlation. This is explained as a probability distribution that arises from two parties (Alice and Bob) that both get a question, which they have to answer. Following that, in Section 2.2, we investigate a classical bipartite correlation. The main idea of this type of correlation is that it only takes into account classical phenomena. Next, in Section 2.3 we look at the state and several measurement operators. We start by giving the definition of the state and of an observable, which is used to measure that state. Then, two more measurement operators are defined, which are needed for the bipartite quantum correlation. Subsequently, in Section 2.4 we can finally define the bipartite quantum correlation. Also, the local dimension is mentioned, which turns out to be extremely important and determines if we can find the states and the operators of bipartite quantum correlation. In the last Section 2.5 we introduce the CHSH game, which we use in Section 6.2 to test the algorithms.

We should mention that this chapter is mostly repeating the literature. The ideas and concepts mentioned in Sections 2.1, 2.2, and 2.4 come from [3] and [1]. The quantum ideas and definitions on state and operators come from [4]. In the last Section 2.5 the ideas from [5] and [6] are used and further build on.

2.1. BIPARTITE CORRELATIONS

In this thesis, we explore bipartite quantum correlations. To understand how we can interpret these correlations, we first look at a thought experiment of Alice, Bob, and a third independent referee. The third independent referee asks a question $s \in S$ to Alice and $t \in T$ to Bob. Bob and Alice are not aware of what the other person's question is and cannot communicate. The possible answers are $a \in A$ for Alice and $b \in B$ for Bob. The set of all combinations of answers and questions is $\Gamma = A \times B \times S \times T$. In this thesis we assume

that Γ is finite. When we ask questions (s,t) we want to establish the probability that the answers to the respective questions are (a,b). In short, what is the bipartite correlation $P(a,b|s,t)$? These bipartite correlations have to satisfy a few basic properties that are given in Definition 2.1.1.

Definition 2.1.1 (Bipartite Correlations)**.** A function P with domain $\Gamma = A \times B \times S \times T$ is a bipartite correlation if:

1. $P(a,b|s,t) \geq 0$ for all $(a,b,s,t) \in \Gamma$,
2. $\sum_{a,b} P(a,b|s,t) = 1$ for all $(s,t) \in S \times T$,
3. (Non-signaling) $\sum_{b} P(a,b|s,t_1) = \sum_{b} P(a,b|s,t_2)$ for all $(a,s,t_1,t_2) \in A \times S \times T \times T$,
4. (Non-signaling) $\sum_{a} P(a,b|s_1,t) = \sum_{a} P(a,b|s_2,t)$ for all $(b,s_1,s_2,t) \in B \times S \times S \times T$.

The first two conditions follow directly from being a probability distribution. The last two conditions are known as non-signaling conditions. They make sure that there the question that is asked to Bob does not influence the answer for Alice and vice versa. Lastly, it can be noted that $P(a,b|s,t)$ can be thought of as a 4d-tensor or, in other words, a matrix of matrices.

2.2. CLASSICAL BIPARTITE CORRELATION

In the classical case we don not use quantum phenomena. In the most general case it is possible to write $P(a,b|s,t)$ as a convex sum of several $P_{A,i}(a|s), P_{B,i}(b|t)$, when we assume that Γ is finite. The correlation can then be written as

$$P(a,b|s,t) = \sum_i \lambda_i P_{A,i}(a|s) P_{B,i}(b|t). \tag{2.1}$$

Here $P(a|s)$ and $P(b|t)$ are the conditional chances for the answer a and b to the questions. Furthermore, $\sum_i \lambda_i = 1$ and $\lambda_i \geq 0 \forall i$.

Example 1. An example of a classical bipartite correlation is the probability distribution that arises when we have Alice and Bob in separate rooms and independently answering questions. Both players have a fair dice and a fair coin. Then the referee can ask them both two questions:

1. How many eyes are on top after rolling the dice?
2. Is head or tails showing after a coin toss?

There are 6 possible answers $1,2,\ldots,6$ to the first question with equal probability. $P(\#\text{ eyes}|\text{How many eyes?}) = \frac{1}{6}$. The second question has 2 answers with probability $P(\text{Heads}|\text{Heads or Teals?}) = P(\text{Tails}|\text{Heads or Teals?}) = \frac{1}{2}$. So far it is hopefully clear and nothing new has been said. However it changes when we start asking them both a question at the same time. Then the probability becomes the product of their chances. So if we ask Alice how many eyes and Bob which side then the probability of Alice getting 4

eyes and Bob getting heads is $P(4, \text{Heads}|\text{How many eyes?}, \text{Heads or Teals?}) = P(4|\text{How many eyes?}) \cdot P(\text{Heads}|\text{Heads or teals?}) = \frac{1}{6} \cdot \frac{1}{2}$.

If the reader is interested in a more detailed categorization of classical bipartite correlations, the reader is advised to take a look at [1].

2.3. MEASUREMENT OPERATORS

Before we can fully understand the bipartite quantum correlation, we first have to repeat some of the physics of quantum states. We start by noting that quantum states cannot be measured directly, but always require a measurement operator to know something about that state. However, when measuring a state, we also change the state. But what is this state?

Definition 2.3.1 (Pure state)**.** Let d be the dimension of the Hilbert space $\mathbb{C}^d$, then a vector $\psi \in \mathbb{C}^d$ is a pure state if $\|\psi\|_2 = 1$.

Most physicists are also familiar with an observable to measure quantum states ψ. In this case it is defined by Definition 2.3.2.

Definition 2.3.2 (Observable)**.** Let d be the dimension of the Hilbert space $\mathbb{C}^d$, then a $d \times d$ matrix A is an observable if A is Hermitian.

When we apply an observable X to a state ψ the state collapses to an eigenvector v_i of X and the measurement device outputs the corresponding eigenvalue of X. The probability that this happens will be discussed later.
Since we know that X is Hermitian, we can apply the spectral composition to X. In this case, this is equal to

$$X = \sum_i \lambda_i v_i v_i^*. \tag{2.2}$$

Here λ_i are the eigenvalues, v_i are the corresponding eigenvectors($\|v_i\|_2 = 1$) and v_i^* denotes the conjugate transposed of v_i. The set of matrices $\{v_i v_i^*\}_i$ form a PVM. This is defined as the state in Definition 2.3.3.

Definition 2.3.3 (PVM)**.** Let d be the dimension of the Hilbert space and let I = $\{1 : n\}$ be an index set. Then a PVM is a set $\{A^i\}_{i \in I}$ where A^i is $d \times d$ matrices for all i, with the following properties:

1. $\sum_i A^i = I$
2. $A^i \quad \forall i \in I$ is a Hermitian matrix and positive semidefinite
3. A_i is a projection matrix $\forall i \in \{1, \ldots, d\}$.

The projection matrix is defined as a square matrix A, with the property $A^2 = A$. This can be easily checked for the $v_i v_i^*$ for a fixed i. We use the fact that v_i is an eigenvector and we can see,

$$(A^i)^2 = (A^i)(A^i) = (v_i v_i^*)(v_i v_i^*) = (v_i (v_i^* v_i) v_i^*) = (v_i v_i^*) = A^i. \tag{2.3}$$

However, not all PVMs have rang$(A^i) = 1$. It is also possible for a PVM to have a higher rang, this happens when an eigenvalue of the observable has a algebraic multiplicity that is larger then one.

Now we can use the PVMs to calculate the probability of each one of the eigenvalues to be measured. This is given by one of the postulates of quantum mechanics.

$$\begin{aligned} P(v_i \text{ is measured}) &= \|A^i|\psi\rangle\|^2 \\ &= \langle\psi(A^i)^H|A^i\psi\rangle \\ &= \langle\psi|(A^i)^2|\psi\rangle \\ &= \langle\psi|A^i|\psi\rangle \\ &= \langle\psi|v_i v_i^*|\psi\rangle \\ &= \|\langle v_i,\psi\rangle\|^2. \end{aligned} \tag{2.4}$$

Here the bra-ket notation is used which means that $\psi\rangle$ is a vector, and $\langle\psi|$ is the conjugated transposed of that vector. To get an even better understanding of the relationship between the PVMs and observables, we look at an Example 2 with an observable.

Example 2. Let us start with an observable $X = 4e_1e_1^* + 2e_2e_2^* = \left(\begin{smallmatrix}4&0\\0&1\end{smallmatrix}\right)$, where e_1 and e_2 are the standard basis in $\mathbb{C}^2$ and a state $|\psi\rangle = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$. Then the eigenvalues of the observable are 4 and 2 and the respective eigenvectors are e_1 and e_2. The probability of eigenvalue $\lambda_1 = 4$ being measured depends on the state ψ and can be calculated as,

$$\|\langle e_1,\psi\rangle\|^2 = (1\cdot\tfrac{1}{\sqrt{2}})^2 + (0\cdot\tfrac{1}{\sqrt{2}})^2 = \tfrac{1}{2}. \tag{2.5}$$

After the measurement, the state has collapsed to e_1, so $\psi = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$. If we apply the observable again, we measure 4, with probability 1.

Lastly we define the POVM, the operator that we are meanly interested in. It is quite similar to Definition 2.3.3, but with the last requirement relaxed. It can be seen as a generalization of a PVM.

Definition 2.3.4 (POVM)**.** Let d be the dimension of the Hilbert space and let I = $\{1,\dots,n\}$ be an index set. Then a POVM is a set $\{A^i\}_{i\in I}$ where A^i is $d\times d$ matrices for all i, with the following properties:

1. $\sum_i A^i = I$
2. $A^i \quad \forall\, i \in I$ is a Hermitian matrix and positive semidefinite

Remark. All operators and states are elements of a Hilbert space with a certain dimension d. This dimension d is important for later parts of this thesis.

2.4. DEFINITION OF BIPARTITE QUANTUM CORRELATIONS

A bipartite quantum correlation is a correlation created with the use of a quantum state ψ. This state is then given to Alice and Bob. Subsequently, they use this state to answer respectively questions s or t. Two models describe this correlation; the Tensor model and the Commuting model. In this thesis we focus on the Tensor model.

In the Tensor model, the state that is passed to both Alice and Bob is described as $\psi \in \mathbb{C}^d \otimes \mathbb{C}^d$. For a dimension $d \in \mathbb{N}$. The $\otimes$ denotes the Kronecker product, or the tensor product. The tensor product is defined as, the following.

Definition 2.4.1 (Tensor product)**.** Let $A \in \mathbb{C}^{2\times 2}$ and $B \in \mathbb{C}^{2\times 2}$, then

$$A \otimes B = \begin{pmatrix} A_{11}B & A_{12}B \\ A_{21}B & A_{22}B \end{pmatrix}.$$

For larger matrices A and B the tensor product scales as expected.

A bipartite quantum correlations is build with two components, the state $\psi \in \mathbb{C}^{d^2}$ and the POVM $\{E_s^a \otimes F_t^b\}_{(a,b)\in A\times B} \in \mathbb{C}^{d^2\times d^2}$. Here d is the local dimension. The state ψ represents two (possibly entangled) states, and the POVM $(E_s^a \otimes F_t^b)$ is the Kronecker product of two POVMs. It should be noted that the Kronecker product of two POVMs is still a POVM, since the Kronecker product preserves the properties of a POVM (Hermitian, positive semidefinite, and summing to the identity matrix).

Now let us finally define the bipartite quantum correlation:

Definition 2.4.2 (Bipartite Quantum Correlation)**.** If Alice uses the POVM $\{E_s^a\}_{a\in A}$ to answer question $s \in S$ and Bob uses the POVM $\{F_t^b\}_{b\in B}$ to answer question $t \in T$, the probability to achieve the answers (a,b) to the questions (s,t) is given by

$$P(a,b|s,t) = \langle\psi|E_s^a \otimes F_t^b|\psi\rangle. \tag{2.6}$$

2.5. THE CHSH GAME

In a famous paper by Clauser, Horne, Shimony, and Holt in 1969 [6] the authors describe a thought experiment that will help us get a better understanding of a bipartite quantum correlation. We use the setting to test our algorithm for lower dimensional correlations. In this thought experiment we have a question set $S = \{0,1\}$ and an answer set $A = \{0,1\}$ for Alice. For Bob we have $T = \{0,1\}$ and $A = \{0,1\}$. Important to note here is that there are 2 questions for each player and 2 possible answers for each player as well.

In this case Alice and Bob play a game.The goal of this game is to satisfy the equation,

$$a \oplus b = s \wedge t, \tag{2.7}$$

where $\oplus$ denotes a XOR function (which is 0 if $a = b$ and 1 if $a \neq b$) and $\wedge$ denotes an AND function (which is 1 if $s = t = 1$ and is 0 in all other cases). The probability of each question combination (s,t) to occur is equal for all four question combinations and therefore is equal to 25%.

When we try to win as a classical player, we can find the maximal win percentage is 75%. An example strategy could be to answer 0 to all questions. The result would then be the following,

$$\begin{aligned} 0 = a_0 \oplus b_o = s_0 \wedge t_0 = 0 \\ 0 = a_0 \oplus b_0 = s_1 \wedge t_0 = 0 \\ 0 = a_0 \oplus b_o = s_0 \wedge t_1 = 0 \\ 0 = a_0 \oplus b_0 \neq s_1 \wedge t_1 = 1, \end{aligned}$$

where it is clear that 3 out of the 4 answers are correct and one is incorrect, making the winning percentage 75%. This is because the classical problem is overconstrained, because at least one equation is always incorrect.

Now if we look at a quantum strategy for this game, we first start of with giving both parties the EPR pair [7], which is named after Einstein, Podolsky and Rosen, and is equal to

$$\psi = \frac{|0\rangle|0\rangle + |1\rangle|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{2.8}$$

The observables for Alice and Bob are expressed in terms of Pauli matrices, the three Pauli matrices are:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The observables for Alice are $E_0 = X$ for question 0 and $E_1 = Y$ for question 1. For Bob the observables are $F_0 = (X - Y)/\sqrt{2}$ for question 0 and $F_1 = (X + Y)/\sqrt{2}$ for question 1. These can be used to determining the PVMs with spectral decomposition. This is done for the first E_0 as an example and can be done similarly for E_1, F_0 and F_1.

We start of by determining the eigenvalues of E_0, this can be done with the characteristic polynomial $(-\lambda)^2 - 1 = 0$. So the eigenvalues are $\lambda_0 = 1$ and $\lambda_1 = -1$. Then we can calculate the unit vector that spans null space of $(E_0 - \lambda_0 I) = \left(\begin{smallmatrix} -1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$ which is the eigenvector $v_0 = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right)$. Similarly, we can calculate for λ_1 that $v_1 = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right)$. If we now use that the PVMs are the outer product of the eigenvectors, we get

$$E_0^0 = v_0 v_0^* = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}, \quad E_0^1 = v_1 v_1^* = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{2.9}$$

The same can be done for E_1, F_0 and F_1, that are given below for the sake of completeness

$$E_1^0 = \begin{pmatrix} \frac{1}{2} & -\frac{i}{2} \\ \frac{i}{2} & \frac{1}{2} \end{pmatrix}, \quad E_1^1 = \begin{pmatrix} \frac{1}{2} & \frac{i}{2} \\ -\frac{i}{2} & \frac{1}{2} \end{pmatrix}, \tag{2.10}$$

$$F_0^0 = \begin{pmatrix} \frac{1}{2} & \frac{1+i}{2\sqrt{2}} \\ \frac{1-i}{2\sqrt{2}} & \frac{1}{2} \end{pmatrix}, \quad F_0^1 = \begin{pmatrix} \frac{1}{2} & -\frac{1+i}{2\sqrt{2}} \\ -\frac{1-i}{2\sqrt{2}} & \frac{1}{2} \end{pmatrix}, \tag{2.11}$$

$$F_1^0 = \begin{pmatrix} \frac{1}{2} & \frac{1-i}{2\sqrt{2}} \\ \frac{1+i}{2\sqrt{2}} & \frac{1}{2} \end{pmatrix}, \quad F_1^1 = \begin{pmatrix} \frac{1}{2} & -\frac{1-i}{2\sqrt{2}} \\ -\frac{1+i}{2\sqrt{2}} & \frac{1}{2} \end{pmatrix}. \tag{2.12}$$

These PVMs can be used to finally create our bipartite quantum correlation with Equation 2.6. This bipartite quantum correlation is hard to display since it is a d4 tensor. However, we can show four times a quarter of it to show the whole correlation, which is equal to:

$$P_{\text{CHSH}}(a,b|0,0) = \begin{pmatrix} \frac{1}{4}+\frac{1}{4\sqrt{2}} & \frac{1}{4}-\frac{1}{4\sqrt{2}} \\ \frac{1}{4}-\frac{1}{4\sqrt{2}} & \frac{1}{4}+\frac{1}{4\sqrt{2}} \end{pmatrix}, \tag{2.13}$$

$$P_{\text{CHSH}}(a,b|1,0) = \begin{pmatrix} \frac{1}{4}+\frac{1}{4\sqrt{2}} & \frac{1}{4}-\frac{1}{4\sqrt{2}} \\ \frac{1}{4}-\frac{1}{4\sqrt{2}} & \frac{1}{4}+\frac{1}{4\sqrt{2}} \end{pmatrix}, \tag{2.14}$$

$$P_{\text{CHSH}}(a,b|0,1) = \begin{pmatrix} \frac{1}{4}+\frac{1}{4\sqrt{2}} & \frac{1}{4}-\frac{1}{4\sqrt{2}} \\ \frac{1}{4}-\frac{1}{4\sqrt{2}} & \frac{1}{4}+\frac{1}{4\sqrt{2}} \end{pmatrix}, \tag{2.15}$$

$$P_{\text{CHSH}}(a,b|1,1) = \begin{pmatrix} \frac{1}{4}-\frac{1}{4\sqrt{2}} & \frac{1}{4}+\frac{1}{4\sqrt{2}} \\ \frac{1}{4}+\frac{1}{4\sqrt{2}} & \frac{1}{4}-\frac{1}{4\sqrt{2}} \end{pmatrix}. \tag{2.16}$$

Here a and b are the indexes of the matrix. The changes of winning the previous described XOR game for $(s,t) = (0,0)$,$(s,t) = (1,0)$ and $(s,t) = (0,1)$ are given on the diagonal and for $(s,t) = (1,1)$ on the anti diagonal. Adding these up (multiplied by the 25% chance of each question occurring) shows that the chance of winning this game using this strategy is $\approx 83\%$. This is higher then the classical maximum of 75%.

Later, in Chapter 6 we will try to find, using only the bipartite quantum correlation of CHSH, the operators and state in local dimension $d = 2$.

3

WIRTINGER CALCULUS

Before we can move on and apply numerical methods to bipartite quantum correlations, we first need some tools to tackle the problems. One of these tools is the Wirtinger calculus, named after Wilhelm Wirtinger, who introduced it in 1927. The Wirtinger derivatives can be seen as a more general approach to derivatives of complex functions. The advantage of this method is its elegance and simplicity to use, especially when trying to determine the derivative of a function with a high dimensional complex domain and a real image.

Firstly, in Section 3.1 we define the Wirtinger derivative and conjugate Wirtinger derivative. We also consider the basic behaviour for simple functions such as $f(z) = z$, $f(z) = z^*$ and $f(z) = zz^*$. Next, in Section 3.2 we discuss what happens when we apply the derivatives to functions that are holomorphic in c. This is done in a lemma that will be proved afterward. After that, in Section 3.3 the more advanced differentiation rules come to light. Using these advanced rules, we look at a few relevant examples that help us in 4. Subsequently, in Section 3.4 we work out the gradient of real-valued function with complex parameters. These are functions that often appear in optimization problems since the complex numbers $\mathbb{C}$ are not ordered. Finally, in Section 3.5 we start with a brief overview of taking the derivative with respect to a matrix, followed by defining the multi-variable CW- and W-derivative. Lastly, the chain rule for multi-variable functions is given.

Again the same as Chapter 2, this chapter is mostly a collection of existing information. The information in this chapter comes from [8], [9] and [10].

3.1. DEFINITION AND BASIC BEHAVIOUR OF THE WIRTINGER DERIVATIVES

Before giving the definition of the Wirtinger equations, we first have to forget all we know about complex derivatives. Especially the Cauchy Riemann equations that are requirements to be complex differentiable. We can start of by giving a definition of a property of complex functions, that makes sure that the functions are well-behaved.

Definition 3.1.1 (Differentiable in a real sense)**.** Let f be a complex function, then it is differentiable in the real sense if

$$\frac{\partial \mathrm{Re}(f(z))}{\partial \mathrm{Re}(z)}, \quad \frac{\partial \mathrm{Im}(f(z))}{\partial \mathrm{Im}(z)}, \quad \frac{\partial \mathrm{Re}(f(z))}{\partial \mathrm{Im}(z)}, \text{ and } \quad \frac{\partial \mathrm{Im}(f(z))}{\partial \mathrm{Re}(z)} \tag{3.1}$$

all exist everywhere.

There are 2 Wirtinger derivatives, the general Wirtinger derivative (W-derivative) and the conjugate Wirtinger derivative (CW-derivative). The first one is defined as

Definition 3.1.2 (Wirtinger Derivative)**.** Let f be differentiable in the real sense, then the general W-derivative of function f is

$$\frac{\partial f}{\partial z} = \frac{1}{2}\left(\frac{\partial f}{\partial \mathrm{Re}(z)} - i\frac{\partial f}{\partial \mathrm{Im}(z)}\right), \tag{3.2}$$

where $\mathrm{Re}(z)$ and $\mathrm{Im}(z)$ indicate respectively the Real and Imaginary parts of z.

Definition 3.1.3 (Conjugate Wirtinger Derivative)**.** Let f be differentiable in the real sense, then the CW-derivative of function f is

$$\frac{\partial f}{\partial z^*} = \frac{1}{2}\left(\frac{\partial f}{\partial \mathrm{Re}(z)} + i\frac{\partial f}{\partial \mathrm{Im}(z)}\right). \tag{3.3}$$

The only difference between the two the + or − sign. However, they behave quite differently. To get a feeling for the two derivatives behaviour, we start by trying a few examples. Using the Definition 3.2 and 3.3 we show that Wirtinger calculus is quite well behaved. This is shown in the following four examples.

Example 3. When we take the W-derivative of the function $f(z) = z$, we get

$$\frac{\partial z}{\partial z} = \frac{1}{2}\left(\frac{\partial(a+bi)}{\partial \mathrm{Re}(z)} - i\frac{\partial(a+bi)}{\partial \mathrm{Im}(z)}\right) = \frac{1}{2}\left(\frac{\partial(a+bi)}{\partial a} - i\frac{\partial(a+bi)}{\partial b}\right) = \frac{1}{2}(1-i^2) = 1, \tag{3.4}$$

which is as we would expect.

Example 4. Next, when we take the CW-derivative of $f(z) = z$ we get a more surprising result

$$\frac{\partial z}{\partial z^*} = \frac{1}{2}\left(\frac{\partial(a+bi)}{\partial \mathrm{Re}(z)} + i\frac{\partial(a+bi)}{\partial \mathrm{Im}(z)}\right) = \frac{1}{2}\left(\frac{\partial(a+bi)}{\partial a} + i\frac{\partial(a+bi)}{\partial b}\right) = \frac{1}{2}(1+i^2) = 0. \tag{3.5}$$

This might spark the feeling that the conjugate of z can be assumed to stay constant in Wirtinger calculus.

Example 5. Now it might be interesting to look at the CW-derivative of $f(z) = z^*$, which is

$$\frac{\partial z^*}{\partial z^*} = \frac{1}{2}\left(\frac{\partial(a-bi)}{\partial \mathrm{Re}(z)} + i\frac{\partial(a-bi)}{\partial \mathrm{Im}(z)}\right) = \frac{1}{2}\left(\frac{\partial(a-bi)}{\partial a} + i\frac{\partial(a-bi)}{\partial b}\right) = \frac{1}{2}(1-i^2) = 1. \tag{3.6}$$

Example 6. Lastly, mostly for completeness, the W-derivative of $f(z) = z^*$, is equal to

$$\frac{\partial z^*}{\partial z} = \frac{1}{2}\left(\frac{\partial(a-bi)}{\partial \mathrm{Re}(z)} - i\frac{\partial(a-bi)}{\partial \mathrm{Im}(z)}\right) = \frac{1}{2}\left(\frac{\partial(a-bi)}{\partial a} - i\frac{\partial(a-bi)}{\partial b}\right) = \frac{1}{2}(1+i^2) = 0. \tag{3.7}$$

These basic properties are useful when it comes to bigger derivatives and can be summarized in the following remark.

Remark. When calculating the W-derivative, z could be considered as the only changing variable, and z^* can be seen as a constant. When calculating the CW-derivative, the z^* is the only changing variable, and z should be considered as a constant. This trick only works when we write $f(z)$ in terms of z and z^*.

Using this remark, it can be straightforward to calculate the CW- and W-derivative of the following example.

Example 7. Let $f(z) = |z|^2 = z^* z$, then CW- and W-derivative of $f(z)$ are,

$$\frac{\partial z^* z}{\partial z^*} = z, \quad \frac{\partial z^* z}{\partial z} = z^*. \tag{3.8}$$

3.2. W- AND CW-DERIVATIVE FOR FUNCTIONS THAT ARE HOLOMORPHIC IN c

To see how functions that are holomorphic in c respond to the derivatives, we start by defining what it means to be holomorphic. This is done with the CR-equations.

Definition 3.2.1 (CR equations)**.** The 2 Cauchy Riemann (CR)-Equations for a complex function f in point c are:

1.
$$\frac{\partial \mathrm{Re}(f)}{\partial \mathrm{Re}(z)}(c) = \frac{\partial \mathrm{Im}(f)}{\partial \mathrm{Im}(z)}(c), \tag{3.9}$$

2.
$$\frac{\partial \mathrm{Re}(f)}{\partial \mathrm{Im}(z)}(c) = -\frac{\partial \mathrm{Im}(f)}{\partial \mathrm{Re}(z)}(c). \tag{3.10}$$

These equations are then used to define what it means to be holomorphic in a point.

Definition 3.2.2 (Holomorphic)**.** We say a function $f(z)$ is holomorphic in $c \iff$ There exists a neighborhood of c for which the two CR-equations are true for all points in the neighbourhood of c.

The functions that are holomorphic in c behave as expected around c when the W-derivative is applied to it, which is stated in Lemma 3.2.3. This makes W-derivative a generalization of complex differentiation.

Lemma 3.2.3. *If f is holomorphic in point c, then the W-derivative is equal to the complex derivative.*

Proof. We start the proof of with the expression of the complex derivative.

$$\frac{df(z)}{dz} = \lim_{\tilde{z}\to z} \frac{f(\tilde{z}) - f(z)}{\|\tilde{z} - z\|} \tag{3.11}$$

This is independent of the direction of $\tilde{z} - z$. Therefore we can chose the direction $\tilde{z}$ approaches z. In this case it useful approach z in the direction of real axis. We do this by writing $\tilde{z}(x) = \tilde{x} + i\text{Im}(z)$.

$$\frac{df(z)}{dz} = \lim_{\tilde{x}\to \text{Im}(z)} \left(\frac{f(\tilde{x} + i\text{Re}(z)) - f(z)}{\|\tilde{x} - \text{Re}(z)\|} \right) = \frac{\partial f(z)}{\text{Re}(z)} \tag{3.12}$$

Now, we almost have our desired result. Last step is to show that $\frac{\partial f(z)}{\partial \text{Re}(z)} = -i\frac{\partial f(z)}{\partial \text{Im}(z)}$, which can be done with the CR-equations,

$$\begin{aligned} \frac{\partial f(z)}{\partial \text{Re}(z)} &= \frac{\partial \text{Re}(f(z))}{\partial \text{Re}(z)} + i\frac{\partial \text{Im}(f(z))}{\partial \text{Re}(z)} \\ &= \frac{\partial \text{Im}(f(z))}{\partial \text{Im}(z)} - i\frac{\partial \text{Re}(f(z))}{\partial \text{Im}(z)} \\ &= -i\frac{\partial f(z)}{\partial \text{Im}(z)}. \end{aligned} \tag{3.13}$$

Now we can combine the two previous results,

$$\frac{df(z)}{dz} = \frac{\partial f(z)}{\partial \text{Re}(z)} = \tfrac{1}{2}\frac{\partial f(z)}{\partial \text{Re}(z)} + \tfrac{1}{2}\frac{\partial f(z)}{\partial \text{Re}(z)} = \tfrac{1}{2}\left(\frac{\partial f(z)}{\partial \text{Re}(z)} - i\frac{\partial f(z)}{\partial \text{Im}(z)} \right) = \frac{\partial f(z)}{\partial z} \tag{3.14}$$

which concludes the proof. □

Next we see what happens when we take the CW-derivative of a function that is holomorphic in c.

Lemma 3.2.4. *If f is holomorphic in c, then the CW-derivative is equal to 0.*

Proof. For this we proof we apply the CR-equations to show that $\frac{\partial f(z)}{\partial \text{Re}(z)} = -i\frac{\partial f(z)}{\partial \text{Im}(z)}$. This is done in the same way as done in the Proof of Lemma 3.2.3. Now we can apply this fact to the definition of the CW-derivative,

$$\frac{\partial f(z)}{\partial z^*} = \tfrac{1}{2}\left(\frac{\partial f(z)}{\partial \text{Re}(z)} + i\frac{\partial f(z)}{\partial \text{Im}(z)} \right) = \tfrac{1}{2}\left(\frac{\partial f(z)}{\partial \text{Re}(z)} - \frac{\partial f(z)}{\partial \text{Re}(z)} \right) = 0. \tag{3.15}$$

This concludes the proof. □

3.3. PROPERTIES OF WIRTINGER DERIVATIVES

In this section, we only mention the properties and not proof them due to simplicity and a large amount of bookkeeping of the proofs. For the proofs, the reader is encouraged to take a look at [8].

Lemma 3.3.1. *If f is differentiable in the real sense at c, then*

$$\left(\frac{\partial f}{\partial z}(c)\right)^* = \frac{\partial f^*}{\partial z^*}(c). \tag{3.16}$$

Lemma 3.3.2. *If f is differentiable in the real sense at c, then*

$$\left(\frac{\partial f}{\partial z^*}(c)\right)^* = \frac{\partial f^*}{\partial z}(c). \tag{3.17}$$

Lemma 3.3.3. *If f and g are differentiable in the real sense at c and $\alpha, \beta \in \mathbb{C}$, then*

$$\frac{\partial(\alpha f + \beta g)}{\partial z}(c) = \alpha \frac{\partial f}{\partial z}(c) + \beta \frac{\partial g}{\partial z}(c), \tag{3.18}$$

$$\frac{\partial(\alpha f + \beta g)}{\partial z^*}(c) = \alpha \frac{\partial f}{\partial z^*}(c) + \beta \frac{\partial g}{\partial z^*}(c). \tag{3.19}$$

Lemma 3.3.4. *If f and g are differentiable in the real sense at c, then*

$$\frac{\partial(f \cdot g)}{\partial z}(c) = \frac{\partial f}{\partial z}(c) \cdot g(c) + f(c) \cdot \frac{\partial g}{\partial z}(c). \tag{3.20}$$

Lemma 3.3.5. *If f and g are differentiable in the real sense at c, then*

$$\begin{aligned} \frac{\partial g(f(z))}{\partial z} &= \frac{\partial g}{\partial u}\frac{\partial u}{\partial z} + \frac{\partial g}{\partial u^*}\frac{\partial u^*}{\partial z} \\ &= \frac{\partial g}{\partial u}\frac{\partial u}{\partial z} + \left(\frac{\partial g^*}{\partial u}\right)^* \frac{\partial u^*}{\partial z}. \end{aligned} \tag{3.21}$$

Notice that the second term in Lemma 3.3.5 becomes zero if f(x) is holomorphic in c, leaving us with the normal chain rule. We can try a few examples.

Example 8. Consider $f_1(z) = z^2 z^*$, a function with z as well as z^* in it. To calculate this, we can use the trick previously mentioned in Section 3.1 namely that we can see z and z^* as independent variables. This way calculating the W-derivative and the CW-derivative becomes quite easy. For the W-derivative, we can see the function as $z^2 a$, where a is a constant. The W-derivative then becomes $\frac{\partial f(z)}{\partial z} = 2zz^*$. It works the same for the CW-derivative. In that case, the function can be seen as bz^*, where b is a constant. The CW-derivative is equal to $\frac{\partial f(z)}{\partial z^*} = z^2$.

Example 9. Next we can look at a larger example where the linearity can be used. Consider a function $f(z) = 5z^5(z^*)^3 + z^5$, then by the linearity the W-derivative is easily calculated to be $\frac{\partial f(z)}{\partial z} = 5\frac{\partial(z^5(z^*)^3)}{\partial z} + \frac{\partial(z^5)}{\partial z} = (25(z^*)^3 + 5)z^4$. The same goes for the CW-derivative which in this case is $\frac{\partial f(z)}{\partial z^*} = 5\frac{\partial(z^5(z^*)^3)}{\partial z^*} + \frac{\partial(z^5)}{\partial z^*} = 15z^5(z^*)^2$.

Example 10. For the last one let us look at $f_3(z) = (\|z\|_2^2 - 1)^2$. There are two ways to determine the derivatives. The first one is with the linearity, which we can use when we fully write out the function $f_3(z) = (zz^* - 1)(zz^* - 1) = z^2(z^*)^2 - 2zz^* + 1$. Then the W-derivative is $\frac{\partial f_3(z)}{\partial z} = 2z(z^*)^2 - 2z^*$. Now for the CW-derivative we use the chain rule, then the CW-derivative is equal to $\frac{\partial f_3(z)}{\partial z^*} = 2(zz^* - 1)\frac{\partial(zz^* - 1)}{\partial z^*} + 0 = 2(zz^* - 1)z$. Here the second part of the chain rule is zero as z^2 is holomorphic.

3.4. GRADIENT OF REAL-VALUED FUNCTIONS

We start this section with a definition for a gradient and then show that it points towards the direction of the steepest ascent for real-valued functions with complex variables.

Definition 3.4.1 (Wirtinger gradient)**.** Let f be a real-valued function with complex variables, then the gradient in point c is

$$\nabla f(c) = 2\frac{\partial f}{\partial z^*}(c) = \frac{\partial f}{\partial \mathrm{Re}(z)}(c) + i\frac{\partial f}{\partial \mathrm{Im}(z)}(c). \tag{3.22}$$

One of the most important theorems that we use is Theorem 3.4.2. We show a proof from [8], with some small changes.

Theorem 3.4.2. *Let f be a real-valued function with complex variables, that is differentiable in the real sense, then the gradient of real-valued functions with complex variables in c points in the direction of $\frac{\partial f}{\partial z^*}(c)$*

Proof. This proof is divided into three parts. The first part gives us the first-order Taylor expansion of a complex function f. The second part simplifies that expansion for real-valued functions with complex variables, and the last part shows us that $\frac{\partial f}{\partial z^*}(c)$ points in the direction of steepest ascent in c.

1. Let us start with writing $f(z)$ as $f(x+iy) = u(x,y) + iv(x,y)$, where x and u correspond to the real parts and y and v to the imaginary parts. Now we can take the first-order Taylor expansion of both u and v in $c = c_1 + ic_2$, with $h = h_1 + ih_2$

$$u(c+h) = u(c) + h_1\frac{\partial u}{\partial x}(c) + h_2\frac{\partial u}{\partial y}(c) + \mathcal{O}(h^2) \tag{3.23}$$

$$v(c+h) = v(c) + h_1\frac{\partial v}{\partial x}(c) + h_2\frac{\partial v}{\partial y}(c) + \mathcal{O}(h^2) \tag{3.24}$$

where $\mathcal{O}$ denotes the big-O, a way to write the remainder. Combining the two in a function for f gives

$$f(c+h) = f(c) + \left(\frac{\partial u}{\partial x}(c) + i\frac{\partial v}{\partial x}(c)\right)h_1 + \left(\frac{\partial u}{\partial y}(c) + i\frac{\partial y}{\partial x}(c)\right)h_2 + \mathcal{O}(h^2) \tag{3.25}$$

Subsequently, we substitute $h_1 = \frac{h+h^*}{2}$ and $h_2 = \frac{h-h^*}{2i}$ and write $\frac{\partial u}{\partial x}(c) + i\frac{\partial v}{\partial x}(c) = \frac{\partial f}{\partial x}(c)$ and $\frac{\partial u}{\partial y}(c) + i\frac{\partial v}{\partial y}(c) = \frac{\partial f}{\partial y}(c)$

$$\begin{aligned} f(c+h) &= f(c) + \frac{1}{2}\left(\frac{\partial f}{\partial x}(c) + \frac{1}{i}\frac{\partial f}{\partial y}(c)\right)h + \frac{1}{2}\left(\frac{\partial f}{\partial x}(c) - \frac{1}{i}\frac{\partial f}{\partial f}(c)\right)h^* + \mathcal{O}(h^2) \\ &= f(c) + \frac{1}{2}\left(\frac{\partial f}{\partial x}(c) - i\frac{\partial f}{\partial y}(c)\right)h + \frac{1}{2}\left(\frac{\partial f}{\partial x}(c) + i\frac{\partial f}{\partial f}(c)\right)h^* + \mathcal{O}(h^2) \\ &= f(c) + \left(\frac{\partial f}{\partial z}, \frac{\partial f}{\partial z^*}\right)\cdot\begin{pmatrix} h \\ h^* \end{pmatrix} + \mathcal{O}(h^2) \end{aligned} \tag{3.26}$$

which concludes our first step.

2. The next step is to use the fact that the function is real-valued and applying Lemma 3.3.1,which gives us $\frac{\partial f}{\partial z^*}(c) = \frac{\partial f^*}{\partial z^*}(c) = \frac{\partial f}{\partial z}(c)$. This can then be filled in to 3.26.

$$\begin{aligned} f(c+h) &= f(c) + \frac{\partial f}{\partial z}h + \frac{\partial f}{\partial z^*}h^* + \mathcal{O}(h^2) \\ &= f(c) + \frac{\partial f}{\partial z}h + \left(\frac{\partial f}{\partial z}h\right)^* + \mathcal{O}(h^2) \\ &= f(c) + 2Re\left(\frac{\partial f}{\partial z}h\right) + \mathcal{O}(h^2) \end{aligned} \tag{3.27}$$

   which already concludes our second step.

3. In the last step, we use the result found in step 2 and come to our final result. We start by fixing the absolute value of h to ϵ with $\epsilon \in \mathbb{R}$, significantly small, such that only first-order phenomena are working. Then we look at what direction would minimize $f(c+h)$. Since we have done that for an arbitrary value of ϵ, it works for all ϵ small enough.

$$\begin{aligned} f(c+h) &= f(c) + 2Re\left(\frac{\partial f}{\partial z}h\right) + \mathcal{O}(h^2) \\ &= f(c) + 2Re\left(\left(\frac{\partial f}{\partial z^*}\right)^* h\right) + \mathcal{O}(h^2) \end{aligned} \tag{3.28}$$

   Since h has a fixed modulus and only the real part is taken of $\left(\left(\frac{\partial f}{\partial z^*}\right)^* h\right)$, $f(c+h)$ takes on the biggest value if h is parallel to $\frac{\partial f}{\partial z^*}$. Therefore, we can conclude that $\frac{\partial f}{\partial z^*}$ points in the direction of the gradient.

□

3.5. MATRIX WIRTINGER CALCULUS

To understand how Matrix Wirtinger Calculus works, it is useful to first repeat how normal matrix differentials work. If we have an function $f(X)$, where X is a $n \times n$ matrix and take the derivative with respect to X we get:

$$\frac{\partial f(X)}{\partial X} = \begin{pmatrix} \frac{\partial f(X)}{\partial X_{11}} & \frac{\partial f(X)}{\partial X_{12}} & \cdots & \frac{\partial f(X)}{\partial X_{1(n-1)}} & \frac{\partial f(X)}{\partial X_{1n}} \\ \frac{\partial f(X)}{\partial X_{21}} & \frac{\partial f(X)}{\partial X_{22}} & \cdots & \frac{\partial f(X)}{\partial X_{2(n-1)}} & \frac{\partial f(X)}{\partial X_{2n}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial f(X)}{\partial X_{(n-1)1}} & \frac{\partial f(X)}{\partial X_{(n-1)2}} & \cdots & \frac{\partial f(X)}{\partial X_{(n-1)(n-1)}} & \frac{\partial f(X)}{\partial X_{(n-1)n}} \\ \frac{\partial f(X)}{\partial X_{n1}} & \frac{\partial f(X)}{\partial X_{n2}} & \cdots & \frac{\partial f(X)}{\partial X_{n(n-1)}} & \frac{\partial f(X)}{\partial X_{nn}} \end{pmatrix}. \tag{3.29}$$

Here X_{kl} is the element of matrix X on the k^{th} row and l^{th} column.

Before we can apply the W-derivative to multi-variable functions, we should first define it. This is done in Definition 3.5.1

Definition 3.5.1 (Wirtinger derivatives)**.** Let f be a function on $\mathbb{C}^n$, then the Wirtinger derivatives (also known as W-derivatives) are operators defined as

$$\begin{aligned}
\frac{\partial}{\partial z_1} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_1)} - i\frac{\partial}{\partial \mathrm{Im}(z_1)}\right),\\
\frac{\partial}{\partial z_2} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_2)} - i\frac{\partial}{\partial \mathrm{Im}(z_2)}\right),\\
&\vdots\\
\frac{\partial}{\partial z_n} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_n)} - i\frac{\partial}{\partial \mathrm{Im}(z_n)}\right).
\end{aligned} \tag{3.30}$$

The same should of course be done for CW-derivative in Definition 3.5.2.

Definition 3.5.2 (Conjugate Wirtinger derivative)**.** Let f be a function on $\mathbb{C}^n$, then the conjugate Wirtinger derivatives (also known as CW-derivatives) are operators defined as

$$\begin{aligned}
\frac{\partial}{\partial z_1^*} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_1)} + i\frac{\partial}{\partial \mathrm{Im}(z_1)}\right),\\
\frac{\partial}{\partial z_2^*} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_2)} + i\frac{\partial}{\partial \mathrm{Im}(z_2)}\right),\\
&\vdots\\
\frac{\partial}{\partial z_n^*} &= \frac{1}{2}\left(\frac{\partial}{\partial \mathrm{Re}(z_n)} + i\frac{\partial}{\partial \mathrm{Im}(z_n)}\right).
\end{aligned} \tag{3.31}$$

Luckily the Lemmas 3.3.1, 3.3.2, 3.3.3, 3.3.4 that were true for 1 complex variable derivatives are still true for multiple variables. The only one that has changed a bit is the chain rule. This is because the chain rule has to take into account all different variables and sum over them. The "new" chain rule becomes:

Lemma 3.5.3. *If, for $m,n,l \in \mathbb{N}$ $f(z_1,z_2,\ldots,z_n)$: $\mathbb{C}^m -> \mathbb{C}^n$ and $g(z_1,z_2,\ldots,z_m)$:$\mathbb{C}^n -> \mathbb{C}^l$ are differentiable in the real sense at c, then the W-derivative is*

$$\begin{aligned}
\frac{\partial g(f(z))}{\partial z_a} &= \sum_{i=1}^{n}\left(\frac{\partial g(f(z))}{\partial f(z_i)}\frac{\partial f(z_i)}{\partial z_a} + \frac{\partial g(f(z))}{\partial f(z_i)^*}\frac{\partial f(z_i)^*}{\partial z_a}\right)\\
&= \sum_{i=1}^{n}\left(\frac{\partial g(f(z))}{\partial f(z_i)}\frac{\partial f(z_i)}{\partial z_a} + \left(\frac{\partial g(f(z))^*}{\partial f(z_i)}\right)^*\frac{\partial f(z_i)^*}{\partial z_a}\right).
\end{aligned} \tag{3.32}$$

The CW-derivative is similarly,

$$\begin{aligned}
\frac{\partial g(f(z))}{\partial z_a^*} &= \sum_{i=1}^{n}\left(\frac{\partial g(f(z))}{\partial f(z_i)}\frac{\partial f(z_i)}{\partial z_a^*} + \frac{\partial g(f(z))}{\partial f(z_i)^*}\frac{\partial f(z_i)^*}{\partial z_a^*}\right)\\
&= \sum_{i=1}^{n}\left(\frac{\partial g(f(z))}{\partial f(z_i)}\frac{\partial f(z_i)}{\partial z_a^*} + \left(\frac{\partial g(f(z))^*}{\partial f(z_i)}\right)^*\frac{\partial f(z_i)^*}{\partial z_a^*}\right).
\end{aligned} \tag{3.33}$$

When we want to determine the gradient of a function with a high dimensional domain (as in 4.20), it can be neat to determine the CW-derivative of each element of a matrix separately [9]. In this case the expression 3.34 can be helpful. Here the property of the trace $\mathrm{tr}(AB^T) = \sum_{i,j} A_{ij}B_{ij}$ are used, where A_{ij} are the entries of matrix A.

$$\begin{aligned}\frac{\partial g(F(X)}{\partial X^*_{kl}} &= \left(\sum_{ij} \frac{\partial g(F(X)}{\partial (F(X)_{ij}} \frac{\partial (F(X))_{ij}}{\partial X^*_{kl}}\right) + \left(\sum_{ij} \frac{\partial g(F(X)}{\partial (F(X)^*_{ij}} \frac{\partial (F(X))^*_{ij}}{\partial X^*_{kl}}\right) \\ &= \mathrm{Tr}\left(\frac{\partial g(F(X)}{\partial (F(X)} \left(\frac{\partial F(X)}{\partial X^*_{kl}}\right)^T\right) + \mathrm{Tr}\left(\frac{\partial g(F(X)}{\partial (F(X)^*} \left(\frac{\partial F(X)^*}{\partial X^*_{kl}}\right)^T\right) \\ &= \mathrm{Tr}\left(\frac{\partial g(F(X)}{\partial (F(X)} \left(\frac{\partial F(X)}{\partial X^*_{kl}}\right)^T\right) + \mathrm{Tr}\left(\frac{\partial g(F(X)}{\partial (F(X)^*} \left(\frac{\partial F(X)}{\partial X_{kl}}\right)^H\right)\end{aligned} \tag{3.34}$$

Here F and X are matrices. ^{H} denotes the Hermitian of a matrix and * denotes only the conjugated of a matrix. For vectors * always means the conjugated transposed. In the last line the identity $(A^T)^* = A^H$ is used.

4

PROBLEM DESCRIPTION

As described before, the bipartite quantum correlation P with local dimension d can be described in the form of Equation 4.2

$$P(a,b|s,t) = \mathrm{Tr}((E_s^a \otimes F_t^b)\psi\psi^*) = \psi^*(E_s^a \otimes F_t^b)\psi \tag{4.1}$$

where a,b are elements of the answer set, s,t are element of question set, ψ is a unit vector in $\mathbb{C}^{d^2}$, and E_s^a and F_t^b are Hermitian positive semidefinite matrices in $\mathbb{C}^{d\times d}$. The $P(a,b|s,t)$, depending on a,b,s,t, can be determined experimentally. However, this does not give the explicit ψ,E_s^a and F_t^b the right hand side of 4.1. This is were the optimization comes into place.

Firstly, in Section 4.1 we define our problem with a cost function and constraints. Next, in Section 4.2 we do 3 things. We start by changing the constraint of $\|\psi\| = 1$ into a penalty function. The same is then done for the constraint that the POVMs have to sum to the identity matrix. Lastly, a method gets introduced that requires the POVM to be Hermitian and positive semidefinite. After that, in Section 4.3 we give a new unconstrained problem description, with the penalty functions. Subsequently, in Section 4.4 we calculate, using the Wirtinger calculus of Chapter 3, the gradient of the unconstrained problem. Lastly, in Section 4.5 we look at two additional penalty functions to make sure that the POVMs are also protective matrices, which make them PVMs.

4.1. PROBLEM DESCRIPTION, WITH CONSTRAINS

Given a bipartite quantum correlation $P(a,b|s,t)$ and dimension d, find ψ,$\{E_s^a\}$ and $\{F_t^b\}$ such that

$$f(\psi,\{E_s^a\},\{F_t^b\}) = \sum_{a,b,s,t} \left(P(a,b|s,t) - \psi^*(E_s^a \otimes F_t^b)\psi\right)^2 \tag{4.2}$$

is minimal. There are several norms that can be used to determine the distance of $(P(a,b|s,t) - \psi^*(E_s^a \otimes F_t^b)\psi$ but for now we use the L^2-norm squared. Important to note is that since we are minimizing towards zero we are allowed to square the norm, which makes it easier to calculate the gradient. Subject to:

$$\psi^*\psi = 1 \tag{4.3}$$

$$\sum_a E_s^a = \sum_b F_t^b = I \quad \forall s,t \in S \times T \tag{4.4}$$

$$(E_s^a)_{ij} = (\bar{E_s^a})_{ji} \quad \forall i,j \in d \times d \quad \forall a,s \in A \times S \tag{4.5}$$

$$x^*(E_s^a)x \geq 0 \quad \forall x \in \mathbb{C}^d \quad \forall a,s \in A \times S \tag{4.6}$$

$$(F_t^b)_{ij} = (\bar{F_s^a})_{ji} \quad \forall i,j \in d \times d \quad \forall b,t \in B \times T \tag{4.7}$$

$$x^*(F_t^b)x \geq 0 \quad \forall x \in \mathbb{C}^d \quad \forall b,t \in B \times T \tag{4.8}$$

Here we can write 4.3 as $\sum_i \bar{\psi}_i \psi_i = 1$, which requires psi to be a unit vector. Furthermore, since $\{E_s^a\}$ and $\{F_t^b\}$ are POVMs then by definition 2.3.4 they are required to obey 4.4, 4.5, 4.6, 4.7 and 4.8.

4.2. PENALTY FUNCTIONS

Gradient descent is only possible when there are no constraints, which of course, is not the case for our current problem. Therefore we modify the problem slightly by expressing the constraints as penalty functions. For equality constraints of the form $Ax = b$ it can be written as $\mu \| Ax - b \|_2$.

Note that the 'type' of the norm and μ can be changed. For the most simplistic case, we assume $\mu = 1$ and not variable and that the norm is the L_2^2 norm. In the case for 4.3 the penalty function becomes,

$$(\psi^*\psi - 1)^2. \tag{4.9}$$

Here $\psi^*\psi \in \mathbb{C}$, so we only have to sum over one square. Next for 4.4 we require the individual elements of the matrices to sum to, one if it is a diagonal element, and to zero if it is a non-diagonal element. For E we have

$$\begin{aligned} f_{penE} = &\sum_s \sum_i \left\| \left(\sum_a (E_s^a)_{ii} \right) - 1 \right\|_2^2 \\ &+ \sum_s \sum_{\substack{ij \\ i \neq j}} \left\| \left(\sum_a (E_s^a)_{ij} \right) \right\|_2^2 \end{aligned} \tag{4.10}$$

and similarly for F

$$\begin{aligned} f_{penF} = &\sum_t \sum_i \left\| \left(\sum_b (F_t^b)_{ii} \right) - 1 \right\|_2^2 \\ &+ \sum_t \sum_{\substack{ij \\ i \neq j}} \left\| \left(\sum_b (F_t^b)_{ij} \right) \right\|_2^2. \end{aligned} \tag{4.11}$$

Although Equation 4.10 and Equation 4.11 do transform two of the constraints into penalty functions, it does not get a lot easier. Therefore we introduce the Frobenius norm.

Definition 4.2.1 (Frobenius norm)**.** The Frobenius norm of a $n \times m$ matrix $M \in \mathbb{C}^{n\times m}$ is

$$\|M\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |M_{ij}|^2} = \sqrt{\mathrm{tr}(A^H A)}.$$

Checking if the Frobenius norm is actually a norm can be considered as an exercise for the reader. The Frobenius norm suits the penalty functions perfectly. Therefore 4.10 can be written as

$$f_{penE} = \sum_s \left\| \left(\sum_a (E_s^a) \right) - I \right\|_F^2 \tag{4.12}$$

and 4.11 as

$$f_{penF} = \sum_t \left\| \left(\sum_b (F_t^b) \right) - I \right\|_F^2. \tag{4.13}$$

The norm can written as he trace form, which make is east to determine the gradient. This is done in Section 4.4.

For the Hermitian constraints, we could add another penalty function, which would add an error for elements that do not obey 4.5 and 4.7. However, this has the downside that it makes the penalty function larger. For the positive semidefinite constrains Equation 4.6 and Equation 4.8, we have a larger problem since it is a more general statement and it is hard to create penalty functions for these expressions. It could be done with $(\mathrm{Tr}E)^2/\mathrm{Tr}(E^2) > d-1$, where d is the local dimension [11](equation 2.39). However, this would make the overall penalty function more complicated.

Therefore we should look at an alternative, namely the Cholesky Decomposition of E and F, which could kill two birds with one stone.

Definition 4.2.2 (Cholesky Decomposition)**.** Let A be a $n \times n$ matrix. Then the Cholesky decomposition of A is of the form

$$A = L^H L,$$

where L is a lower triangle matrix.

We repeat that in this thesis we use ^{H} to denote the Hermitian conjugate of a matrix. The usefulness of Cholesky decomposition can be best shown in the next theorem.

Theorem 4.2.3 (Cholesky decomposition are Hermitian and positive semidefinite)**.** *Let A be a $n \times n$ matrix. Then the following holds:*
A is Hermitian and positive semidefinite $\iff$ There exists a Cholesky decomposition of A.

To prove Theorem 4.2.3 we first have to prove Lemma 4.2.4.

Lemma 4.2.4. *Let A be a positive semidefinite $n \times n$ matrix, then there is a $n \times n$ a Hermitian matrix B such that $A = BB$*

Proof. We know that A is Hermitian since A is positive semidefinite. Using diagonalization, we know it is possible to write the Hermitian matrix A as $A = SDS^H$ where D is a diagonal matrix with entries of the eigenvalues of A and S as an orthogonal matrix.

Next, we define D_s as a diagonal matrix with on the diagonal the square roots of all eigenvalues. We can do this since all eigenvalues $\lambda_i \in \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ are greater or equal then 0, because A is positive semidefinite. Furthermore, we define S as before. Now we have a SD_sS^H which is the square root of A, namely

$$BB = SD_sS^HSD_sS^H = SD_sD_sS^H = SDS^H = A, \tag{4.14}$$

which concludes the proof. □

Following this we give a theorem about QR decomposition. Proving this can be done by reader, by writing matrix product of QL out.

Lemma 4.2.5. *Let B be a $n \times n$ matrix, then it is possible to write B as,*

$$B = QL, \tag{4.15}$$

with L a lower triangular matrix and Q an unitary matrix.

Now we go back to Theorem 4.2.3 and use the previous 2 lemmas to get the desired result.

Proof of Theorem 4.2.3. ($\Longrightarrow$)We first start with the forward implication. Let A be a positive semidefinite $n \times n$ matrix, then by Lemma 4.2.4 there is $n \times n$ Hermitian matrix B such that $B^2 = A$. Furthermore, by Lemma 4.2.5 all matrices have a QL decomposition. Combining these give us

$$A = BB^H = (QL)^H(QL) = L^HQ^HQL = L^HL,$$

which completes the first part of the proof.
($\Longleftarrow$) Next we do the backward implication, which is a bit more straight forward. Let us start with the assumption that $A = L^HL$, then Hermitian part is done directly,

$$A^H = (L^HL)^H = L^HL = A. \tag{4.16}$$

That A is positive semidefinite is proven by,

$$x^*Ax = x^*(L^HL)x = (Lx)^*(Lx) \geq 0. \tag{4.17}$$

Here the inequality comes from the fact that $(Lx)^*Lx$ is the modulus squared of Lx. Now we have shown that when matrix A has a Cholesky decomposition, it is Hermitian and positive semidefinite. This completes our proof. □

Using the previous proven result of Lemma 4.2.3 and the function $G(X) = X^HX$, we can write

$$E_s^a(X_s^a) = G(X_s^a) = (X_s^a)^H(X_s^a) \quad \forall (a,s) \in A \times S, \tag{4.18}$$

where E_s^a has to be Hermitian and positive semidefinite. We can do the same for F,

$$F_t^b(Y_t^b) = G(Y_t^b) = (Y_t^b)^H(Y_t^b) \quad \forall (b,t) \in B \times T, \tag{4.19}$$

then again we know that F_t^b is Hermitian and positive.

4.3. PROBLEM DESCRIPTION, WITH PENALTIES

So we can use the Cholesky decomposition to make sure that Equations 4.5,4.10,4.7 and4.11 are always true and we use the penalty function to replace constraints Equation 4.3 and Equation 4.4. The new problem is now expressed in terms of X and Y instead of E and F description, which is equal to

$$\begin{aligned} f(\psi,\{X_s^a\},\{Y_t^b\}) = &\sum_{a,b,s,t}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right)^2 \\ &+\left(\psi^*\psi-1\right)^2+\sum_s\left\|\left(\sum_a G(X_s^a)\right)-I\right\|_F^2+\sum_t\left\|\left(\sum_b G(Y_t^b)\right)-I\right\|_F^2. \end{aligned} \tag{4.20}$$

Here the first rule is called the f_{obj} and the second is called f_{pen}. In the next section, we tackle both of them separately. We can do this due to the linearity of the Wirtinger derivatives.

4.4. GRADIENT

The next step is to determine the gradient of f_{pen} and f_{obj}, this is done separately in the following to subsections. The total gradient has been numerically tested in Section 6.1.

4.4.1. GRADIENT OF f_{obj} WITH RESPECT TO ψ, X AND Y

The function of f_{obj} is

$$f_{obj}(\psi,\{X_s^a\},\{Y_t^b\}) = \sum_{a,b,s,t}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right)^2. \tag{4.21}$$

First let's take the derivative w.r.t. ψ. This is quite easily done with the CW-derivative and the product rule. This leave us with

$$\begin{aligned} \nabla_\psi f_{obj}(\psi,\{X_s^a\},\{Y_t^b\}) &= 2\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial\psi^*} \\ &= 2\frac{\partial}{\partial\psi^*}\sum_{a,b,s,t}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right)^2 \\ &= 4\sum_{a,b,s,t}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right) \\ &\quad\cdot\frac{\partial}{\partial\psi^*}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right) \\ &= -4\sum_{a,b,s,t}\left(P(a,b|s,t)-\psi^*(G(X_s^a)\otimes G(Y_t^b))\psi\right)\cdot\left(G(X_s^a)\otimes G(Y_t^b))\psi\right). \end{aligned} \tag{4.22}$$

The next step is to take the derivative with respect to X_s^a. This is done in two steps, the first step considers a version of Equation 4.21, where we take the derivative w.r.t. $E_{s_1}^{a_1}$ and $(E_{s_1}^{a_1})^*$, with s_1 and a_1 fixed. This is done because we need these for the chain rule. Then we determine the both W-derivatives of $G(X)=X^HX$. To eventually apply Equation 3.34

to Equation 4.21. So starting with a single element of the matrix $(E_{s_1}^{a_1})_{ij}$. The single element derivatives use the fact that our objective is the sum of a lot of linear equations, the chain rule, and the fact that $\frac{\partial E}{\partial E_{ij}} = J^{ij}$, where J^{ij} is a single element matrix. This gives us

$$\begin{aligned}
\frac{\partial f_{obj}(\psi, \{E_s^a\}, \{F_t^b\})}{\partial (E_{s_1}^{a_1})_{ij}} &= \frac{\partial}{\partial (E_s^a)_{ij}} \sum_{a,b,s,t} \left(P(a,b|s,t) - \psi^*(E_s^a \otimes F_t^b)\psi\right)^2 \\
&= \frac{\partial}{\partial (E_s^a)_{ij}} \sum_{b,t} \left(P(a_1,b|s_1,t) - \psi^*(E_{s_1}^{a_1} \otimes F_t^b)\psi\right)^2 \\
&= -2 \sum_{b,t} \left(P(a_1,b|s_1,t) - \psi^*(E_{s_1}^{a_1} \otimes F_t^b)\psi\right) \cdot \frac{\partial}{\partial (E_s^a)_{ij}} \psi^*(E_{s_1}^{a_1} \otimes F_t^b)\psi \\
&= -2 \sum_{b,t} \left(P(a_1,b|s_1,t) - \psi^*(E_{s_1}^{a_1} \otimes F_t^b)\psi\right) \cdot \psi^*(J^{ij} \otimes F_t^b)\psi.
\end{aligned} \tag{4.23}$$

Next we note that the CW-derivatives is equal to zero, for all elements of $E_{s_1}^{a_1}$. This is because $f_{obj}(\psi, \{E_s^a\}, \{F_t^b\})$ is everywhere holomorphic, when we consider the only $\{E_s^a\}$ as a variable. This also means that

$$\frac{\partial}{\partial (E_s^a)_{ij}^*} f_{obj}(\psi, \{E_s^a\}, \{F_t^b\}) = 0. \tag{4.24}$$

Next up, we calculate $\frac{\partial (X^H X)}{\partial (X_{kl}^*)}$ and $\frac{\partial (X^H X)}{\partial (X_{kl})}$ [9], these are equal to

$$\frac{\partial G(X)}{\partial (X_{kl}^*)} = J^{lk} X, \tag{4.25}$$

$$\frac{\partial G(X)}{\partial (X_{kl})} = X^H J^{kl}. \tag{4.26}$$

Notice that the single entry matrix J^{lk} is flipped in the case of Equation 4.25, this is because X^H is Hermitian transposed.

Now the fun starts, we can start combining. We start with Equation 3.22 and apply Equation 3.34. The combination results in

$$\begin{aligned}
2\frac{\partial f_{obj}(\psi, \{X_s^a\}, \{Y_t^b\})}{\partial (X_{s_1}^{a_1})_{kl}^*} &= 2\left(\sum_{i,j,a,s} \frac{\partial f_{obj}(\psi, \{X_s^a\}, \{Y_t^b\})}{\partial G(X_s^a)_{ij}} \frac{\partial (E_s^a)_{ij}}{\partial (X_{s_1}^{a_1})_{kl}^*} \right) \\
&\quad + 2\left(\sum_{i,j,a,s} \frac{\partial f_{obj}(\psi, \{X_s^a\}, \{Y_t^b\})}{\partial G(X_s^a)_{ij}^*} \frac{\partial \left(X^H X\right)_{ij}^*}{\partial (X_{s_1}^{a_1})_{kl}^*} \right) \\
&= 2 \sum_{a,s} \mathrm{Tr}\left(\left(\frac{\partial f_{obj}(\psi, \{X_s^a\}, \{Y_t^b\})}{\partial G(X_s^a)} \right)^T \frac{\partial ((X_s^a)^H X_s^a)}{\partial (X_{s_1}^{a_1})_{kl}^*} \right) + 0.
\end{aligned} \tag{4.27}$$

This can be used as an expression for the gradient. However, since we will use this in computation, we should try to make it more neatly. This will be done by filling in $\frac{\partial (X^H X)}{\partial X_{kl}^*}$

and using two characteristics of the trace namely

$$Tr(ABC) = Tr(BCA) \tag{4.28}$$

and

$$Tr(J^{ij}A) = A_{ji}. \tag{4.29}$$

Working the equation further out gives us

$$\begin{aligned}
2\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial (X_{s_1}^{a_1})^*_{kl}} &= 2\sum_{a,s}\mathrm{Tr}\left(\left(\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_s^a)}\right)^T \frac{\partial ((X_s^a)^H X_s^a)}{\partial (X_{s_1}^{a_1})^*_{kl}}\right) \\
&= 2\,\mathrm{Tr}\left(\left(\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_{s_1}^{a_1})}\right)^T J^{lk} X_{s_1}^{a_1}\right) \\
&= 2\,\mathrm{Tr}\left(J^{lk} X_{s_1}^{a_1}\left(\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_{s_1}^{a_1})}\right)^T\right) \\
&= 2\left(X_{s_1}^{a_1}\left(\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_{s_1}^{a_1})}\right)^T\right)_{kl}.
\end{aligned} \tag{4.30}$$

Next, we write out the multiplication and fill in the function, leaving us with

$$\begin{aligned}
2\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial (X_{s_1}^{a_1})^*_{kl}} &= 2\sum_i\left((X_{s_1}^{a_1})_{ki}\left(\left(\frac{f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_{s_1}^{a_1})}\right)_{il}\right)^T\right) \\
&= 2\sum_i\left((X_{s_1}^{a_1})_{ki}\left(\frac{f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial G(X_{s_1}^{a_1})}\right)_{li}\right) \\
&= -4\sum_{i,b,t}\left((X_{s_1}^{a_1})_{ki}\big(P(a,b|s,t) - \psi^*(E_{s_1}^{a_1}\otimes F_t^b)\psi\big)\big(\psi^*(J^{li}\otimes F_t^b)\psi\big)\right).
\end{aligned} \tag{4.31}$$

This leaves us with quite a nice expression, which can be used for computation. However, we can make it even faster by rewriting $\psi^*(E\otimes J^{li})\psi$, as the sum over the non zero elements. This is used in the code in Appendix A.

The same steps can be taken for Y, this gives us

$$2\frac{\partial f_{obj}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial (Y_{t_1}^{b_1})^*_{kl}} = -4\sum_{i,a,s}\left((Y_{t_1}^{b_1})_{ki}\big(P(a,b|s,t) - \psi^*(E_s^a\otimes F_{t_1}^{b_1})\psi\big)\big(\psi^*(E_s^a\otimes J^{li})\psi\big)\right). \tag{4.32}$$

4.4.2. GRADIENT OF THE PENALTY FUNCTIONS

To obtain the total function over which we want to use gradient descent, we simply add the penalty functions to the objective function. Therefore, we can calculate the gradient over the penalty function separately. The penalty function of which we want to calculate the gradient is

$$f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\}) = \big(\psi^*\psi - 1\big)^2 + \sum_s\left\|\left(\sum_a G(X_s^a)\right) - I\right\|_F^2 + \sum_t\left\|\left(\sum_b G(Y_t^b)\right) - I\right\|_F^2. \tag{4.33}$$

We start of by calculating the (Wirtinger) gradient with respect to ψ,

$$\begin{aligned}\nabla_\psi f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\} &= 2\frac{\partial(\psi^*\psi-1)^2}{\partial\psi^*}\\ &= 4(\psi^*\psi-1)\psi.\end{aligned} \tag{4.34}$$

Now we want to determine the gradient with respect to the elements of X_s^a and Y_t^b. This is done in 2 steps, first we calculate the derivative for the with respect to $(E_{s_1}^{a_1})_{ij}$ (the diagonal and non diagonal elements separate) and then we apply the chain rule in 4.39. The first step is done in four sub-steps, first we calculate the W- and CW-derivative and of the diagonal elements and then we calculate the W- and CW-derivative of the none diagonal elements. The W-derivative of the diagonal element is equal to,

$$\begin{aligned}\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial(E_{s_1}^{a_1})_{ii}} &= \frac{\partial\sum_s\left(\left(\left(\sum_a(E_s^a)_{ii}\right)-1\right)\left(\left(\sum_a(E_s^a)_{ii}\right)-1\right)^*\right)}{\partial(E_{s_1}^{a_1})_{ii}}\\ &= \left(\left(\sum_a(E_{s_1}^{a_1})_{ii}^*\right)-1\right).\end{aligned} \tag{4.35}$$

which makes sense, as the constraint only says something about the sum. The derivative is the same for F. Similarly the conjugated derivative is equal to,

$$\begin{aligned}\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial(E_{s_1}^{a_1})_{ii}^*} &= \frac{\partial\sum_s\left(\left(\left(\sum_a(E_s^a)_{ii}\right)-1\right)\left(\left(\sum_a(E_s^a)_{ii}\right)-1\right)^*\right)}{\partial\left(E_{s_1}^{a_1}\right)_{ii}^*}\\ &= \left(\left(\sum_a(E_{s_1}^{a_1})_{ii}\right)-1\right).\end{aligned} \tag{4.36}$$

This, of course is not equal to zero as the f_{penalty} is not holomorphic. For the non diagonal element the W- and CW-derivatives respectively are

$$\begin{aligned}\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial(E_{s_1}^{a_1})_{ij}} &= \frac{\partial\sum_s\left(\left(\sum_a(E_s^a)_{ii}\right)\left(\sum_a(E_s^a)_{ij}\right)^*\right)}{\partial(E_{s_1}^{a_1})_{ij}}\\ &= \sum_a(E_{s_1}^{a_1})_{ij}^*,\end{aligned} \tag{4.37}$$

and

$$\begin{aligned}\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\})}{\partial(E_{s_1}^{a_1})_{ij}^*} &= \frac{\partial\sum_s\left(\left(\sum_a(E_s^a)_{ii}\right)\left(\sum_a(E_s^a)_{ij}\right)^*\right)}{\partial(E_{s_1}^{a_1})_{ij}^*}\\ &= \sum_a(E_{s_1}^{a_1})_{ij}.\end{aligned} \tag{4.38}$$

Keep in mind this is only part of the gradient, as the total Equation 4.33 is expressed in terms of X_s^a and Y_t^b. In the next calculation the gradient is calculated. In this case f_{penalty}

is not holomorphic and therefore the conjugate derivative is not equal to zero, as was the case for f_{obj}. This leaves us with

$$\begin{aligned} 2\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\}}{\partial (X_s^a)^*_{kl}} &= 2\text{Tr}\left(\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial (X_s^a)^H(X_s^a)}\right)^T \frac{\partial ((X_s^a)^H(X_s^a))}{\partial (X_s^a)^*_{kl}}\right) \\ &+ 2\text{Tr}\left(\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)^T \left(\frac{\partial \left((X_s^a)^H(X_s^a)\right)}{\partial (X_s^a)_{kl}}\right)^*\right). \end{aligned} \tag{4.39}$$

We apply a similar trick with the single entry matrices as before in 4.30. This results in

$$\begin{aligned} &= 2\text{Tr}\left(\left(\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\}}{\partial (X_s^a)^H(X_s^a)}\right)^T J^{lk}(X_s^a)\right) + 2\text{Tr}\left(\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)^T (X_s^a)^T J^{kl}\right) \\ &= 2\text{Tr}\left(J^{lk}(X_s^a)\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial (X_s^a)^H(X_s^a)}\right)^T\right) + 2\text{Tr}\left(J^{kl}\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)^T (X_s^a)^T\right) \\ &= 2\left(X_s^a\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial (X_s^a)^H(X_s^a)}\right)^T\right)_{kl} + 2\left(\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)^T (X_s^a)^T\right)_{lk} \\ &= 2\left(\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial (X_s^a)^H(X_s^a)}(X_s^a)^T\right)_{lk} + \left((X_s^a)\frac{\partial f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)_{kl}. \end{aligned} \tag{4.40}$$

Writing out the matrix product and fill in the function, which gives us

$$\begin{aligned} &= 2\sum_i \left((X_s^a)_{ki}\left(\frac{f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial (X_s^a)^H(X_s^a)}\right)_{li}\right) + 2\sum_i \left(\left(\frac{f_{\text{penalty}}((X_s^a)^H(X_s^a))}{\partial ((X_s^a)^H(X_s^a))^*}\right)_{il}(X_s^a)_{ki}\right) \\ &= 2\sum_i \left((X_s^a)_{ki}\left(\left(\sum_a \left((X_s^a)^H(X_s^a)\right)_{li}\right)^* - \delta(l,i)\right)\right) + 2\sum_i \left((X_s^a)_{ki}\left(\left(\sum_a \left((X_s^a)^H(X_s^a)\right)_{il}\right) - \delta(l,i)\right)\right) \\ &= 2\sum_i \left((X_s^a)_{ki}\left(\left(\sum_a \left((X_s^a)^H(X_s^a)\right)_{li}\right)^* + \left(\sum_a \left((X_s^a)^H(X_s^a)\right)_{il}\right) - 2\delta(l,i)\right)\right) \\ &= 4\sum_i \left((X_s^a)_{ki}\left(\sum_a \left((X_s^a)^H(X_s^a)\right)_{il} - \delta(l,i)\right)\right). \end{aligned} \tag{4.41}$$

For Y exactly the same steps can be taken which results in a gradient that is equal to

$$2\frac{\partial f_{\text{penalty}}(\psi,\{X_s^a\},\{Y_t^b\}}{\partial (Y_t^b)^*_{kl}} = 4\sum_i \left((Y_t^b)_{ki}\left(\sum_b \left((Y_t^b)^H(Y_t^b)\right)_{il} - \delta(l,i)\right)\right). \tag{4.42}$$

4.5. PROBLEM DESCRIPTION AND GRADIENT FOR PVMS

In the previous sections we looked at POVMs, however it is possible to add a constraint to make sure that the POVMs are projection matrices, which makes them PVMs. The constraints then are,

$$(E_s^a)^2 = E_s^a \quad \forall a, s \in A \times S, \tag{4.43}$$

and

$$(F_t^b)^2 = F_t^b \quad \forall b, t \in B \times T. \tag{4.44}$$

Both of these can be written as a penalty function, this is done in a same way as in Equation 4.10 and 4.11. The penalty function for constraint Equation 4.43 is

$$f_{\text{PVM,E}}(\{E_s^a\}) = \sum_{a,s} \left\| (E_s^a)^2 - E_s^a \right\|_F^2, \tag{4.45}$$

and for constraint Equation 4.44 the penalty function is

$$f_{\text{PVM,F}}(\{F_t^b\}) = \sum_{b,t} \left\| (F_t^b)^2 - F_t^b \right\|_F^2. \tag{4.46}$$

The last step is to calculate the gradients, when we write Equation 4.45 and 4.46 in terms of respectively X_s^a and Y_t^b. The gradient of Equation 4.45 with respect to an element of X_s^a is equal to

$$\begin{aligned}
\frac{2\partial\left(f_{\text{PVM,E}}(\{X_s^a\})\right)}{\partial (X_{s_1}^{a_1})^*_{kl}} &= 2\frac{\partial\left(\sum_{a,s}\text{Tr}\left(\left((X_s^a)^H X_s^a\right)^2 - \left((X_s^a)^H X_s^a\right)\right)^2\right)}{\partial (X_{s_1}^{a_1})^*_{kl}} \\
&= 2\frac{\partial\left(\sum_{a,s}\text{Tr}\left(\left((X_s^a)^H X_s^a\right)^4 - 2\left((X_s^a)^H X_s^a\right)^3 + \left((X_s^a)^H X_s^a\right)^2\right)\right)}{\partial (X_{s_1}^{a_1})^*_{kl}} \\
&= 2\frac{\partial\left(\sum_{a,s}\text{Tr}\left(\left((X_s^a)^H X_s^a\right)^4\right) - 2\text{Tr}\left(\left((X_s^a)^H X_s^a\right)^3\right) + \text{Tr}\left(\left((X_s^a)^H X_s^a\right)^2\right)\right)}{\partial (X_{s_1}^{a_1})^*_{kl}} \\
&= 4\text{Tr}\left(J^{lk} X_{s_1}^{a_1} \left(G(X_{s_1}^{a_1})\right)^3\right) - 6\text{Tr}\left(J^{lk} X_{s_1}^{a_1} \left(G(X_{s_1}^{a_1})\right)^2\right) + 2\text{Tr}\left(J^{lk} X_{s_1}^{a_1} G(X_{s_1}^{a_1})\right).
\end{aligned} \tag{4.47}$$

The gradient of Equation 4.46 can be determined in a similar way and is equal to,

$$\frac{2\partial\left(f_{\text{PVM,F}}(\{Y_t^b\})\right)}{\partial (Y_{t_1}^{b_1})^*_{kl}} = 4\text{Tr}\left(J^{lk} Y_{t_1}^{b_1} \left(G(Y_{t_1}^{b_1})\right)^3\right) - 6\text{Tr}\left(J^{lk} Y_{t_1}^{b_1} \left(G(Y_{t_1}^{b_1})\right)^2\right) + 2\text{Tr}\left(J^{lk} Y_{t_1}^{b_1} G(Y_{t_1}^{b_1})\right). \tag{4.48}$$

5

GRADIENT DESCENT

In this chapter, we explore different first-order descent methods. For each method, we give a brief overview of how it works, then we give an algorithm that suits the variables of our problem, and after that, we give the advantages and disadvantages.

In Section 5.1 we introduce the most basic gradient descent method, namely the fixed step size method. Next, in Section 5.2 we look at an improved version of the fixed step size method that keeps the step size variable and does an exact optimization step each iteration. This way, each iteration has to have a lower value than the previous one. After that, in Section 5.3 we change out the exact line search with a backtracking line search. We do not have to go through a full optimization step but accept a step size for which the cost function reduces enough. Subsequently, in Section 5.4 we look at momentum-based gradient descent, there are many different momentum-based methods [12], but we focus on the most basic one. In this method, we add a velocity to each iteration based on the previous steps. Lastly, in Section 5.5 we take a step from first-order methods and look at potential second-order methods.

The first three methods come from [13], whereas the momentum-based method comes from [12]. In the last section the ideas come from [14].

5.1. FIXED STEP SIZE

The most general gradient descent method works as the following. We start with a random point x_0, calculate the gradient $\nabla f(x_0)$, and then take a step in the negative direction of the gradient. So to find the next point the following formula is used,

$$x_{k+1} = x_k - t\nabla f(x_k), \tag{5.1}$$

where $t > 0$ is the step size, and $k = 0, 1, 2\ldots$ denotes the iteration number. Note that the name for step size t can be confusing, since in general $\|f(x_{k+1}) - f(x_k)\| \neq t$, only if the gradient is an unit vector. Let us now try to implement an algorithm that uses Equation 5.1. In this case f is Equation 4.20 and x is $\psi, \{X_s^a\}, \{Y_t^b\}$. Here the gradient is

split into 3 parts and reshaped in the shape of a vector and two lower diagonal matrices. The algorithm stops if $f(\{X_s^a\},\{Y_t^b\},\psi) \leq 10^{-5}$. The algorithm then is:

Algorithm 1: Gradient descent with fixed step size

Result: $\{X_s^a\}$, $\{Y_t^b\}$ and ψ such that $f(\{X_s^a\},\{Y_t^b\},\psi) \leq 10^{-5}$
Start with random $\{X_s^a\}$, $\{Y_t^b\}$ and ψ ;
choose t >0 ;
while $f(\{X_s^a\},\{Y_t^b\},\psi) > 10^{-5}$ **do**
  Calculate gradient $\{dX_s^a\}$,$\{dY_t^b\}$ and $d\psi$;
  $X = X - t * dX$;
  $Y = Y - t * dY$;
  $\psi = \psi - t * d\psi$.
end

With the right choice of t the algorithm turns out to work quite well. This correct value t has to be found through trial and error. In practice, it can be seen after 100 iterations if t is sufficiently small. If it is not sufficiently small, the cost function might increase or oscillate. If t is chosen too small, the algorithm converges slow. So an ideal algorithm has value for the step size t, that is as large as possible, while still provides convergence towards a minimum. Furthermore, it should be noted that monotonicity is not guaranteed. This whole searching for an ideal t is, of course, far from ideal. We can do better. Next, we look at a variable step size t, which hopefully works better.

5.2. EXACT LINE SEARCH

A solution to the problem of guessing to the step size is to keep t as a variable and do an one dimensional optimization for the ideal step size each step. For each iteration we choose $t = \text{argmin}_{s\geq 0} f(x - s\nabla f(x))$. This could be useful, if it would be easy to parameterize $g_x(s) = f(x - s\nabla f(x))$. This can be done with the following algorithm:

Algorithm 2: Gradient descent with exact line search

Result: $\{X_s^a\}$, $\{Y_t^b\}$ and ψ such that $f(\{X_s^a\},\{Y_t^b\},\psi) \leq 10^{-5}$
Start with random $\{X_s^a\}$, $\{Y_t^b\}$ and ψ ;
while $f(\{X_s^a\},\{Y_t^b\},\psi) > 10^{-5}$ **do**
  Calculate gradient $\{dX_s^a\}$,$\{dY_t^b\}$ and $d\psi$;
  $t = argmin_{s\geq 0} f(\{X_s^a\} - s\{dX_s^a\},\{Y_t^b\} - s\{dY_t^b\},\psi - sd\psi)$;
  $X = X - t * dX$;
  $Y = Y - t * dY$;
  $\psi = \psi - t * d\psi$.
end

The most significant benefit of this method is that the need to guess the step size is removed. The downside is that for each step, we get a new optimization problem. This could be solved with parameterization, but this makes our problem more complex and could eventually slow it down.

5.3. BACKTRACKING LINE SEARCH

When an exact line search is too complicated, one alternative is the backtracking line search. It is quite simple and depends on 2 parameters α, β, with $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$. The method starts with a step size $t = 1$, then it checks if

$$f(x + t\Delta x) \leq f(x) - \alpha t \nabla f(x)^T \Delta x. \tag{5.2}$$

Here x is the starting point, Δx the search direction and f the function that has to be optimized. In case of minimization and gradient descent $\Delta x = -\nabla f$, as is the case for our problem. Then the condition becomes

$$f(x - t\nabla f(x)) \leq f(x) - \alpha t \|\nabla f(x)\|^2. \tag{5.3}$$

If this condition is true then we know that $f(x_k)$ is larger then $f(x_{k+1})$ which make sure that the series $f(x_k)_{k\geq 1}$ is a strictly descending series. If the condition is false t is multiplied with β until the condition is true and the new found step size t is used to take a step. This is described by the following algorithm:

Algorithm 3: Gradient descent with backtracking line search

Result: $\{X_s^a\}$, $\{Y_t^b\}$ and ψ such that $f(\{X_s^a\}, \{Y_t^b\}, \psi) \leq 10^{-5}$
Start with random $\{X_s^a\}$, $\{Y_t^b\}$ and ψ ;
choose $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$;
while $f(\{X_s^a\}, \{Y_t^b\}, \psi) > 10^{-5}$ **do**
  Calculate gradient $\{dX_s^a\}$,$\{dY_t^b\}$ and $d\psi$;
  while $f(x - t\nabla f(x)) \geq f(x) - \alpha t \|\nabla f(x)\|^2$ **do**
    $t = t * \beta$
  end
  $X = X - t * dX$;
  $Y = Y - t * dY$;
  $\psi = \psi - t * d\psi$;
  update $f(\{X_s^a\}, \{Y_t^b\}, \psi)$.
end

The inner while loop will always eventually terminate since when t is small enough. Since $\alpha < 0.5$

$$f(x - t\nabla f(x)) \approx f(x) - t\|\nabla f(x)\|^2 < f(x) - \alpha t \|\nabla f(x)\|^2 \tag{5.4}$$

One of the great benefits of this method is that it can be relatively fast, especially when the cost function is easy to compute, but the gradient is not. However, it should be considered that it adds an extra computation step over fixed step size gradient descent. Furthermore, it should be noted that the backtracking parameters α and β do influence the convergence rate, but their effect is not that large [13]. Lastly, this method can have some problems with saddle points, where it first slowly converges towards the critical points and then goes down along the negative sides of the saddle.

5.4. MOMENTUM-BASED GRADIENT DESCENT

Next, we are going to look into an advanced gradient descent method. This method works similarly to the previously mentioned fixed step size method in Section 5.1. However, this method tries to incorporate some learning aspects because they remember the previous step and use them to make the next step. This done by calculating the velocity terms $\mathscr{X}_s^a$, $\mathscr{Y}_t^b$ and Ψ each iteration step. The velocity terms are equal to

$$v_k = \gamma \cdot v_{k-1} + t \cdot \nabla f(x_k). \tag{5.5}$$

Here $\gamma \in (0,1)$ is the learning rate and v_{k-1} the previous velocity term. This is applied in the following algorithm:

$$v_k = \gamma \cdot v_{k-1} + t \cdot \nabla f(x_k). \tag{5.6}$$

Here $\gamma \in (0,1)$ is the learning rate and v_{k-1} the previous velocity term. This is applied in the next algorithm:

Algorithm 4: Gradient descent with momentum

Result: $\{X_s^a\}$, $\{Y_t^b\}$ and ψ such that $f(\{X_s^a\},\{Y_t^b\},\psi) \leq 10^{-5}$
Start with random $\{X_s^a\}$, $\{Y_t^b\}$ and ψ ;
choose t >0 ;
choose $\gamma \in (0,1)$;
initialize $\mathscr{X}_s^a = 0$;
initialize $\mathscr{Y}_t^b = 0$;
initialize $\Psi = 0$;
while $f(\{X_s^a\},\{Y_t^b\},\psi) > 10^{-5}$ **do**
  Calculate gradient $\{dX_s^a\}$,$\{dY_t^b\}$ and $d\psi$;
  $\mathscr{X}_s^a = \gamma * \mathscr{X}_s^a - t * dX_s^a$;
  $\mathscr{Y}_t^b = \gamma * \mathscr{Y}_t^b - t * dY_t^b$;
  $\Psi = \gamma * \Psi - t * d\psi$;
  $X = X - \mathscr{X}_s^a$;
  $Y = Y - \mathscr{Y}_t^b$;
  $\psi = \psi - \Psi$.
end

The significant advantage is that the oscillations are dampened and convergence is accelerated. However, there is one downside: this method does not have to be descending, in contrast with the line search methods. Furthermore, if we choose the learning rate and the step size wrongly, the program keeps on overshooting each step and, therefore, diverges. This concludes our most advanced first-order method.

5.5. SECOND-ORDER METHODS

As the focus of this thesis is mostly on first-order methods, this chapter might seem a bit out of place. However, this section should be seen as creating some possibilities for faster algorithms, for in the future. Most second order methods require a Hessian and luckily it is possible to calculate the Hessian using the same Wirtinger calculus. This could make it possible to make the algorithms converge even faster. The downside could be that each

iteration might be a lot slower. Let us first start by a second-order Taylor expansion to show what the Hessian looks like. This can be done via a similar way as the first part of the proof of Theorem 3.4.2.

$$f(c+h) = f(c) + \left(\frac{\partial f}{\partial z}, \frac{\partial f}{\partial z^*}\right) \cdot \begin{pmatrix} h \\ h^* \end{pmatrix} + \frac{1}{2}(h, h^*) \cdot \begin{pmatrix} \frac{\partial^2 f}{\partial z^2} & \frac{\partial^2 f}{\partial z \partial z^*} \\ \frac{\partial^2 f}{\partial z^* \partial z} & \frac{\partial^2 f}{\partial (z^*)^2} \end{pmatrix} \cdot \begin{pmatrix} h \\ h^* \end{pmatrix} \tag{5.7}$$

Here the Hessian is $\begin{pmatrix} \frac{\partial^2 f}{\partial z^2} & \frac{\partial^2 f}{\partial z \partial z^*} \\ \frac{\partial^2 f}{\partial z^* \partial z} & \frac{\partial^2 f}{\partial (z^*)^2} \end{pmatrix}$. It be could be interesting to investigate if using second-order methods could be beneficial, since it requires extra computation work per iteration, but it could speed up the convergence.

Something else that could be done is determining the Hessian with the use of first-order methods. The Broyden–Fletcher–Goldfarb–Shanno algorithm [14] does precisely this. We only have to calculate the gradient and then do some linear algebra such that we get a matrix that approaches the Hessian.

6

COMPUTATIONAL EXPERIMENTS

In this chapter all our previous work comes together. We start with three types of bipartite quantum correlation from different papers and then try to find the states and the operators for an as low possible local dimension.

In Section 6.1 we check our work of Chapter 4. This is done by perturbing our gradient and then applying Theorem 3.4.2. Then, in Section 6.2 we try to find the states and operators of the CHSH bipartite quantum correlation that was introduced in 2.5. After that, in Section 6.3 we look at the correlations given in a recent paper [15] and try to find their operators and state. Later, in Section 6.4 we search for the operators and state of the correlations described in [16]. Finally, in Section 6.5 we compare four of the algorithms that were used to find the state and operators.

6.1. VALIDATION OF THE GRADIENT

Before we can apply our first-order gradient descent methods described in Chapter 5, we first should check if the gradient that we found in Chapter 4 is correct. This is to assure us that during the mathematical derivation, we did not make any mistakes. Thus, the mathematical derivation is a proof, whereas this section is a numerical validation of that proof.

So how do we check if the gradient points in the right direction? Let us first start by thinking about the gradient and what characterizes it. A gradient points towards the direction of the steepest increase of a function for all points on the domain by Theorem 3.4.2. Therefore if look at a function f at point c and calculate the gradient, then the directional derivative in the direction of the gradient is equal to

$$\frac{\partial f}{\partial v}(c) = \lim_{h \to 0} \frac{f(c + h * v) - f(c)}{h}, \tag{6.1}$$

where v is in the direction of the gradient. This directional derivative is larger than all other directional derivatives. So if we perturb v a small bit to v', where $\|v - v'\| = \epsilon$ then $\frac{\partial f}{\partial v'}(c)$ should always be smaller then $\frac{\partial f}{\partial v}(c)$.

The program first calculates the directional derivative of the gradient and then compared it to the directional derivative of the perturbed directions. The code to check this is the function *testgrad()* in Appendix A. The results show that the gradient of the total function, with and without PVM penalty, is likely to be correct since it did not find an improvement over the gradient.

6.2. CHSH

In this section, we test three algorithms by looking at the bipartite quantum correlation given in Section 2.5. Using only the bipartite quantum correlation (Equations 2.13, 2.14, 2.15 and 2.16) we try to recover the a state and POVMs. Furthermore, it should be noted that all three methods have the same starting point.

6.2.1. FIXED STEP SIZE

Here we apply the algorithm described in Section 5.1. The step size has to be chosen beforehand. After some trial and error, it can be seen that the largest step size for which the algorithm converges, in this case, is $t = 1/20$. In the figure below, an instance can be seen that the algorithm finds a solution with an error smaller than 1e-5 in 10532 iterations.

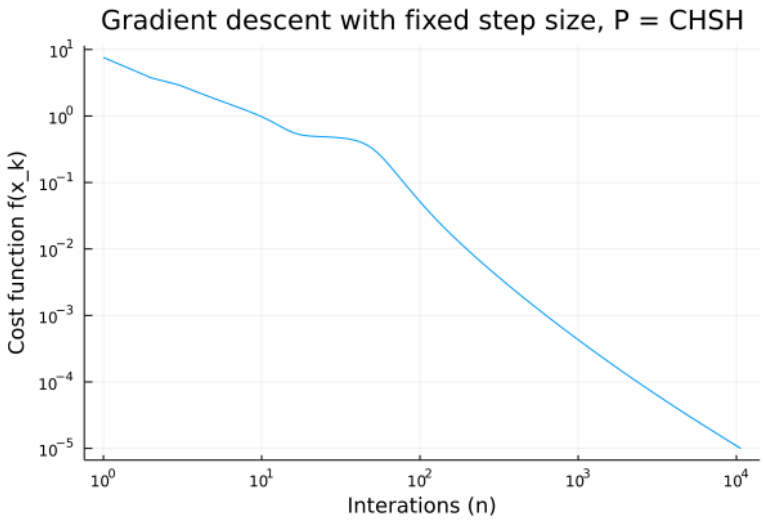

Figure 6.1: A log-log plot of an instance of the fixed step size algorithm applied to the bipartite quantum correlation. The local dimension d chosen here is 2.

Since the figure has two logarithmic axes, the figure is ideal for spotting power functions. The slope in such plots is equal to the power of $a \cdot x^k$. In our case $k \approx -1.5$.

6.2.2. BACKTRACKING LINE SEARCH

Next we apply the backtracking line search algorithm described in Section 5.3. We chose $\alpha = 0.3$ and $\beta = 0.7$ for the parameters. In the figure 6.2 an instance can be seen that the algorithm finds a solution with a error smaller then 1e-5 in 8653 iterations.

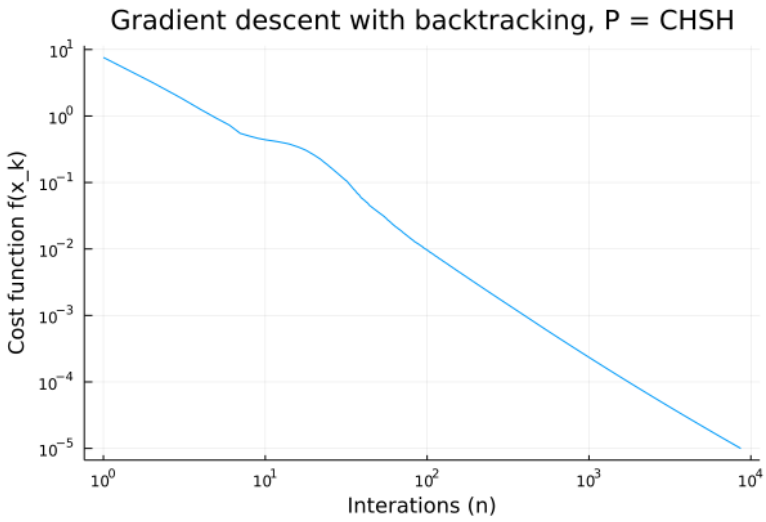

Figure 6.2: A log-log plot of an instance of the backtracking line search algorithm applied to the bipartite quantum correlation of CHSH. The local dimension d chosen here is 2.

Here we can again use logarithmic axis to determine the power of the power function $k \approx -1.55$, which is almost then the convergence of the fixed step size method.

6.2.3. PVM

We can also look for the PVMs of CHSH. We mostly do this as a proof of concept that the algorithm is also able to produce PVMs. The PVMs have a smaller subspace then the POVMs, since all PVMs are POVMs, but not the other way around. Therefore it is expected for the gradient descent to take longer; surprisingly this is not the case as can been seen in Figure 6.3. An explanation could be that the extra penalty function helps to "push" the operators in the right direction.

The power function that imitates the descent has an exponent of $k = -1.67$, which is better than the backtracking line search without PVM penalty functions. This does, however, not always have to be the case, as can been seen in Section 6.5

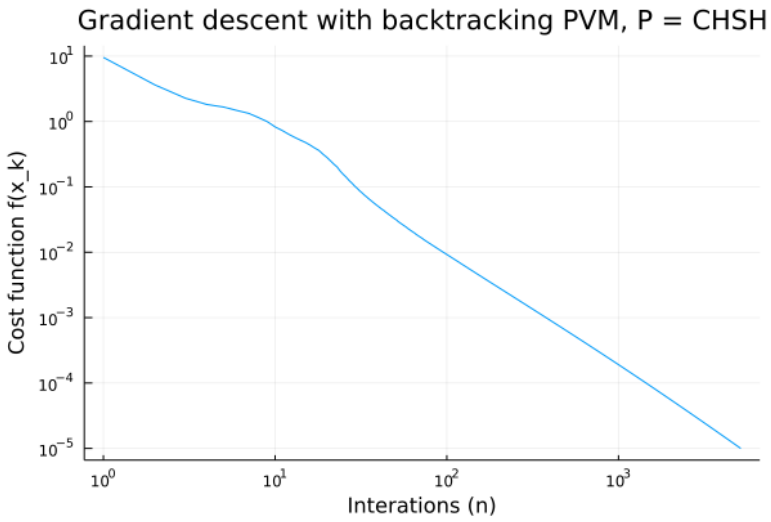

Figure 6.3: A log-log plot of an instance of the backtracking line search algorithm applied to the bipartite quantum correlation. The local dimension d chosen here is 2.

6.3. FOUR INPUT TWO OUTPUT FAMILY

Apart from CHSH, we also look at correlations of the family of where $|S| = 4$, $|T| = 4$, $|A| = 2$ and $|B| = 2$. In other words, we look at correlations with four questions and two answers. This family had been recently studied in [15]. Where it shows that for several examples the correlation is realizable in local dimension $d = 2k+1 \quad k \in \mathbb{N}^+$. The recipe for generating these correlations is given as: For $k \in \mathbb{N}^+$, let $x = \frac{4k}{2k+1}$. If both parties receive the same question $s = t$, then

$$P(a,b,s,s) = \begin{pmatrix} \frac{x}{4} & 0 \\ 0 & 1-\frac{x}{4} \end{pmatrix}. \tag{6.2}$$

Where s is fixed. If the questions differ $s \neq t$, then

$$P(a,b,s,t) = \begin{pmatrix} \frac{x(x-1)}{12} & \frac{x}{4} - \frac{x(x-1)}{12} \\ \frac{x}{4} - \frac{x(x-1)}{12} & 1 - \frac{x}{2} + \frac{x(x-1)}{12} \end{pmatrix}. \tag{6.3}$$

If we look at the situation for $s = t$, then it is only possible to give the same answer. This is called a synchronous bipartite quantum correlation. For $k = 1$ the program was able to find states and POVMs that have a cost function $f(\{X_s^a\},\{Y_t^b\},psi) \leq 10^{-5}$ in dimension $d = 3$. Next for $k = 2$ the lower bound for the dimension was shown to be $d = 5$. In the appendix of [15] the author even shows what the PVMs look like. Sadly our algorithm was not able to find the POVMs in ±25 tries it always ended at a local minimum with a cost value of $f(\{X_s^a\},\{Y_t^b\},psi) = 0.003617282390680469$. Luckily when we upped the dimension we were able to find a state and operators in local dimension $d = 9$, which is the greatest problem that the program was able to solve with $(d*d+1)/2 \cdot (|A| \cdot |B| \cdot |S| \cdot |T|) + d^2 = 45(2 \cdot 2 \cdot 4 \cdot 4) + 81 = 2961$ variables. This calculation was done on a normal

laptop and took about 100 min. The convergence can been seen in figure 6.4 The total results are shown in table 6.1.

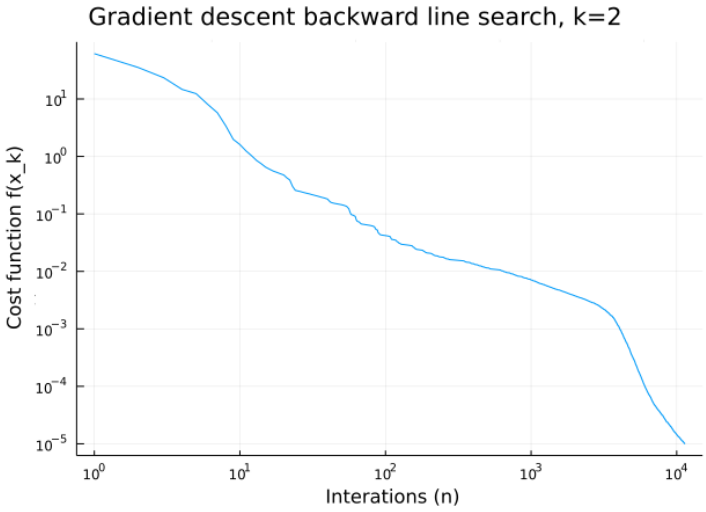

Figure 6.4: A log-log plot of an instance of the backtracking line search algorithm applied to the synchronous bipartite quantum correlation with 4 inputs, 2 outputs and $k = 2$. The local dimension d chosen here is 9.

Table 6.1: This table shows for dimension found and given for a family of bipartite quantum correlations with |A| = |B| = 2 and |S| = |T| = 4. The table also shows the amount of iterations that the program took for each k in the local dimension.

k	d given	d found	Iterations
1	3	3	10953
2	5	9	11374
3	7	?	?

6.4. CORRELATION BASED BIPARTITE QUANTUM CORRELATIONS

The next type of correlations are a bit more challenging to construct. They are also quite a bit different than previously examined correlations. This is because they have not the same amount of questions for Alice and Bob. The function that generates these correlations is *generatePGD(k)* in Appendix A. To fully understand all the steps, the reader is strongly advised to read the papers [16],[17] and [18]. The paper shows several different correlations, which are indexed by k. For each correlation C_k, the size, a lower bound for the dimension given by the paper, the dimension that was used to build the correlation, the dimension found by backtracking line search and the number of iterations it took are given in Table 6.2.

The most notable results are C_3 and C_5, for which we can find the states and operators in a lower dimension than it was built. It should be noted that the POVMs that were

Table 6.2: This table shows the results of several tries of the algorithm to find the state and operators in a dimension for different bipartite quantum correlations, with different sizes.

	|A|	|B|	|S|	|T|	d paper	d minimum	d found	iterations
C_1	2	2	1	1	2	2	2	848
C_2	2	2	2	2	2	2	2	4090
C_3	2	2	3	4	4	2	2	3194
C_4	2	2	4	7	4	4	4	125461
C_5	2	2	5	11	8	4	4	158204
C_6	2	2	6	16	8	8	?	

found create an error of less than 1e-5. It can, therefore, be possible that there are no POVMs that generate the bipartite quantum correlations in the dimension mentioned.

6.5. COMPARISON ALGORITHMS

To find the most effective gradient descent algorithm we investigate the four algorithms on the same starting location on C_4 and same local dimension $d = 4$. C_4 has $(d \cdot (d+1))/2 \cdot (|A| \cdot |B| \cdot |S| \cdot |T|) + d^2 = 656$. The algorithms used are:

1. fixed step gradient descent with step size $t = 1/50$,
2. backtracking line search with parameters $\beta = 0.7$ and $\alpha = 0.3$,
3. momentum-based gradient descent with step size $t = 1/50$ and learning rate $\gamma = 0.9$,
4. backtracking line search with parameters $\beta = 0.7$ and $\alpha = 0.3$ with the PVM penalty terms.

The results are shown in Figure 6.5.

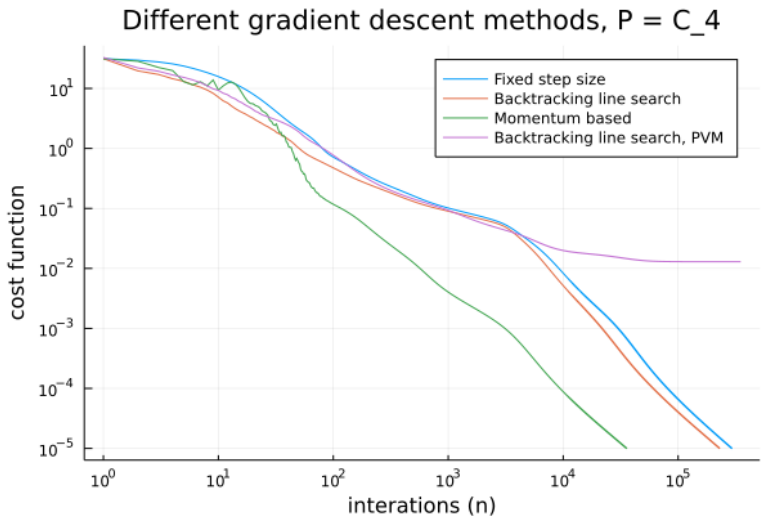

Figure 6.5: A log-log plot of four different algorithms applied to the C_4 bipartite quantum correlation. The local dimension d is 4.

If we now take a better look at the results in figure 6.5, we can note a few things:

- First of all, there is one line that does not go towards zero. This is the line of the PVM. This is because when there it is possible to find a POVM, it does not have to imply that it is possible to find a PVM. In this case, the algorithm cannot find a PVM and strands at a local minimum after 350011 iterations. This has happened for the ten different starting points in this dimension.
- Next, there are two quite similar lines, the line corresponding with backtracking line search (229724 iterations) and the line corresponding to the fixed step size method (293502 iterations). The backtracking line search works a bit better initially and is a bit smoother, but they do not differ that much. However, they have quite exciting behaviour at the kink at ±3300. Before that point, their convergence is relatively slow, but it accelerates strongly after the kink. Similar behaviour has been seen for all other correlations. If cost functions would be able to get below $\pm 10^{-3}$ it always converged towards zero. The reason why this happens is yet unknown.
- Lastly, we have the momentum-based gradient descent. The line shows many exciting behaviour at several parts. First of all, as stated in Section 5.4 it is not always descending. In the beginning, it overshoots, which can be seen by the saw-tooth patterns. This can be made less by fine-tuning the learning rate γ and step size t. The biggest surprise from looking at this line is the overall steepness it has. It converges below 10^{-5} in 35678 iterations. This is about six times faster than the backtracking line search method.

7

CONCLUSION

In the first chapter of this thesis, we give a mathematical description of bipartite quantum correlations. This is done by defining a bipartite correlation, followed by explaining how the quantum side works.

In the following chapter the Wirtinger calculus is presented. This chapter consists of explaining how two operators can be used to calculate the gradient of real-valued functions with complex variables.

Then the optimization problem is introduced. Here the Cholesky decomposition and penalty functions are used to transform all constraints into the unconstrained cost function. This unconstrained cost function perfectly suits gradient descent methods, and a few of these are then given.

These algorithms are applied to several different bipartite quantum correlations. The first is the CHSH correlations. Here the state and operators (POVMs and PVMs) are easily found in ± 2s on a regular laptop. The state and operators of a family of bipartite quantum correlations with four inputs and two outputs is also recovered. The dimension of the Hilbert space of these states and operators is higher than expected in the literature. For an even more complex bipartite quantum correlation, the state and correlations are found. The results show some states and operators with a lower local dimension that was required to build them.

The following recommendations are put forward for future studies. In light of the promising momentum-based method results, it could be exciting to investigate other advanced descent methods. Also, it would be interesting to try to find the state and operators of an experimentally determined bipartite quantum correlation. Lastly, it could be interesting to look into alternatives to the penalty functions, such a projected gradient descent.

ACKNOWLEDGEMENTS

I would like to thank Dr. D. de Laat for his overall help with the research, practical programming tips, time to meet weekly, and general enthusiasm. Furthermore, I would like to thank Dr. S. Gribling for his help with finding several good examples of interesting bipartite quantum correlations. I would also like to thanks all members of the assessment committee, for their willingness to help me with this project. I also would like to thank Boris, Joris, Tom, Suus, and Lucas for reviewing this thesis, while they should be studying for their exams. I want to, of course, thank my family, friends, and girlfriend for their support. Lastly, I would like to thank everyone from VS42 for their jokes in between study sessions and especially Joris, Dries and Gijs for their coffee in the morning.

A

CODE

In this Appendix, we give the code that is used in 6. The code is written in Version 1.5.3 of Julia.

The first part of the code contains all functions and the second part contains some commented functions. To run the specific function the lines can be uncommented.

The entire code is also available at GitHub:
https://github.com/Vgoverse/Bipartite-Quantum-Correlation-Recovery

If you are intending on using this code and want to know more about it, you can always contact me at v.goverse@gmail.com.

```
using LinearAlgebra, Plots, QuantumInformation
function Pcalc(E, F, psi)
    P = zeros(Float64,size(E,3),size(F,3),size(E,4),size(F,4))
    psipsi = psi*psi'
    for a=1:size(E,3), b=1:size(F,3), s=1:size(E,4), t=1:size(F,4)
        P[a,b,s,t] = real(tr(kron(E[:,:,a,s], F[:,:,b,t]) * psipsi))
    end
    return P
end

function CHSH()
    psi = [1/sqrt(2), 0, 0, 1/sqrt(2)]

    E = zeros(ComplexF64,(2,2,2,2))
    E[:,:,1,1] = [1/2+0im 1/2; 1/2 1/2] #E_0^0
    E[:,:,2,1] = [1/2 -1/2; -1/2+0im 1/2]#E_0^1
    E[:,:,1,2] = [1/2 -im/2; im/2 1/2] #E_1^0
    E[:,:,2,2] = [1/2 im/2; -im/2 1/2] #E_1^1

    F = zeros(ComplexF64,(2,2,2,2))
    F[:,:,1,1] = [1/2 1/(sqrt(2)*2)*(1+im); 1/(sqrt(2)*2)*(1-im) 1/2] #F_0^0
    F[:,:,2,1] = [1/2 -1/(sqrt(2)*2)*(1+im); -1/(sqrt(2)*2)*(1-im) 1/2] #F_0^1
    F[:,:,1,2] = [1/2 1/(2*sqrt(2))*(1-im); 1/(2*sqrt(2))*(1+im) 1/2] #F_1^0
    F[:,:,2,2] = [1/2 1/(2*sqrt(2))*(-1+im); 1/(2*sqrt(2))*(-1-im) 1/2] #F_1^1

    return E, F, psi
end


function costobj(P, E, F, psi)
    obj = 0.0
    P_calc = Pcalc(E, F, psi)
    for a=1:size(E,3), b=1:size(F,3), s=1:size(E,4), t=1:size(F,4)
        obj +=(P[a,b,s,t] - P_calc[a,b,s,t])^2
    end
    return obj

end

function cost(P, E, F, psi)
    obj = 0.0
```

```
    P_calc = Pcalc(E, F, psi)
    for a=1:size(E,3), b=1:size(F,3), s=1:size(E,4), t=1:size(F,4)
        obj +=(P[a,b,s,t] - P_calc[a,b,s,t])^2
    end

    #pentalties
    p_unitpsi = (psi'*psi-1)^2

    p_sumE= 0.0::Float64
    for s=1:size(E,4)
        p_sumE += tr((sum(E[:,:,a1,s] for a1 = 1:size(E,3))-I)*(sum(E[:,:,a2,s] for a2 = 1:size(E,3))-I))
    end
    p_sumF = 0.0::Float64
    for t=1:size(F,4)
        p_sumF += tr((sum(F[:,:,b1,t] for b1 = 1:size(F,3))-I)*(sum(F[:,:,b2,t] for b2 = 1:size(F,3))-I))
    end

    penalties = real(p_unitpsi +p_sumE +p_sumF)
    return(real(obj + penalties), p_unitpsi, p_sumE, p_sumF,obj,penalties)
end
function costPVM(P, E, F, psi)

    obj = 0.0
    P_calc = Pcalc(E, F, psi)
    for a=1:size(E,3), b=1:size(F,3), s=1:size(E,4), t=1:size(F,4)
        obj +=(P[a,b,s,t] - P_calc[a,b,s,t])^2
    end

    #pentalties
    p_unitpsi = (psi'*psi-1)^2

    p_sumE= 0.0::Float64
    for s=1:size(E,4)
        p_sumE += tr((sum(E[:,:,a1,s] for a1 = 1:size(E,3))-I)*(sum(E[:,:,a2,s] for a2 = 1:size(E,3))-I))
    end
    p_PVME = 0.0
    for s=1:size(E,4),a=1:size(E,3)
        p_PVME += tr((E[:,:,a,s]^2-E[:,:,a,s])*((E[:,:,a,s]^2-E[:,:,a,s])))
    end

    p_sumF = 0.0::Float64
    for t=1:size(F,4)
        p_sumF += tr((sum(F[:,:,b1,t] for b1 = 1:size(F,3))-I)*(sum(F[:,:,b2,t] for b2 = 1:size(F,3))-I))
    end
    p_PVMF = 0.0
    for t=1:size(F,4),b=1:size(F,3)
        p_PVMF += tr((F[:,:,b,t]^2-F[:,:,b,t])*((F[:,:,b,t]^2-F[:,:,b,t])))
    end

    penalties = real(p_unitpsi +p_sumE +p_sumF+ p_PVME+ p_PVMF)
    return(real(obj + penalties), p_unitpsi, p_sumE, p_sumF,obj,penalties)
end
function single!(i,j,d,E)
    E .= zeros(ComplexF64,d,d)
    E[i,j]=1
    E
end

function delta(i,j)
    if i == j
        return 1.0
    else
        return 0.0
    end
end
function gradientNOPEN!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)
    # add full cost function
    for i1 = 1:Pa, i2 = 1:Ps
        E[:,:,i1,i2] = adjoint(X[:,:,i1,i2])*X[:,:,i1,i2]
    end
    for i1 = 1:Pb, i2 = 1:Pt
        F[:,:,i1,i2] = adjoint(Y[:,:,i1,i2])*Y[:,:,i1,i2]
    end

    #dX :

    P_calc = Pcalc(E, F, psi)
    Single = zeros(ComplexF64,d,d)

    for a = 1:Pa, s = 1:Ps #X_a^s
        for k = 1:d, l in 1:k #row,column of a,s matrix
            dX[k,l,a,s] = -4*sum(X[k,i,a,s]*(P[a,b,s,t]-P_calc[a,b,s,t])* dot(psi[(((l-1)*d)+1):(l*d)],F[:,:,b,t],psi[((i-1)*d+1):(i*d)]) for i=1:d,b=1:Pb,t=1:Pt)
        end
    end

    for b = 1:Pb, t = 1:Pt #Y_b^t
        for k = 1:d, l in 1:k #row,column of a,s matrix
            dY[k,l,b,t] = -4*sum(Y[k,i,b,t]*(P[a,b,s,t]-P_calc[a,b,s,t])* dot(psi[l:d:(d^2 - d)+ l], E[:,:,a,s], psi[i:d:((d^2-d)+i )]) for i=1:d,a=1:Pa,s=1:Ps)
```

```
        end
    end
    dpsi .= zeros(ComplexF64,d^2)
    for a =1:Pa, s =1:Ps, b= 1:Pb, t= 1:Pt
        kronEF= (kron(E[:,:,a,s],F[:,:,b,t])*psi)
        dpsi .+= −4 * (P[a,b,s,t] − P_calc[a,b,s,t]) * kronEF
    end

end
function gradient!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)
    # add full cost function
    for i1 = 1:Pa, i2 = 1:Ps
        E[:,:,i1,i2] = adjoint(X[:,:,i1,i2])*X[:,:,i1,i2]
    end
    for i1 = 1:Pb, i2 = 1:Pt
        F[:,:,i1,i2] = adjoint(Y[:,:,i1,i2])*Y[:,:,i1,i2]
    end

    #dX :

    P_calc = Pcalc(E, F, psi)
    Single = zeros(ComplexF64,d,d)

    for a = 1:Pa, s = 1:Ps #X_a^s
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dX[k,l,a,s] = −4*sum(X[k,i,a,s]*(P[a,b,s,t]−P_calc[a,b,s,t])* dot(psi[(((l−1)*d)+1):(l*d)],F[:,:,b,t],psi[((i−1)*d+1):(i*d)]) for i=1:d,b=1:Pb,t=1:Pt)
            dX[k,l,a,s]+=4*sum(X[k,i,a,s] * (sum(E[i,l,a2,s] for a2 = 1:Pa)−delta(i,l)) for i = 1:d)
        end
    end

    for b = 1:Pb, t = 1:Pt #Y_b^t
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dY[k,l,b,t] = −4*sum(Y[k,i,b,t]*(P[a,b,s,t]−P_calc[a,b,s,t])* dot(psi[l:d:(d^2 − d)+ l], E[:,:,a,s], psi[i:d:((d^2−d)+i )]) for i=1:d,a=1:Pa,s=1:Ps)
            dY[k,l,b,t]+=4*sum(Y[k,i,b,t] * (sum(F[i,l,b2,t] for b2 = 1:Pb)−delta(i,l)) for i = 1:d)
        end
    end


    dpsi .= (4* ((psi'*psi) −1)*psi)
    for a =1:Pa, s =1:Ps, b= 1:Pb, t= 1:Pt
        kronEF= (kron(E[:,:,a,s],F[:,:,b,t])*psi)
        dpsi .+= −4 * (P[a,b,s,t] − P_calc[a,b,s,t]) * kronEF
    end

end

function gradientPVM!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)
    # add full cost function
    for i1 = 1:Pa, i2 = 1:Ps
        E[:,:,i1,i2] = adjoint(X[:,:,i1,i2])*X[:,:,i1,i2]
    end
    for i1 = 1:Pb, i2 = 1:Pt
        F[:,:,i1,i2] = adjoint(Y[:,:,i1,i2])*Y[:,:,i1,i2]
    end

    #dX :

    P_calc = Pcalc(E, F, psi)
    Single = zeros(ComplexF64,d,d)

    for a = 1:Pa, s = 1:Ps #X_a^s
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dX[k,l,a,s] = −4*sum(X[k,i,a,s]*(P[a,b,s,t]−P_calc[a,b,s,t])* dot(psi[(((l−1)*d)+1):(l*d)],F[:,:,b,t],psi[((i−1)*d+1):(i*d)]) for i=1:d,b=1:Pb,t=1:Pt)
            dX[k,l,a,s]+=4*sum(X[k,i,a,s] * (sum(E[i,l,a2,s] for a2 = 1:Pa)−delta(i,l)) for i = 1:d)
            dX[k,l,a,s]+=2*(4*tr(single!(l,k,d,Single)*X[:,:,a,s]*E[:,:,a,s]^3) + 2*tr(single!(l,k,d,Single)*X[:,:,a,s]*E[:,:,a,s]) −
            6*tr(single!(l,k,d,Single)*X[:,:,a,s]*E[:,:,a,s]^2) )
        end
    end

    for b = 1:Pb, t = 1:Pt #Y_b^t
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dY[k,l,b,t] = −4*sum(Y[k,i,b,t]*(P[a,b,s,t]−P_calc[a,b,s,t])* dot(psi[l:d:(d^2 − d)+ l], E[:,:,a,s], psi[i:d:((d^2−d)+i )]) for i=1:d,a=1:Pa,s=1:Ps)
            dY[k,l,b,t]+=4*sum(Y[k,i,b,t] * (sum(F[i,l,b2,t] for b2 = 1:Pb)−delta(i,l)) for i = 1:d)
            dY[k,l,b,t]+=2*(4*tr(single!(l,k,d,Single)*Y[:,:,b,t]*F[:,:,b,t]^3) + 2*tr(single!(l,k,d,Single)*Y[:,:,b,t]*F[:,:,b,t]) −
            6*tr(single!(l,k,d,Single)*Y[:,:,b,t]*F[:,:,b,t]^2) )
        end
    end


    dpsi .= (4* ((psi'*psi) −1)*psi)
    for a =1:Pa, s =1:Ps, b= 1:Pb, t= 1:Pt
        kronEF= (kron(E[:,:,a,s],F[:,:,b,t])*psi)
        dpsi .+= −4 * (P[a,b,s,t] − P_calc[a,b,s,t]) * kronEF
    end

end
```

```
function gradientNOPSl!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)
    # add full cost function
    for i1 = 1:Pa, i2 = 1:Ps
        E[:,:,i1,i2] = adjoint(X[:,:,i1,i2])*X[:,:,i1,i2]
    end
    for i1 = 1:Pb, i2 = 1:Pt
        F[:,:,i1,i2] = adjoint(Y[:,:,i1,i2])*Y[:,:,i1,i2]
    end

    P_calc = Pcalc(E, F, psi)
    Single = zeros(ComplexF64,d,d)

    for a = 1:Pa, s = 1:Ps #X_a^s
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dX[k,l,a,s] = -4*sum(X[k,i,a,s]*(P[a,b,s,t]-P_calc[a,b,s,t])* dot(psi[(((l-1)*d)+1):(l*d)],F[:,:,b,t],psi[((i-1)*d+1):(i*d)]) for i=1:d,b=1:Pb,t=1:Pt)
            dX[k,l,a,s]+=4*sum(X[k,i,a,s] * (sum(E[i,l,a2,s] for a2 = 1:Pa)-delta(i,l)) for i = 1:d)
        end
    end

    for b = 1:Pb, t = 1:Pt #Y_b^t
        for k = 1:d, l = 1:k #row,column of a,s matrix
            dY[k,l,b,t] = -4*sum(Y[k,i,b,t]*(P[a,b,s,t]-P_calc[a,b,s,t])* dot(psi[l:d:(d^2 - d)+ l], E[:,:,a,s], psi[i:d:((d^2-d)+i )]) for i=1:d,a=1:Pa,s=1:Ps)
            dY[k,l,b,t]+=4*sum(Y[k,i,b,t] * (sum(F[i,l,b2,t] for b2 = 1:Pb)-delta(i,l)) for i = 1:d)
        end
    end
end

function gradientsquare(dX,dY,dpsi)
    som = 0.0
    for a = 1:size(dX,3), s = 1:size(dX,4), i = 1:size(dX,1), j = 1:size(dX,2)
        som += abs2(dX[i,j,a,s])
    end
    for b = 1:size(dY,3), t = 1:size(dY,4), i = 1:size(dY,1), j = 1:size(dY,2)
        som += abs2(dY[i,j,b,t])
    end
    som += real(dpsi'*dpsi)
    som
end



function gradientdescent(P, d; alpha = 0.5, beta=0.9)
    println("-----------------------------------------------------------------------------------------")
    println("\n\n\n\n\n")
    println("New Gradient descent")

    Pa= size(P,1)
    Pb= size(P,2)
    Ps= size(P,3)
    Pt= size(P,4)

    println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
    # random initialization
    psi = randn(ComplexF64,d^2)
    psi /= norm(psi)

    X = zeros(ComplexF64,d,d,Pa,Ps) #dimensies uit p halen
    Y = zeros(ComplexF64,d,d,Pb,Pt)
    for a = 1:Pa,s =1:Ps,i=1:d, j=1:i
        X[i,j,a,s] = randn(ComplexF64)/d
    end
    for b = 1:Pb,t =1:Pt,i=1:d, j=1:i
        Y[i,j,b,t] = randn(ComplexF64)/d
    end

    dX = zeros(ComplexF64,d,d,Pa,Ps)
    dY = zeros(ComplexF64,d,d,Pb,Pt)
    dpsi = zeros(ComplexF64, d^2)

    Xtest = zeros(ComplexF64,d,d,Pa,Ps)
    Ytest = zeros(ComplexF64,d,d,Pb,Pt)
    psitest = zeros(ComplexF64, d^2)

    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)

    Etest = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    Ftest = zeros(ComplexF64,d,d,Pb,Pt)
    c_array = []
    t1 = 1
    t3 = 0 #counter
    while t1< 1000002
        gradient!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)

        for a = 1:Pa, s = 1:Ps
            E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
        end
```

```
        for b = 1:Pb, t = 1:Pt
            F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
        end
        c = cost(P, E, F, psi)
        if t1 % 100 ==1
            println(t1,c) #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
        end
        if t1 % 100 == 0
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, legend = false, title = "Gradient descent backward linesearch, P = C_6", xlabel = "Interations (n)", ylabel
            ↪ = "Cost function f(x_k)"))
        end
        t2 =1.0::Float64
        while true
            Xtest = X - t2.*dX
            Ytest = Y - t2.*dY
            psitest = psi - t2.*dpsi

            for a = 1:Pa, s = 1:Ps
                Etest[:,:,a,s] = (Xtest[:,:,a,s]')*Xtest[:,:,a,s]
            end

            for b = 1:Pb, t = 1:Pt
                Ftest[:,:,b,t] = (Ytest[:,:,b,t]')*Ytest[:,:,b,t]
            end

            if (cost(P,Etest,Ftest,psitest)[1]) < c[1]- alpha*t2 *gradientsquare(dX,dY,dpsi)
                X = Xtest
                Y = Ytest
                psi = psitest
                push!(c_array,c[1])
                break
            else
                t2 *= beta
                if t2 < 10.0^(-10)
                    X -= dX
                    Y -= dY
                    psi -= dpsi
                    print("t2 heel klein")
                    return(c[1],t1)
                    break
                end
            end

        end
        if abs(c[1]) < 1e-5
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, legend = false, title = "Gradient descent backward linesearch, P = C_6", xlabel = "Interations (n)", ylabel
            ↪ = "Cost function f(x_k)"))
            println("E, $E")
            println("X , $X")
            println("F, $F")
            println("Y, $Y")
            println("psi $psi")
            output_file = open("output_file.jl","w") # this will create a file named output_file.jl, where we will write the data
            write(output_file, "\n\n\n\n a:$Pa b:$Pb s:$Ps t:$Pt d:$d; \n \n")
            write(output_file, "X = ") # writes A =
            show(output_file, X) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "E = ") # writes A =
            show(output_file, E) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "Y = ") # writes A =
            show(output_file, Y) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "F = ") # writes A =
            show(output_file, F) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "psi = ") # writes A =
            show(output_file, psi) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            close(output_file)
            println(t1)
            return(c[1])
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return(10)
        end
        t1+=1
    end

    return(c[1])
    println("E, $E")
    println("X , $X")
    println("F, $F")
    println("Y, $Y")
    println("psi $psi")
end

function gradientdescentPVM(P, d; alpha = 0.5, beta=0.9, Pname = "P")
    println("--------------------------------------------------------------------------------")
```

```
println("\n\n\n\n\n")
println("New Gradient descent")

Pa= size(P,1)
Pb= size(P,2)
Ps= size(P,3)
Pt= size(P,4)

println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
# random initialization
psi = randn(ComplexF64,d^2)
psi /= norm(psi)

X = zeros(ComplexF64,d,d,Pa,Ps) #dimensies uit p halen
Y = zeros(ComplexF64,d,d,Pb,Pt)
for a = 1:Pa,s =1:Ps,i=1:d, j=1:i
    X[i,j,a,s] = randn(ComplexF64)/d
end
for b = 1:Pb,t =1:Pt,i=1:d, j=1:i
    Y[i,j,b,t] = randn(ComplexF64)/d
end


dX = zeros(ComplexF64,d,d,Pa,Ps)
dY = zeros(ComplexF64,d,d,Pb,Pt)
dpsi = zeros(ComplexF64, d^2)

Xtest = zeros(ComplexF64,d,d,Pa,Ps)
Ytest = zeros(ComplexF64,d,d,Pb,Pt)
psitest = zeros(ComplexF64, d^2)

E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
F = zeros(ComplexF64,d,d,Pb,Pt)

Etest = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
Ftest = zeros(ComplexF64,d,d,Pb,Pt)
c_array = []
t1 = 1
t3 = 0 #counter
while t1< 1000002
    gradientPVM!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)

    for a = 1:Pa, s = 1:Ps
        E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
    end
    for b = 1:Pb, t = 1:Pt
        F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
    end
    c = costPVM(P, E, F, psi)
    if t1 % 100 ==1
        println(t1,c) #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
    end
    if t1 % 100 == 0
        display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, title = "gradient descent $Pname", xlabel = "interations (n)", ylabel = "cost function"))
    end
    t2 =1.0::Float64
    while true
        Xtest = X - t2.*dX
        Ytest = Y - t2.*dY
        psitest = psi - t2.*dpsi

        for a = 1:Pa, s = 1:Ps
            Etest[:,:,a,s] = (Xtest[:,:,a,s]')*Xtest[:,:,a,s]
        end

        for b = 1:Pb, t = 1:Pt
            Ftest[:,:,b,t] = (Ytest[:,:,b,t]')*Ytest[:,:,b,t]
        end

        if (costPVM(P,Etest,Ftest,psitest)[1]) < c[1]- alpha*t2 *gradientsquare(dX,dY,dpsi)
            X = Xtest
            Y = Ytest
            psi = psitest
            push!(c_array,c[1])
            break
        else
            t2 *= beta
            if t2 < 10.0^(-20)
                X -= dX
                Y -= dY
                psi -= dpsi
                print("t2 heel klein")
                return(c[1],t1)
                break
            end
        end

    end

    if abs(c[1]) < 1e-5
```

```
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, title = "gradient descent PCHSH", xlabel = "interations (n)", ylabel = "cost function"))
            println("Geweldig gedaan!! amount of iterations: $t1")
            println("E, $E")
            println("X , $X")
            println("F, $F")
            println("Y, $Y")
            println("psi $psi")
            output_file = open("output_file.jl","w") # this will create a file named output_file.jl, where we will write the data
            write(output_file, "\n\n\n\n a:$Pa b:$Pb s:$Ps t:$Pt d:$d; \n \n")
            write(output_file, "X = ") # writes A =
            show(output_file, X) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "E = ") # writes A =
            show(output_file, E) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "Y = ") # writes A =
            show(output_file, Y) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "F = ") # writes A =
            show(output_file, F) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "psi = ") # writes A =
            show(output_file, psi) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            close(output_file)
            return(c[1])
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return(10)
        end

        t1+=1
    end

    return(c[1])
    println("E, $E")
    println("X , $X")
    println("F, $F")
    println("Y, $Y")
    println("psi $psi")
end

function gradientdescentNOPSI(P, d; alpha = 0.5, beta=0.9)
    println("-----------------------------------------------------------------------------------------------------------")
    println("\n\n\n\n\n")
    println("New Gradient descent")



    Pa= size(P,1)
    Pb= size(P,2)
    Ps= size(P,3)
    Pt= size(P,4)

    println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
    # random initialization
    psi = zeros(ComplexF64,d^2)
    psi[1] = 1/sqrt(2)
    psi[d^2] = 1/sqrt(2)

    X = zeros(ComplexF64,d,d,Pa,Ps) #dimensies uit p halen
    Y = zeros(ComplexF64,d,d,Pb,Pt)
    for a = 1:Pa,s =1:Ps,i=1:d, j=1:i
        X[i,j,a,s] = randn(ComplexF64)/d
    end
    for b = 1:Pb,t =1:Pt,i=1:d, j=1:i
        Y[i,j,b,t] = randn(ComplexF64)/d
    end


    dX = zeros(ComplexF64,d,d,Pa,Ps)
    dY = zeros(ComplexF64,d,d,Pb,Pt)
    dpsi = zeros(ComplexF64, d^2)

    Xtest = zeros(ComplexF64,d,d,Pa,Ps)
    Ytest = zeros(ComplexF64,d,d,Pb,Pt)
    psitest = zeros(ComplexF64, d^2)

    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)

    Etest = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    Ftest = zeros(ComplexF64,d,d,Pb,Pt)
    c_array = []
    t1 = 1
    t3 = 0 #counter
    while t1< 1000002
        gradientNOPSI!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
```

```
        for a = 1:Pa, s = 1:Ps
            E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
        end
        for b = 1:Pb, t = 1:Pt
            F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
        end
        c = cost(P, E, F, psi)
        if t1 % 100 ==1
            println(t1,c) #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
        end
        if t1 % 100 == 0
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log))
        end
        t2 =1.0::Float64
        while true
            Xtest = X - t2.*dX
            Ytest = Y - t2.*dY
            # psitest = psi - t2.*dpsi

            for a = 1:Pa, s = 1:Ps
                Etest[:,:,a,s] = (Xtest[:,:,a,s]')*Xtest[:,:,a,s]
            end

            for b = 1:Pb, t = 1:Pt
                Ftest[:,:,b,t] = (Ytest[:,:,b,t]')*Ytest[:,:,b,t]
            end

            if (cost(P,Etest,Ftest,psi)[1]) < c[1]- alpha*t2 *gradientsquare(dX,dY,dpsi)
                X = Xtest
                Y = Ytest
                push!(c_array,c[1])
                break
            else
                t2 *= beta
                if t2 < 10.0^(-10)
                    X -= dX
                    Y -= dY
                    psi -= dpsi
                    print("t2 heel klein")
                    return(c[1],t1)
                    break

                end
            end

        end

        if abs(c[1]) < 1e-5
            display(plot(1:(t1), c_array,xaxis=:log, yaxis=:log))
            println("Geweldig gedaan!! amount of iterations: $t1")
            println("E, $E")
            println("X , $X")
            println("F, $F")
            println("Y, $Y")
            println("psi $psi")
            output_file = open("output_file.jl","w") # this will create a file named output_file.jl, where we will write the data
            write(output_file, "\n\n\n\n a:$Pa b:$Pb s:$Ps t:$Pt d:$d; \n \n")
            write(output_file, "X = ") # writes A =
            show(output_file, X) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "E = ") # writes A =
            show(output_file, E) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "Y = ") # writes A =
            show(output_file, Y) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "F = ") # writes A =
            show(output_file, F) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "psi = ") # writes A =
            show(output_file, psi) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            close(output_file)
            return(c[1])
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return(10)
        end
        t1+=1
    end

    return(c[1])
    println("E, $E")
    println("X , $X")
    println("F, $F")
    println("Y, $Y")
    println("psi $psi")
end
function gradientdescentNOPEN(P, d; alpha = 0.5, beta=0.95)
```

```
println("-----------------------------------------------------------------------------------------")
println("\n\n\n\n\n\n")
println("New Gradient descent")



Pa= size(P,1)
Pb= size(P,2)
Ps= size(P,3)
Pt= size(P,4)

println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
# random initialization
psi = rand(ComplexF64,d^2)
psi /= norm(psi)

X = rand(ComplexF64,d,d,Pa,Ps) ./d #dimensies uit p halen
Y = rand(ComplexF64,d,d,Pb,Pt) ./d

dX = zeros(ComplexF64,d,d,Pa,Ps)
dY = zeros(ComplexF64,d,d,Pb,Pt)
dpsi = zeros(ComplexF64, d^2)

Xtest = zeros(ComplexF64,d,d,Pa,Ps)
Ytest = zeros(ComplexF64,d,d,Pb,Pt)
psitest = zeros(ComplexF64, d^2)

E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
F = zeros(ComplexF64,d,d,Pb,Pt)

Etest = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
Ftest = zeros(ComplexF64,d,d,Pb,Pt)

t1 = 1
t3 = 0 #counter
while t1< 1002
    gradientNOPEN!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)

    for a = 1:Pa, s = 1:Ps
        E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
    end
    for b = 1:Pb, t = 1:Pt
        F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
    end
    c = costobj(P, E, F, psi)

    println(t1,c) #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
    t2 =1.0
    while true
        Xtest = X - t2.*dX
        Ytest = Y - t2.*dY
        psitest = psi - t2.*dpsi

        for a = 1:Pa, s = 1:Ps
            Etest[:,:,a,s] = (Xtest[:,:,a,s]')*Xtest[:,:,a,s]
        end

        for b = 1:Pb, t = 1:Pt
            Ftest[:,:,b,t] = (Ytest[:,:,b,t]')*Ytest[:,:,b,t]
        end

        if (costobj(P,Etest,Ftest,psitest)[1]) < c[1]- alpha*t2 *gradientsquare(dX,dY,dpsi)
            X = Xtest
            Y = Ytest
            psi = psitest
            break
        else
            t2 *= beta
            if t2 < 10.0^(-20)
                X -= dX
                Y -= dY
                psi -= dpsi
                print("t2 heel klein")
                t3 += 1
                if t3 > 10
                    return(10)
                end
                break
            end
        end

    end

    # println("E, $E")
    # println("F, $F")
    if abs(c[1]) < 1e-8

        println("Geweldig gedaan!! amount of iterations: $t1")
        println("E, $E")
        println("X , $X")
```

```
            println("F, $F")
            println("Y, $Y")
            println("psi $psi")
            output_file = open("output_file.jl","w") # this will create a file named output_file.jl, where we will write the data
            write(output_file, "\n\n\n\n\n NOPEN a:$Pa b:$Pb s:$Ps t:$Pt d:$d; \n \n")
            write(output_file, "X = ") # writes A =
            show(output_file, X) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "E = ") # writes A =
            show(output_file, E) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "Y = ") # writes A =
            show(output_file, Y) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "F = ") # writes A =
            show(output_file, F) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            write(output_file, "psi = ") # writes A =
            show(output_file, psi) # writes the content of A
            write(output_file, "; \n \n") # puts a semicolon to suppress the output and two line breaks
            close(output_file)
            return(c[1])
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return(10)
        end
        t1+=1
    end
    return(c[1])
    println("E, $E")
    println("X , $X")
    println("F, $F")
    println("Y, $Y")
    println("psi $psi")
end

function gradientdescentMomentum(P, d; t= 1/20, gamma = 0.9)
    println("-----------------------------------------------------------------------------------------------------------")
    println("\n\n\n\n\n\n")
    println("New Gradient descent no backward linesearch")


    Pa= size(P,1)
    Pb= size(P,2)
    Ps= size(P,3)
    Pt= size(P,4)

    println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
    # random initialization
    psi = rand(ComplexF64,d^2)
    psi /= norm(psi)

    X = rand(ComplexF64,d,d,Pa,Ps) ./d #dimensies uit p halen
    Y = rand(ComplexF64,d,d,Pb,Pt) ./d

    dX = zeros(ComplexF64,d,d,Pa,Ps)
    dY = zeros(ComplexF64,d,d,Pb,Pt)
    dpsi = zeros(ComplexF64, d^2)


    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)

    XM = zeros(ComplexF64,d,d,Pa,Ps)
    YM = zeros(ComplexF64,d,d,Pb,Pt)
    psiM = zeros(ComplexF64, d^2)
    c_array = []

    t1 = 1
    while t1< 1000000
        gradient!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)

        for a = 1:Pa, s = 1:Ps
            E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
        end
        for b = 1:Pb, t = 1:Pt
            F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
        end

        c = cost(P, E, F, psi)
        if t1%100==1
            println(t1,c) #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
        end
        push!(c_array,c[1])

        XM = gamma * XM + t*dX
        YM = gamma * YM + t*dY
        psiM = gamma * psiM + t*dpsi
```

```
        X -= XM
        Y -= YM
        psi -= psiM



        # println("E, $E")
        # println("F, $F")
        if abs(c[1]) < 1e-5
            println("Geweldig gedaan!! amount of iterations: $t1")
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, legend = false, title = "Gradient descent with fixed stepsize, P = CHSH", xlabel = "Interations (n)", ylabel
                ↪ = "Cost function f(x_k)"))
            return c[1]
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return c[1]
        end
        t1+=1
    end

    return c_array
end

function gradientdescentNOLINE(P, d; t = 1/20)
    println("-----------------------------------------------------------------------------------------------------------")
    println("\n\n\n\n\n\n")
    println("New Gradient descent no backward linesearch")


    Pa= size(P,1)
    Pb= size(P,2)
    Ps= size(P,3)
    Pt= size(P,4)

    println("a $Pa,b $Pb, s $Ps, t $Pt, d $d")
    # random initialization
    psi = rand(ComplexF64,d^2)
    psi /= norm(psi)

    X = rand(ComplexF64,d,d,Pa,Ps) ./d #dimensies uit p halen
    Y = rand(ComplexF64,d,d,Pb,Pt) ./d

    dX = zeros(ComplexF64,d,d,Pa,Ps)
    dY = zeros(ComplexF64,d,d,Pb,Pt)
    dpsi = zeros(ComplexF64, d^2)


    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)

    c_array = []

    t1 = 1
    while t1< 10000
        gradient!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)

        for a = 1:Pa, s = 1:Ps
            E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
        end
        for b = 1:Pb, t = 1:Pt
            F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
        end

        c = cost(P, E, F, psi)
        if t1%100==1
            @info c #real(obj + penalties), p_unitpsi, p_sumE, p_sumF)
        end
        push!(c_array,c[1])

        X -= dX*t
        Y -= dY*t
        psi -= dpsi*t

        if abs(c[1]) < 1e-5
            println("Geweldig gedaan!! amount of iterations: $t1")
            display(plot(1:size(c_array,1), c_array,xaxis=:log, yaxis=:log, legend = false, title = "Gradient descent with fixed stepsize, P = CHSH", xlabel = "Interations (n)", ylabel
                ↪ = "Cost function f(x_k)"))
            return c_array
        end
        if isnan(c[1])
            println("NaN, annuleer die hele zooi maar")
            return c[1]
        end

        t1+=1
    end
end
```

```
function CorrelationGen(n,x)
    P = zeros(Float64,2,2,n,n)
    for v = 1:n, w = 1:n
        if v == w
            P[1,1,v,w] = x/n
            P[2,1,v,w] = 0
            P[1,2,v,w] = 0
            P[2,2,v,w] = 1 - x/n
        else
            P[1,1,v,w] = (x*(x-1))/(n*(n-1))
            P[2,1,v,w] = x/n - x*(x-1)/(n*(n-1))
            P[1,2,v,w] = x/n - x*(x-1)/(n*(n-1))
            P[2,2,v,w] = 1 - 2*x/n + x*(x-1)/(n*(n-1))
        end
    end
    return P
end

function find_d(P)
    d = 1
    while d<10 #loop looking for dimension
        c = gradientdescent(P,d, alpha=0.3, beta=0.8)
        if c < 1e-5
            println("Dimension is less or equal then $d")
            return d
        end

        d+=1
    end
    print("NO DIMENSION LESS THEN 10 WAS FOUND")
end

function finddims(n,m)
    x = zeros(m)
    Pcor = zeros(ComplexF64,m,2,2,n,n)

    x[1] = 1 + 1 /(n-1)
    Pcor[1,:,:,:,:] = CorrelationGen(n,x[1])

    for i in 2:(m)
        x[i] = 1 + 1 / (n - 1 - x[i-1])
        Pcor[i,:,:,:,:] = CorrelationGen(n,x[i])
    end
    println(x)

    d = zeros(Int64,m)
    for i in 3:m
        d[i] = find_d(Pcor[i,:,:,:,:] )
    end
    for i = 1:m
        println("dimension of x_$i = $(d[i])")
    end

end

function f(P,X,Y,psi)
    Pa= size(X,3)
    Pb= size(Y,3)
    Ps= size(X,4)
    Pt= size(Y,4)
    d= size(X,1)
    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)
    for a = 1:Pa, s = 1:Ps
        E[:,:,a,s] = adjoint(X[:,:,a,s])*X[:,:,a,s]
    end
    for b = 1:Pb, t = 1:Pt
        F[:,:,b,t] = adjoint(Y[:,:,b,t])*Y[:,:,b,t]
    end
    cost(P,E,F,psi)[1]
end

function directionalderivative(P,f, X, Y, psi, dX, dY, dpsi)
    ns = sqrt(gradientsquare(dX, dY, dpsi))
    dXtest = dX/ns
    dYtest = dY/ns
    dpsitest = dpsi/ns
    #println(" gradientsquare $(gradientsquare(dXtest, dYtest, dpsitest))")
    eps = 1e-7
    (f(P,X + eps*dXtest, Y + eps*dYtest, psi + eps*dpsitest) - f(P,X, Y, psi)) / eps
end

function test(P,f, X, Y, psi, dX, dY, dpsi)
    steep = directionalderivative(P,f, X, Y, psi, dX, dY, dpsi)
    println("steep: $steep")
    for i1 = 1:d, j1 =1:i1, a=1:Pa, s=1:Ps
        println("|")
        for k1 = 1:2, k2 = 1:2
```

```
            Xtest .= dX
            Xtest[i1,j1,a,s] = dX[i1,j1,a,s]+real(dX[i1,j1,a,s])*0.1*(-1)^k1 +imag(dX[i1,j1,a,s])*im*0.1*(-1)^k2
            dir = directionalderivative(P,f, X, Y, psi, Xtest, dY, dpsi)
            if dir > steep
                println("MORE i1: $i1, j1: $j1, a: $a, s: $s, k1: $k1, k2: $k2, dif: $(dir-steep)")
            else
                println("LESS i1: $i1, j1: $j1, a: $a, s: $s, k1: $k1, k2: $k2, dif: $(dir-steep)")
            end
        end
    end
    for i2 =1:d, j2=1:i2,b=1:Pb, t=1:Pt
        println("|")
        for k1 = 1:2, k2 = 1:2
            Ytest.= dY
            Ytest[i2,j2,b,t] = dY[i2,j2,b,t]+real(dY[i2,j2,b,t])*0.1*(-1)^k1+imag(dY[i2,j2,b,t])*im*(0.1)*(-1)^k2
            dir = directionalderivative(P,f, X, Y, psi, dX, Ytest, dpsi)
            if dir > steep
                println("MORE i2: $i2, j2: $j2, b: $b, t: $t, k1: $k1, k2: $k2, dif: $(dir-steep)")
            else
                println("LESS i2: $i2, j2: $j2, b: $b, t: $t, k1: $k1, k2: $k2, dif: $(dir-steep)")
            end
        end
    end
    for i3 = 1:d^2
        println("|")
        for k1 = 1:2, k2= 1:2
            psitest .= dpsi
            psitest[i3] = dpsi[i3]+real(dpsi[i3])*(0.1)*(-1)^k1+imag(dpsi[i3])*im*(0.1)(-1)^k2
            dir = directionalderivative(P,f, X, Y, psi, dX, dY, psitest)
            if dir > steep
                println("MORE i3: $i3, k1: $k1, k2: $k2, dif: $(dir-steep)")
            else
                println("LESS i3: $i3, k1: $k1, k2: $k2, dif: $(dir-steep)")
            end
        end
    end
    steep = directionalderivative(P,f, X, Y, psi, dX, dY, dpsi)
    println("steep end: $steep")
end



function testgrad()
    d =2
    Pa= 2
    Pb= 2
    Ps= 2
    Pt= 2

    psi = randn(ComplexF64,d^2)
    psi /= norm(psi)

    X = zeros(ComplexF64,d,d,Pa,Ps) #dimensies uit p halen
    Y = zeros(ComplexF64,d,d,Pb,Pt)
    for a = 1:Pa,s =1:Ps,i=1:d, j=1:i
        X[i,j,a,s] = randn(ComplexF64)/d
    end
    for b = 1:Pb,t =1:Pt,i=1:d, j=1:i
        Y[i,j,b,t] = randn(ComplexF64)/d
    end

    dX = zeros(ComplexF64,d,d,Pa,Ps)
    dY = zeros(ComplexF64,d,d,Pb,Pt)
    dpsi = zeros(ComplexF64, d^2)

    Xtest = zeros(ComplexF64,d,d,Pa,Ps)
    Ytest = zeros(ComplexF64,d,d,Pb,Pt)
    psitest = zeros(ComplexF64, d^2)

    E = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    F = zeros(ComplexF64,d,d,Pb,Pt)

    Etest = zeros(ComplexF64,d,d,Pa,Ps) #matrix of matrici
    Ftest = zeros(ComplexF64,d,d,Pb,Pt)

    gradient!(PCHSH, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt)
    test(PCHSH,f, X, Y, psi, dX, dY, dpsi)
end



function parameterana(P,d)
    alpha = 0.1:0.1:0.5
    beta = 0.1:0.1:0.9
    timemeas = zeros(Float64, size(alpha,1), size(beta,1))
    for a = 1: size(alpha,1), b = 1:size(beta,1)
        for t = 1:10
            timemeas[a,b] += @elapsed gradientdescent(P,d,alpha = alpha[a], beta = beta[b])
        end
    end
```

```
    timememeas /= 5
    return(timememeas)
end

function timetaken()
    P = PcorGen8
    cst = []
    grad = []
    Pa= size(P,1)
    Pb= size(P,2)
    Ps= size(P,3)
    Pt= size(P,4)

    for d = 1:20
        psi = randn(ComplexF64,d^2)
        psi /= norm(psi)

        X = zeros(ComplexF64,d,d,Pa,Ps) #dimensies uit p halen
        Y = zeros(ComplexF64,d,d,Pb,Pt)
        for a = 1:Pa,s =1:Ps,i=1:d, j=1:i
            X[i,j,a,s] = randn(ComplexF64)/d
        end
        for b = 1:Pb,t =1:Pt,i=1:d, j=1:i
            Y[i,j,b,t] = randn(ComplexF64)/d
        end

        dX = zeros(ComplexF64,d,d,Pa,Ps)
        dY = zeros(ComplexF64,d,d,Pb,Pt)
        dpsi = zeros(ComplexF64, d^2)



        push!(cst , @elapsed f(P,X,Y,psi))
        push!(grad, @elapsed gradient!(P, X, Y, psi, dX, dY, dpsi, d, Pa, Pb, Ps, Pt))
    end
    plot(hcat(cst,grad),xlabel = "dimension", ylabel = "time elapsed (s)",label = ["cost" "gradient"])
end



function C(r)
    M = zeros(r, binomial(r, 2)+1)
    c = 1
    for i = 1:(r-1), j=i+1:r
        M[i, c] = 1/sqrt(2)
        M[j, c] = -1/sqrt(2)
        c += 1
    end
    for j=1:r
        M[j, end] = 1/sqrt(r)
    end
    M
end

function getxandy(M)
    L, U, p = lu(M')
    s = zeros(Int, length(p))
    for i=1:length(p)
        s[p[i]] = i
    end
    U, L[s,:]'
end

function b(i, d)
    v = zeros(d)
    v[i] = 1.0
    v
end

function psi(d)
    1/sqrt(d) * sum(kron(b(i, d), b(i, d)) for i=1:d)
end

function phi(r, i)
    X = [0.0 1.0; 1.0 0.0]
    Y = [0.0 -im; im 0]
    Z = [1.0 0.0; 0.0 -1.0]
    if isodd(i)
        A = reduce(kron, [Z for _ = 1:div(i-1, 2)], init=ones(1,1))
        A = kron(A, X)
        s = div(r+1,2) - div(i+1,2)
        A = kron(A, Matrix(I,2^s,2^s))
        A
    else
        A = reduce(kron, [Z for _ = 1:div(i-2, 2)], init=ones(1,1))
        A = kron(A, Y)
        s = div(r+1,2) - div(i,2)
        A = kron(A, Matrix(I,2^s,2^s))
        A
    end
```

```
end

function Xmat(v)
    sum(v[i] * phi(length(v), i) for i=1:length(v))
end

function Ymat(v)
    sum(v[i] * transpose(phi(length(v), i)) for i=1:length(v))
end

function getcliffordrep(r)
    M = C(r)
    d = 2^div(r+1,2)
    mypsi = psi(d)
    X, Y = getxandy(M)
    mypsi, [Xmat(X[:,i]) for i=1:size(X,2)], [Ymat(Y[:,i]) for i=1:size(Y,2)]
end

function obsertoPOVM(X)
    r = size(X,1)
    d = size(X[1],1)
    XP = zeros(ComplexF64,d,d,2,r)
    for s = 1:size(X,1)
        for a = 1:2
            XP[:,:,a,s] = (I + (-1)^a*X[s])/2
        end
    end
    XP
end

function generatePGD(r)
    mpsi, X,Y = getcliffordrep(r)
    XPOVM = obsertoPOVM(X)
    YPOVM = obsertoPOVM(Y)
    PGD = Pcalc(XPOVM,YPOVM,mpsi)
end
PCHSH = Pcalc(CHSH()...)

PGD1 = generatePGD(1)
PGD2 = generatePGD(2)
PGD3 = generatePGD(3)
PGD4 = generatePGD(4)
PGD5 = generatePGD(5)
PGD6 = generatePGD(6)

PcorGen3= CorrelationGen(3,3.0/2)
PcorGen4= CorrelationGen(4,(4.0/3.0))
PcorGen4_2= CorrelationGen(4,(8.0/5.0))
PcorGen4_3= CorrelationGen(4,(12.0/7.0))
PcorGen5= CorrelationGen(5,(5.0/4.0))
PcorGen6 = CorrelationGen(6,6.0/5)
PcorGen8 = CorrelationGen(8,8.0/7)



#
# gradientdescentPVM(PcorGen4_2,5)
#
# gradientdescent(PcorGen4_2,9)
#
# gradientdescentNOLINE(PCHSH,2)#nope
# gradientdescent(PCHSH,2)
# gradientdescentMomentum(PCHSH,2)
# gradientdescentNAG(PCHSH,2,t = 1/30, gamma = 0.6)
#
# gradientdescent(PcorGen4_2,5)
#
#
# gradientdescent(PcorGen4_2,9)
# gradientdescent(PcorGen4_2,27) #300 0.016
# gradientdescent(PcorGen4_2,29) # todo
#
# gradientdescent(PCHSH,2, alpha = .5, beta = .5)
# gradientdescentNOLINE(PCHSH,2,t=20)
#
# gradientdescent(PcorGen4_3,7)
# gradientdescent(PcorGen4_3,9)
# gradientdescent(PcorGen4_3,11)
# gradientdescent(PcorGen4_3,11)
# gradientdescent(PcorGen4_3,13)
# gradientdescent(PcorGen4_3,15)
#
# gradientdescentNOPEN(PcorGen4,3)
# #
# # gradientdescentNAG(PcorGen6,5,eta = 1/130) #3??? niet verwacht gaat wel echt naar nul
# #
# # gradientdescent(PcorGen4,3) #3??? niet verwacht gaat wel echt naar nul
# gradientdescent(PcorGen8,5)
# # @profiler gradientdescent(PcorGen6,4)
# # @profiler gradientdescent(PcorGen6,5)
```

```
# # gradientdescent(PcorGen6,6)
# #
# # gradientdescent(PcorGen8,6)
# gradientdescent(PcorGen8,7)
# gradientdescent(PcorGen8,8)
# # #
# # #
# # # gradientdescentNOLINE(PCHSH,4)
# # # gradientdescentNOLINE(PcorGen4,3)
# # #
# # #
# # gradientdescentNOLINE(PcorGen3,3)
# # # find_d(PCHSH)
# # find_d(PcorGen) #3!??!
#
#
#
#
#

# gradientdescent(PGD1,2)

# @elapsed gradientdescent(PGD2,2)

# gradientdescent(PGD3,2)

# gradientdescentNOLINE(PGD4,4,t= 1/70)
# gradientdescent(PGD4,4,beta = .5)
# gradientdescentMomentum(PGD4, 4, t = 1/30, gamma = .9)
# gradientdescentPVM(PGD4,4,beta= .7, alpha = .3)

# gradientdescent(PGD5,3)

# gradientdescentMomentum(PGD6,8, t = 1/70)
#
# r=4
# mpsi, X,Y = getcliffordrep(r)
# XPOVM = obsertoPOVM(X)
# YPOVM = obsertoPOVM(Y)
# PGD4 =Pcalc(XPOVM,YPOVM,mpsi)
# gradientdescent(PGD4,4)
```

REFERENCES

[1] J. Bosma. Geometry and reconstruction of bipartite quantum correlations. 2020.

[2] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M.S. Blok, J. Ruitenberg, R.F.L. Vermeulen, R.N. Schouten, C. Abellán, et al. Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres. *Nature*, 526(7575):682–686, 2015.

[3] S. Gribling, D. de Laat, and M. Laurent. Bounds on entanglement dimensions and quantum graph parameters via noncommutative polynomial optimization. *Mathematical Programming*, 170(1):5–42, 2018.

[4] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[5] J. Briët. Grothendieck inequalities, nonlocal games and optimization. 2011.

[6] J.F. Clauser, M.A. Horne, A. Shimony, and R.A. Holt. Proposed experiment to test local hidden-variable theories. *Physical review letters*, 23(15):880, 1969.

[7] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, 47:777–780, May 1935.

[8] P. Bouboulis. Wirtinger's calculus in general hilbert spaces. 2010.

[9] K.B. Petersen and M.S. Pedersen. The matrix cookbook, nov 2012. Version 20121115.

[10] K. Kreutz-Delgado. The complex gradient operator and the cr-calculus, 2009.

[11] H. WoIkowicz and G. Styan. Bounds for elgenvalues using traces *. 2002.

[12] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[13] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, USA, 2004.

[14] R. Fletcher. *Practical Methods of Optimization; (2nd Ed.)*. Wiley-Interscience, USA, 1987.

[15] L. Mančinska, J. Prakash, and C. Schafhauser. Constant-sized robust self-tests for states and measurements of unbounded dimension. 2021.

[16] S. Gribling, D. de Laat, and M. Laurent. Matrices with high completely positive semidefinite rank. *Linear Algebra and its Applications*, 513:122–148, Jan 2017.

[17] B. S. Tsirel'son. Quantum analogues of the bell inequalities. the case of two spatially separated domains. *Journal of Soviet Mathematics*, 1987.

[18] B. S. Tsirel'son. Some results and problems on quan-tum bell-type inequalities. 1993.