Factorizing sum-of-squares polynomials

by

Quirine Langeveld

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 3, 2025 at 10:45 AM.

Student number:	560 7779	
Project duration:	April 1, 2025 – July 3, 2025	
Thesis committee:	Dr. D. de Laat,	TU Delft, supervisor
	Dr. V. Dwarka,	TU Delft

This thesis is confidential and cannot be made public until July 3, 2025.

TU Delft

Layman's summary

This thesis looks at how to tell whether or not a polynomial (a mathematical expression with powers such as $x^2 + 2x + 1$) is nonnegative, i.e., at least equal to 0. One common way to check this, is by writing it as a sum-of-squared polynomials, since squares are never negative (less than 0). This is illustrated by the previous example: $x^2 + 2x + 1 = (x + 1)^2 \geq 0$. Traditionally, finding this decomposition is done with semidefinite programming; however, this can be slow for large problems.

In this thesis we looked at different methods for finding such a decomposition, namely a neural network and the Burer-Monteiro approach. Our results showed that Burer-Monteiro seemed to be faster overall. We also found that automatic differentiation (letting the computer do more of the work), was faster than manual differentiation.

Abstract

Determining whether or not a polynomial is nonnegative is a fundamental problem with applications in various fields of mathematics and optimization. A popular relaxation technique is to write the polynomial as a sum-of-squares polynomial, since squared polynomials are automatically nonnegative. Finding such a decomposition can be reformulated as a semidefinite program, which can then be solved using interior point methods. However, due to the scalability for large semidefinite programs, we explored different methods to find a sum-of-squares decomposition, specifically a neural network and the Burer-Monteiro approach for quartics (homogeneous polynomials of degree 4).

The neural network outputs a sum-of-squares polynomial by construction, and the Burer-Monteiro approach rewrites the semidefinite program by a low-rank decomposition. A special case of univariate polynomials was studied as well for Burer-Monteiro. For both approaches, the gradient was also calculated manually.

The experiments showed that the univariate polynomials followed the theory and converged for a very low rank. After multiple tests, Burer-Monteiro seemed to converge faster in both the number of iterations and in time. Automatic differentiation outperformed manual differentiation, and overparameterization led to faster convergence. However, all tests were performed for polynomials with at most ten variables, and the only method used without parameter optimization was LBFGS, which influences the results. For future research, one can look at different optimizers and higher-degree polynomials in more variables.

Preface

This thesis marks the end of a journey through the fascinating world of polynomials, specifically sum-of-squares polynomials, that has taught me to never underestimate them again! It initially seemed like such a simple mathematical problem, but it turned out to be filled with theory and complexity.

I am grateful for the opportunity to work on this project, which has introduced me to a range of different topics, including semidefinite programming and even a bit of Julia programming. It has been sometimes challenging but mostly a rewarding experience.

First, I would like to thank my advisor, David de Laat, for his guidance and support throughout this project. He provided me with insightful feedback and gave me a very small insight into the world of academic research.

I would also like to thank Vandana Dwarka for taking the time to read my thesis and asses my work. Next, I would like to thank my friends and family for their encouragement and support during my studies. In particular, I would like to thank my friend Inez for our study sessions and shared motivation, which made a real difference. Lastly, I would like to thank my parents for encouraging my curiosity and believing in me.

Quirine Langeveld
Delft, July 2025

Contents

1

Introduction

Factorizing polynomials as a sum-of-squares is a powerful technique for several applications in mathematics and engineering. It appears in various fields of mathematics, such as graph theory, combinatorics, and fluid dynamics. In [6], sum-of-squares optimization is used for a new analytical method to determine if fluid flow is globally stable.

But what does it mean to factorize a polynomial into a sum-of-squares? In essence, it is a method to verify whether a polynomial is nonnegative everywhere. If a polynomial p can be factorized into a sum-of-squares of other polynomials, then p is guaranteed to be nonnegative for all inputs. Formally, this results in the following definition.

Definition 1.0.1. A polynomial $p \in \mathbb{R}[x_1, x_2, \ldots, x_n]$ is a sum-of-squares (SOS) if $\exists k \in \mathbb{N}$ such that $\exists q_1, q_2, \ldots, q_k \in \mathbb{R}[x_1, x_2, \ldots, x_n]$ such that: $p = \sum_{i=1}^{k} q_i^2$

Here, $\mathbb{R}[x_1, x_2, \ldots, x_n]$ denotes the set of real polynomials in n variables. Although k can be any natural number, finding the minimal k for which such polynomials exist can be desirable.

It is also important to note that every polynomial that is a sum-of-squares polynomial is nonnegative, but the converse is not true. There do exist polynomials that are nonnegative everywhere, which are not sum-of-squares polynomials. Hilbert showed that every nonnegative polynomial is guaranteed to be a sum-of-squares polynomial only in the following 3 cases: univariate polynomials, quadratic polynomials, and bivariate polynomials of degree 4. In all other cases Hilbert showed the existence of nonnegative polynomials that are not sums-of-squares [2] but he did not provide them explicitly. Motzkin found the first nonnegative polynomial that cannot be expressed as a sum-of-squares [10], given by the following equation:

$$p(x, y) = x^4 y^2 + x^2 y^4 - 3x^2 y^2 + 1 \tag{1.1}$$

This may raise the question of why it would even be worthwhile to study sum-of-squares polynomials when there still exist polynomials that are nonnegative but can't be expressed as a sum of squares expression. The answer lies in computational utility. In many practical cases, SOS approximations are sufficient.

However, determining whether or not a polynomial p is a sum-of-squares polynomial is still computationally challenging. As a result, researchers have been working on developing efficient methods for SOS decompositions. Traditional methods include semidefinite programming (SDP) and the Burer-Monteiro approach. In a recent preprint, [7], neural networks showed promising results. In this thesis, we will first reflect on those previous methods, and then we will explore which method works best in terms of runtime, convergence rate, and avoiding spurious local minima.

2

Semidefinite programming

As mentioned in the introduction, one of the methods to find a sum-of-squares factorization is semidefinite programming (SDP). It is a subfield of mathematical programming, where a linear objective function is optimized over a cone of positive semidefinite matrices in an affine space.

Semidefinite programming is a relatively new field in mathematics. It gained attention in the 90s due to multiple discoveries. First, interior-point methods, originally developed for linear programs (LPs), were extended to convex programs, including SDPs, making them computationally tractable and thus more widely applicable to use. Secondly, in 1995, an algorithm with an approximation ratio of 0.878 was found to approximate the Max-Cut problem [5], which was much better than the bounds found by LP, which were around 0.5.

In this chapter, we will rewrite the general form of an SDP program to an SDP program that finds the sum-of-squares factorization for a target polynomial p. For this, we want to make the assumption that the target polynomial p is homogeneous, meaning that all terms of p have the same degree. The proof of why this assumption can be made, will be given in the next section. Following that, we will look at what the general form of an SDP program looks like and how it can be reformulated to find the sum-of-squares factorization.

2.1. Homogenizing a polynomial

As mentioned earlier, there exist different kinds of polynomials: univariate (with only one variable), sparse (almost all coefficients are 0), but also homogeneous polynomials (polynomials where every term has the same degree). It is easier to work with a smaller, more structured class of polynomials. While it is generally impossible to transform an arbitrary polynomial into a sparse or univariate one, it is possible to convert any polynomial into a homogeneous polynomial. This process is called homogenizing.

In this section, we examine what homogenizing a polynomial exactly means and we will prove that stating that a polynomial is a sum-of-squares polynomial is equivalent to stating that the homogenized form is a sum-of-squares polynomial. This justifies the assumption that the target polynomial p is homogeneous. We can now state the theorem we aim to prove:

Theorem 2.1.1. A polynomial p is a sum-of-squares $\Leftrightarrow$ the homogenized form of p is a sum-of-squares.

In order to prove Theorem 2.1.1, we first need to formally define what it means to homogenize a polynomial. Let us consider a polynomial p with n variables and a maximum degree of $2d$.

We assume that the the maximum degree of the polynomial is even, this assumption is justified since polynomials with an odd maximum degree can never be sum-of-squares polynomials in the first place since they are not nonnegative (just set $x_i = \alpha$ for all variables and take the limit of α to $-\infty$). Furthermore, the (maximum) degree of the homogenized polynomial stays the same as the starting polynomial p. This means that 2.1.1 holds for the odd case since both p and the homogenized version of p are not sum-of-squares in that case.

Now we consider the case where the maximum degree is even, $2d$, with variables $x_1, \ldots, x_n$. We can homogenize p by introducing a new variable z. We do this by multiplying every term of p by an appropriate power of z such that every term in the resulting polynomial p' has degree $2d$ For clarification, we consider

the following non-homogeneous polynomial:

$$p(x_1, x_2) = x_1^2 + x_2 \tag{2.1}$$

This polynomial is not homogeneous, since the terms have different degrees:

- x_1^2 has degree 2,
- x_2 has degree 1.

To homogenize it, we introduce a new variable z, and rewrite the polynomial so that all terms have total degree 2:

$$p_{\text{hom}}(x_1, x_2, z) = x_1^2 + x_2 z \tag{2.2}$$

Here p_{hom} is the homogenized version of p, and now p_{hom} is a homogeneous polynomial of degree 2.

Now that we have gathered all the necessary background information, we are ready to prove Theorem 2.1.1:

Proof. (Theorem 2.1.1) Suppose p_{hom} is the homogenized form of a polynomial p and suppose we know that p_{hom} is a sum-of-squares polynomial. Then we know we can write:

$$p_{\text{hom}}(x_1, \ldots, x_n, z) = \sum_{i=1}^{k} q_i(x_1, \ldots, x_n, z)^2.$$

For some k$\in \mathbb{N}$ and $q_i \in \mathbb{R}[x_1, \ldots x_n, z]$. By setting $z = 1$, this results to

$$p(x_1, \ldots, x_n) = p_{\text{hom}}(x_1, \ldots, x_n, 1) = \sum_{i=1}^{k} q_i(x_1, \ldots, x_n, 1)^2,$$

which shows that p is a sum-of-squares polynomial.

Conversely, suppose p is a sum-of-squares polynomial with maximum degree $2d$, then by definition there exist some k$\in \mathbb{N}$ and $q_i \in \mathbb{R}[x_1, \ldots x_n]$ such that:

$$p(x_1, \ldots, x_n) = \sum_{i=1}^{k} q_i(x_1, \ldots, x_n)^2.$$

Then, by definition of homogenization,

$$p_{\text{hom}}(x_1, \ldots, x_n, z) = z^{2d} p\left(\frac{x_1}{z}, \ldots, \frac{x_n}{z}\right) = \sum_{i=1}^{k} \left(z^d q_i\left(\frac{x_1}{z}, \ldots, \frac{x_n}{z}\right)\right)^2,$$

which shows that p_{hom} is also a sum of squares polynomial. □

A homogeneous polynomial of n variables with degree $2d$ will be denoted as $p \in \mathbb{R}[x_1, x_2, \ldots, x_n]_{2d}$. Using this theorem, we can now examine the general form of an SDP and how it can be used to find a sum-of-squares factorization.

2.2. Semidefinite programming to find a sum-of-squares factorization

The general form of a semidefinite program is of the following form:

$$\begin{aligned} \text{minimize} \quad & \langle C, X \rangle \\ \text{subject to} \quad & X \succeq 0, \\ & \langle A_i, X \rangle = b_i, \quad i = 1, \ldots, m \end{aligned}$$

The notation $X \succeq 0$ means that X is a positive semidefinite matrix, which distinguishes an SDP from a regular linear program. But what does it mean for a matrix to be positive semidefinite? And how does this program relate to finding a sum-of-squares decomposition? We will first look at the definition of a positive semidefinite matrix:

Definition 2.2.1. A symmetric matrix $X \in \mathbb{R}^l$ is positive semidefinite (PSD) (denoted by $X \succeq 0$) if $v^T X v \geq 0$ for all v $\in \mathbb{R}^l$

This definition already hints at the connection to finding the sum-of-squares decomposition. When a target polynomial p can be rewritten as $p = v^T X v$ where v is a vector of monomials and $X \succeq 0$, then we know p is nonnegative. While positivity of a polynomial is generally a weaker condition than having a sum-of-squares factorization, as mentioned in the Introduction, the existence of such a positive semidefinite matrix representation is actually equivalent to the existence to the definition of a sum-of-squares decomposition.

We can formalize this link by showing that p is a sum-of-squares if and only if there exists a PSD matrix X such that:

$$p = m_d X m_d^T \tag{2.3}$$

Here, m_d denotes the vector of all monomials of degree d in the variables of the polynomial p. The goal of the next part of this section is to prove this statement. In order to do so, we first formalize the definition of the vector m_d

Definition 2.2.2. m_d is the column vector of length

$$l = \binom{n+d-1}{d}$$

containing all monomials of degree d with coefficient 1 in n variables.

Using this definition, we can now formally state the theorem that establishes the link between the positive semidefinite matrices and sum-of-squares polynomials.

Theorem 2.2.1. $p \in \mathbb{R}[x_1, \ldots, x_n]_{2d}$ is a sum-of-squares polynomial $\Leftrightarrow$ there exist a PSD matrix X $\succeq 0$ such that $p = m_d X m_d^T$

In order to prove this theorem, we need an additional theorem known as the Cholesky decomposition. This theorem allows us to decompose the positive semidefinite matrix:

Theorem 2.2.2. Let $X \succeq 0$, then there exists a lower triangular matrix L$\in \mathbb{R}^{l \times l}$ such that $X = LL^t$

Unfortunately, in order to prove theorem 2.2.2 an additional lemma is needed, also known as the spectral theorem. This lemma reads as follows:

Lemma 1 (Spectral Theorem)**.** *Let $X \in S^n$ and $r = \text{rank}(X)$, then*

$$X = \sum_{i=1}^{r} \lambda_i u_i u_i^T,$$

where $\lambda_i > 0$ are the nonzero eigenvalues and u_i are the corresponding set of orthonormal eigenvectors.

Proof. This lemma follows from the spectral theorem for symmetric matrices. □

Using lemma 1, we can first prove the theorem 2.2.2, which can then be used for 2.2.1.

Proof. (Theorem 2.2.2) Let $X \succeq 0$ and r = rank(X), then using the spectral theorem:

$$X = \sum_{i=1}^{r} \lambda_i u_i u_i^T.$$

Where $\lambda_i > 0$ are the nonzero eigenvalues and u_i are the corresponding orthonormal eigenvectors. Now, since all eigenvalues are positive let:

$$L = \left(\sqrt{\lambda_1} u_1, \sqrt{\lambda_2} u_2, \ldots, \sqrt{\lambda_r} u_r\right) A,$$

where $A \in \mathbb{R}^{r \times r}$ is an orthogonal matrix chosen such that L is lower triangular. Then,

$$LL^T = (\sqrt{\lambda_1} u_1, \ldots, \sqrt{\lambda_r} u_r) A A^T (\sqrt{\lambda_1} u_1, \ldots, \sqrt{\lambda_r} u_r)^T = X.$$

We know that X must be of full rank, since the eigenvectors are orthonormal. □

Finally, using the Cholesky theorem, we can prove theorem 2.2.1:

Proof. (Theorem 2.2.1) ($\Rightarrow$) Suppose p is a sum of squares polynomial, then:

$$p(x) = \sum_{i=1}^{k} q_i(x)^2,$$

where each q_i is a homogeneous polynomial of degree d. Then each q_i can be written as

$$q_i = v_i^T m_d,$$

where v_i is a row vector of coefficients. Then, substituting $v_i^T m_d$ for q_i this results to:

$$p = \sum_{i=1}^{k} (v_i^T m_d)^2 = \sum_{i=1}^{k} m_d^T v_i v_i^T m_d = m_d^T \left(\sum_{i=1}^{k} v_i v_i^T \right) m_d.$$

By definition, $X = \sum_{i=1}^{k} v_i v_i^T \succeq 0$, since it $m_d X m_d^T \geq 0$ for all m_d since p is a sum of squares polynomial.

($\Leftarrow$) Conversely, if there exists $X \succeq 0$ such that

$$p = m_d^T X m_d,$$

then by Theorem 2.2.2, write $X = LL^T$. Then,

$$p = m_d^T LL^T m_d = \|L^T m_d\|^2 = \sum_{i=1}^{r} (L_i^T m_d)^2,$$

where L_i is the i-th column of L. Thus, p is a sum of squares polynomial. □

Now theorem 2.2.1 is proven, we can use it to write the sum-of-squares decomposition in to the general form of an SDP program 2.2. This results in the following feasibility program:

$$p = m_d^T X m_d \tag{2.4}$$
$$s.t. X \succeq 0 \tag{2.5}$$
$$\tag{2.6}$$

The first constraint means that the coefficients of p must match those of $m_d X m_d^T$. Since both sides are polynomials, this requires us to match the coefficients for all monomials of degree $2d$.

Therefore, we can rewrite the equality constraint of the target polynomial into linear constraints on the entries of X. Doing so, we rewrite the problem in the general form:

$$\langle A_i, X \rangle = b_i, \quad i = 1, \ldots, m, \tag{2.7}$$
$$X \succeq 0. \tag{2.8}$$

Here, m is the number of possible monomials. Since p has degree $2d$ and n variables, this means that m = $\binom{n+2d-1}{2d}$. Note that since this is a feasibility problem, there is no objective that needs to be optimized.

2.3. Solving the SDP problem

As mentioned earlier, we can solve SDPs using interior point methods. Their convexity ensures that during the optimization, we won't get stuck at spurious local minima. However, there are some notable challenges when using SDPs when applied to sum-of-squares (SOS) problems. They are known for their poor scalability in both time and memory. The complexity of the SOS also grows really fast as the dimensions and the degree increase, so there is a very informal upper bound on the size of the problem that can be feasibly solved on a standard machine. That is why, in the next two chapters, we will explore two different methods: a neural network and Burer-Monteiro. We will start with the neural network

3

Neural network

Another approach to finding a sum-of-squares decomposition is by using a neural network. A neural network combines biological principles with advanced statistics to solve problems in domains such as pattern recognition and game-play (for example, Alpha Go; the first program that beat a professional Go player without handicap at a 19x19 board).

Neural networks are considered to be black boxes; they take an input, which is then taken to multiple layers of neurons and produces an output. They are mainly used in machine learning and have the strength of being able to model complex relationships without requiring explicit knowledge of the inner structure of the system, but relying on large training sets with inputs and outputs.

In this chapter, we investigate how a neural network works and how we can use it to find a sum-of-squares decomposition. But before diving into the neural network for the sum-of-squares decomposition, we begin by taking a look at a simple form of a neural network to gain a basic understanding of a neural network. We consider the following simple example of a feedforward neural network:

Definition 3.0.1. Let $M : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ be a feedforward neural network defined by

$$M(x) = (f^L \circ \sigma \circ f^{L-1} \circ \sigma \circ \cdots \circ \sigma \circ f^1)(x),$$

where for each layer $i = 1, \ldots, L$,

$$f^i(z) = W^i z + b^i,$$

with weight matrices W^i, bias vectors b^i, and

$$\sigma : \mathbb{R} \to \mathbb{R}$$

is a nonlinear activation function applied element-wise.

This definition may still appear a bit abstract, but we can gain a better understanding by looking at an example.

A common example of a neural network is the one defined in the book by Michael Nielsen [9]. In this book, he describes how a neural network is used to enable a computer to recognize handwritten digits. The network is trained to predict which number is written on paper by a human. It is illustrated in the following picture:

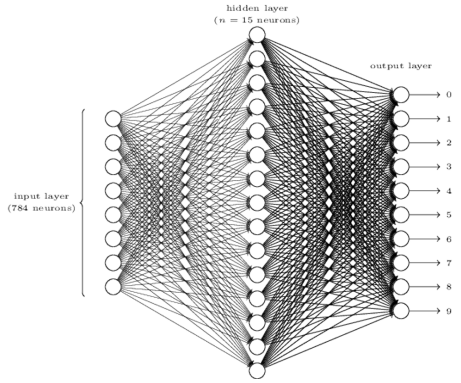


Figure 3.1: The network used to predict the handwritten numbers. The first layer is the input layer with 28x28 pixels, followed by an inner layer and the 10 output layers predicting the written number [9]

The input layer consisted of 28x28= 784 pixels, which contained the average brightness of the pixels (varying between 0 and 1). The weights of the network were trained as follows: there were 60,000 pictures taken into the network, with numbers varying from 0 to 9. Of those 60,000 pictures, 50,000 were used to train the network, and 10,000 were used to determine the accuracy of the network. The prediction was determined by the output layer, which consisted of 10 neurons. Those neurons represented the numbers 0 to 9, and if the output of a neuron was above some threshold, then that number was the predicted number. This simple network resulted in an accuracy of over 98 percent!

However, here we do not address the role of the activation functions. Those are typically chosen before training the network, and there exist various types. One of the simplest forms is called the perceptron. The output of a perceptron is defined by the following equation:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1, & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Here, w_j denotes the weight corresponding to input x_j. Since $\sum_j w_j x_j$ is also denoted as z_j, the output of layer j can also be denoted with $\sigma(z_j)$. However, this function is not differentiable, which makes training the neural network harder with gradient descent, for example. To address this, we introduce alternative neurons, starting with the sigmoid neuron:

$$\sigma(z_j) = \frac{1}{1+\exp(-z_j)} \tag{3.1}$$

Which can be illustrated in the following picture:

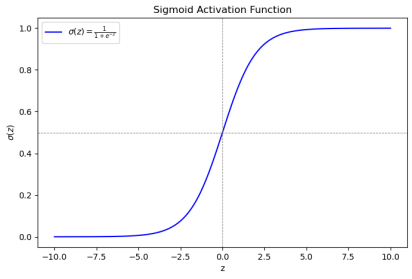


Figure 3.2: The sigmoid neuron, plotted between -10 and 10.

This function is indeed differentiable. Now, the output from the sigmoid neuron is approximately 1 when the weighted input of the other neurons is large and approximately 0 when the weighted input is small, just as it would have been for a perceptron. Other activation functions are, for example, the tanh:

$$\sigma(z_j) = \frac{e_j^z - e^{-z_j}}{e_j^z + e^{-z_j}} \tag{3.2}$$

or the RelU:

$$\sigma(z_j) = \max(0, z_j) \tag{3.3}$$

Now that we know how a neural network works in general, we can look at the neural network we will use to find a sum-of-squares factorization.

3.1. Neural network to find a sum-of-squares decomposition

In this section, we dive into how a neural network can be reformulated as an optimization problem that finds a sum-of-squares decomposition. In the previous chapter we assumed that the target polynomial $p \in \mathbb{R}[x_1, x_2, \ldots, x_n]_{2d}$ where d was taken arbitrary. For the neural network, we will assume that $d = 2$, also called quartics. We follow the work of Keren, Osadchy, and Poranne [7].

For this, we use a neural network starting with a linear layer, then an augmentation operation (non-linear layer), followed by a second linear layer.

- **A linear Layer** (A): A matrix $A \in \mathbb{R}^{m \times n}$ applied the vector $x \in \mathbb{R}^n$ containing the variables $x_1, \ldots, x_n$
- **Augmentation Layer/Non-linear Layer**: Computes the tensor product $x \otimes Ax$
- **Second Linear Layer** (B): A matrix $B \in \mathbb{R}^{k \times mn}$ applied to the tensor product

The output of the network is defined as the norm squared of the result. Since the result is a column vector of length k containing a homogeneous polynomial of degree 2, this results in a sum-of-squares polynomial of degree 4 with k squares. A sketch of the neural network is illustrated in the following picture:

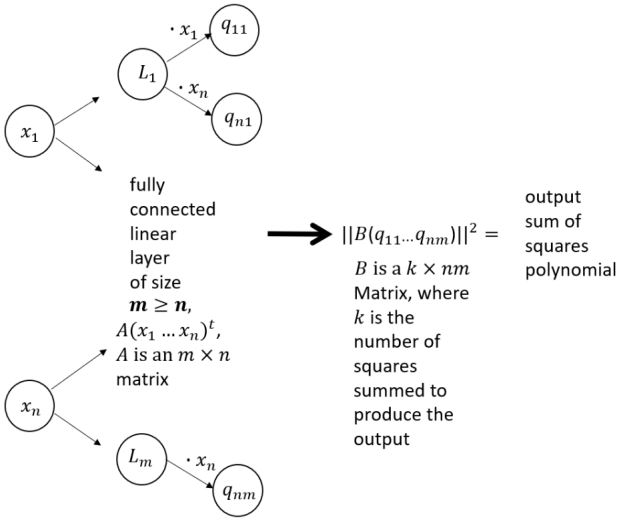


Figure 3.3: The network used to generate the SOS polynomials [7]

This leads us to the following optimization program, which we can solve using first-order optimization methods.

$$\text{minimize} \sum_{\alpha \in \mathbb{N}^n s.t. \|\alpha\|=4} (p(x)_\alpha - \|Bx \otimes Ax\|_\alpha^2)^2 \tag{3.4}$$

$$s.t. A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{k \times mn} \tag{3.5}$$

$$\tag{3.6}$$

In order to use first-order methods to solve the optimization problem, we need to provide the gradient. This can be done both manually (computing the gradient by hand) and with alternative differentiation methods (such as numerical differentiation, symbolic differentiation, or automatic differentiation). In the following section, we will derive the gradient manually.

3.2. Manual differentiation

In order to train the neural network, we need to compute the gradient with respect to its weights. These weights are the matrices A and B. Since we compute the gradient of the loss function with respect to A and B, we need to clearly state what the loss function is:

$$L(A,B) = \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - \|B(x \otimes Ax)\|_\alpha^2)^2 \tag{3.7}$$

Here α denotes the coefficient before the monomial. So it can be seen as a vector of length n, where the number at every index determines how much the variable occurs in the monomial. Since the polynomial has degree 4, we know that $\|\alpha\| = 4$. We can interpret this loss function as the l_2 norm squared for polynomials. Now we can compute the gradients. In the following section, we start with the gradient with respect to layer A, denoted by $\nabla_A L(A,B)$.

3.2.1. Gradient with respect to A

As we mentioned earlier, the gradient will be denoted by $\nabla_A L(A,B)$. To simplify calculations, even though A is a matrix, we will write the gradient as a column vector:

$$\nabla_A L(A,B) = \begin{bmatrix} \frac{\partial L(A,B)}{\partial A_{11}} & \cdots & \frac{\partial L(A,B)}{\partial A_{1n}} & \cdots & \frac{\partial L(A,B)}{\partial A_{mn}} \end{bmatrix}^T \tag{3.8}$$

We can now proceed to calculate $\nabla_A L(A,B)$;

$$\nabla_A L(A,B) = \nabla_A \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - \|B(x \otimes Ax)\|_\alpha^2)^2 \tag{3.9}$$

We can simplify the gradient by applying the chain rule to eliminate the outer square. This yields;

$$\nabla_A L(A,B) = -2 \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - \|B(x \otimes Ax)\|_\alpha^2) \nabla_A (\|B(x \otimes Ax)\|_\alpha^2) \tag{3.10}$$

We can also get rid of the square in the norm by applying the chain rule once more;

$$\nabla_A L(A,B) = -4 \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - \|B(x \otimes Ax)\|_\alpha^2)(D_A(B(x \otimes Ax))^T (B(x \otimes Ax)))_\alpha \tag{3.11}$$

We took the extra "2" to the front here. We also shifted α and the ∇_A operator here; now we first take the ∇_A and then take the α coefficient of the polynomial. This can be done due to linearity.

The remaining term to compute is the derivative of the tensor product, $B(x \otimes Ax)$, with respect to A. We can first simplify this term by taking B out, due to linearity. Then we can write the remaining column vector out, this yields;

$$D_A(B(x \otimes Ax)) = BD_A(x \otimes Ax) = BD_A(\begin{bmatrix} x_1 \sum_{i=1}^n A_{1i} xi \\ \vdots \\ x_1 \sum_{i=1}^n A_{mi} xi \\ \vdots \\ x_n \sum_{i=1}^n A_{mi} xi \end{bmatrix}) \tag{3.12}$$

If we expand $BD_A(x \otimes Ax)$ even further, then we see that it follows a specific pattern;

$$BD_A(\begin{bmatrix} x_1 \sum_{i=1}^n A_{1i} xi \\ \vdots \\ x_1 \sum_{i=1}^n A_{mi} xi \\ \vdots \\ x_n \sum_{i=1}^n A_{mi} xi \end{bmatrix}) = B \begin{bmatrix} x_1 x^T & 0^T & \cdots & \cdots & \cdots & \cdots & 0^T \\ 0^T & x_1 x^T & 0^T & \cdots & \cdots & \cdots & 0^T \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0^T & \cdots & 0^T & x_1 x^T & 0^T & \cdots & 0^T \\ \vdots & \ddots & \ddots & \ddots & x_2 x^T & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0^T \\ 0^T & \cdots & \cdots & 0^T & \cdots & \cdots & x_n x^T \end{bmatrix} = Bx \otimes I_m \otimes x^T \tag{3.13}$$

Now we can finally substitute $B(x \otimes I_m \otimes x^T)$ for $D_A(B(x \otimes Ax))$ in the equation for $\nabla_A L(A,B)$ while at the same time substituting $(x^T \otimes x^T A^T)B^T B(x \otimes Ax)$ for $\|B(x \otimes Ax)\|^2$, which then results to:

$$\nabla_A L(A,B) = -4 \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - ((x^T \otimes x^T A^T)B^T B(x \otimes Ax))_\alpha)((x^T \otimes I_m \otimes x)B^T B(x \otimes Ax))_\alpha \tag{3.14}$$

This is a form that we can implement in Julia later. Now we have determined the gradient with respect to A, it is time to determine the gradient with respect to B in the next section.

3.2.2. Gradient with respect to B

The computation of the gradient of the loss function with respect to B, denoted as $\nabla_B L(A,B)$, is very similar to that of A. Again, B is written as a column vector, with its entries indexed as follows:

$$\nabla_B L(A,B) = \begin{bmatrix} \frac{\partial L(A,B)}{\partial B_{1,11}} & \cdots & \frac{\partial L(A,B)}{\partial B_{1,1n}} & \cdots & \frac{\partial L(A,B)}{\partial B_{1,mn}} \cdots & \frac{\partial L(A,B)}{\partial B_{k,mn}} \end{bmatrix}^T \tag{3.15}$$

With a similar computation as for A, we get the following form for $\nabla_B L(A,B)$:

$$\nabla_B L(A,B) = -4 \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - \|B(x \otimes Ax)\|_\alpha^2)(D_B(B(x \otimes Ax))^T (B(x \otimes Ax)))_\alpha \tag{3.16}$$

Since all components are known except for $D_B(B(x \otimes Ax))$, we focus on this term by writing out its full expression:

$$D_B(Bx \otimes Ax) = D_B(B \begin{bmatrix} x_1 \sum_{i=1}^n A_{1i} xi \\ \vdots \\ x_1 \sum_{i=1}^n A_{mi} xi \\ \vdots \\ x_n \sum_{i=1}^n A_{mi} xi \end{bmatrix}) = \begin{bmatrix} (x \otimes Ax)^T & 0 & \cdots & 0 \\ 0 & (x \otimes Ax)^T & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots \\ 0 & 0 & \cdots & (x \otimes Ax)^T \end{bmatrix} \tag{3.17}$$

We can rewrite the last matrix into a tensor product, as follows:

$$D_B(B(x \otimes Ax)) = I_k \otimes (x \otimes Ax)^T \tag{3.18}$$

We can now substitute $I_k \otimes (x \otimes Ax)^T$ for $D_B(B(x \otimes Ax))$ and again substitute $(x^T \otimes x^T A^T)B^T B(x \otimes Ax)$ for $\|B(x \otimes Ax)\|^2$ to get to:

$$\nabla_B L(A,B) = -4 \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - ((x^T \otimes x^T A^T)B^T B(x \otimes Ax))_\alpha)(I_k \otimes (x \otimes Ax)B(x \otimes Ax))_\alpha \tag{3.19}$$

Again, this can be implemented in Julia. But one may wonder, is computation by hand even the fastest implementation? In the next section, we will explore this question by taking a closer look at alternative differentiation methods.

3.3. Alternative methods for differentiation

In order to apply first-order optimization methods, such as gradient descent or stochastic gradient descent, it is necessary to compute the gradient of the objective function. While it was manageable for the neural network to find the sum-of-squares decomposition to manually derive the gradient, it can be either not possible or tedious and impractical for other, more complex models. Moreover, it is not always optimal in terms of efficiency.

In this section, we explore alternative differentiation methods for computing gradients, such as symbolic differentiation, numerical differentiation (finite differences), and automatic differentiation. We discuss which of these methods is, in theory, the most suitable for our neural network. We begin with numerical differentiation.

3.3.1. Numerical differentiation

Numerical differentiation provides an estimate of a function's derivative by using its function values, rather than computing the derivative analytically. In this subsection, we discuss the three most common methods for numerical differentiation. The first one we introduce is called the forward difference method.

To introduce this idea, we consider a function of a single variable, $f : \mathbb{R} \to \mathbb{R}$, then the forward difference (denoted by $Q_f(h)$) is given by:

$$Q_f(h) = \frac{f(x+h) - f(x)}{h}, h > 0 \tag{3.20}$$

Here we assume h is small. The derivative of f is approximated by the slope between x and $x+h$. This is the reason why it is called "forward" difference. However, in the case of the sum-of-squares factorization, there is not a single variable input variable but multiple variables. Let $g : \mathbb{R}^s \to \mathbb{R}$ ($s = (k+1)mn$ for the neural network). In the case of multiple variables, forward difference can be used component-wise. Specifically, the approximation using the forward difference of the derivative of g with respect to the i-th is given by:

$$Q_f(h)_i = \frac{g(\mathbf{x} + h\mathbf{e}_i) - g(\mathbf{x})}{h}, \quad h > 0, i \in 1 \ldots s \tag{3.21}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{e}_i$ is the i-th standard basis vector. Since $Q_f(h)$ is only an approximation, the question might rise how accurate this even is? To be more specific, can we quantify the error between the true gradient and the $Q_f(h)$ in terms of h? Fortunately, the answer is yes. There exist an upperbound, as stated in the following theorem.

Theorem 3.3.1. Let $g : \mathbb{R} \to \mathbb{R}^s$ and $h > 0$, if $g \in C^2[\mathbf{x}, \mathbf{x} + h\mathbf{e}_i]$ then there exist an $\boldsymbol{\xi} \in (\mathbf{x}, \mathbf{x} + h\mathbf{e}_i)$ such that:

$$\left| Q_f(h)_i - \frac{\partial g}{\partial x_i}(\mathbf{x}) \right| \leq \frac{h}{2} \frac{\partial^2 g}{\partial x_i^2}(\xi).$$

Proof. By Taylor's theorem with remainder, there exists some $\boldsymbol{\xi}$ between $\mathbf{x}$ and $\mathbf{x} + h\mathbf{e}_i$ such that

$$g(\mathbf{x} + h\mathbf{e}_i) = g(\mathbf{x}) + h\frac{\partial g}{\partial x_i}(\mathbf{x}) + \frac{h^2}{2}\frac{\partial^2 g}{\partial x_i^2}(\boldsymbol{\xi}).$$

Dividing both sides by h;

$$\frac{g(\mathbf{x} + h\mathbf{e}_i) - g(\mathbf{x})}{h} = \frac{\partial g}{\partial x_i}(\mathbf{x}) + \frac{h}{2}\frac{\partial^2 g}{\partial x_i^2}(\boldsymbol{\xi}).$$

Taking the absolute value of the error,

$$\left| Q_f(h)_i - \frac{\partial g}{\partial x_i}(\mathbf{x}) \right| = \left| \frac{h}{2}\frac{\partial^2 g}{\partial x_i^2}(\boldsymbol{\xi}) \right|$$

□

In this theorem, we make the assumption that g is continuous and twice differentiable. In that case, we know that $\frac{\partial^2 g}{\partial x_i^2}(\boldsymbol{\xi})$ is bounded by some value L, and thus for small h, the error converges to 0, in $O(h)$.

However, as stated earlier, forward difference is not the only numerical differentiation method. Two other methods are the backward difference and the central difference, which we will introduce next. The backward difference is defined as follows:

$$Q_b(h)_i = \frac{g(\mathbf{x}) - g(\mathbf{x} - h\mathbf{e}_i)}{h}, \quad h > 0, i \in 1 \dots s \tag{3.22}$$

The backward difference uses the slope between $x - h$ and x, which is why we call it the backward difference. An upper bound for the error of the backward difference is given by the following theorem. The proof is similar to that of the forward difference.

Theorem 3.3.2. Let $g : \mathbb{R} \to \mathbb{R}^s$ and $h > 0$, if $g \in C^2[\mathbf{x} - h\mathbf{e}_i, \mathbf{x}]$ then there exist an $\boldsymbol{\xi} \in (\mathbf{x} - h\mathbf{e_i}, \mathbf{x})$ such that:

$$\left| Q_b(h)_i - \frac{\partial g}{\partial x_i}(\mathbf{x}) \right| \leq \frac{h}{2}\frac{\partial^2 g}{\partial x_i^2}(\boldsymbol{\xi}).$$

Which means that the error of the backward difference is also scaled with $O(h)$ Finally, the central finite difference is defined as follows:

$$Q_c(h)_i = \frac{g(\mathbf{x} + h\mathbf{e}_i) - g(\mathbf{x} - h\mathbf{e}_i)}{2h}, \quad h > 0, i \in 1 \dots s \tag{3.23}$$

This is called the central finite difference, since it uses both the $x - h$ and $x + h$. We can bound the error of the finite difference as follows:

Theorem 3.3.3. Let $g : \mathbb{R} \to \mathbb{R}^s$ and $h > 0$, if $g \in C^3[\mathbf{x} - h\mathbf{e}_i, \mathbf{x} + h\mathbf{e}_i]$ then there exist an $\boldsymbol{\xi} \in (\mathbf{x} - h\mathbf{e_i}, \mathbf{x} + h\mathbf{e_i})$ such that:

$$\left| Q_c(h)_i - \frac{\partial g}{\partial x_i}(\mathbf{x}) \right| \leq \frac{h^2}{6}\frac{\partial^3 g}{\partial x_i^3}(\boldsymbol{\xi}).$$

Note that for the finite difference, the function must be three times continuously differentiable, but the error scales better, namely with $O(h^2)$. Still, the backward difference and the forward difference are used in practice, for example, for boundary points. Besides that, the finite difference is twice as expensive when there are many values. Since for the forward difference and the backward difference, you can reuse half of the evaluations.

But why use numerical differentiation methods like the forward difference or the backward difference in the first place? If the function is differentiable, why not just compute the gradient exact?

As explained at the beginning of this chapter, this can be tedious and time-consuming, especially for complex models like neural networks. Secondly, in many practical applications, the function itself is not explicitly known! In such cases, manual differentiation is not an option.

However, numerical differentiation also has some drawbacks. Even though the "truncation error" scales with $O(h)$ for the forward difference and the backward difference, and $O(h^2)$ for the central finite difference, the total error can be larger in practice. This is due to the floating-point precision of the computer. If h is taken to be really small this might reduce the "truncation error", but the "rounding error" starts to increase from a certain point. This happens since for small h almost all terms in $g(\mathbf{x}+h\mathbf{e}_i)-g(\mathbf{x})$ cancel, which reduces precision, before dividing by h.

Given this trade-off, is numerical differentiation a good idea for computing gradients in a neural network? Probably not. We already computed the gradients by hand, so using numerical differentiation is unnecessary. Moreover, finite difference requires evaluating the full network once for each variable in the neural network, which is time-consuming ($O(s)$). Therefore, it makes sense to consider alternative methods, one of these methods is symbolic differentiation, which we will discuss in the next subsection.

3.3.2. Symbolic differentiation

Symbolic differentiation is an automated version of manual differentiation. Most of the time, it uses a tree to compute the gradient. Every node contains an elementary function here. These nodes relate to basic functions whose derivatives could be derived from elementary derivative operations like trigonometric functions, the derivative of powers, and scalar products. The derivative is then computed, using compound derivative functions like sum, product, quotient, and chain rules. This might seem much better than manual computing, or at least less error-prone than humans, and hopefully also faster. There is however a common problem with symbolic differentiation, called expression swell, which can be illustrated with the following function:

$$c(x) = a(x)b(x) \tag{3.24}$$

$$c'(x) = a'(x)b(x) + a(x)b'(x) \tag{3.25}$$

$$\tag{3.26}$$

suppose $a(x)$ also consists of two functions: $f(x), g(x), a(x) = f(x)g(x)$. Then the whole expression becomes:

$$c'(x) = c\left(f'(x)g(x) + f(x)g'(x)\right)b(x) + f(x)g(x)b'(x) \tag{3.27}$$

Now if we use symbolic differentiation careless, the computation time will increase a lot. Since f(x) and g(x) will be computed multiple times, this makes the differentiation very slow. When computing the gradient manually, this can also happen to a certain extend. However humans are capable of simplifying the gradient.

Even though this method may be a good method when used with care, since there is no previous knowledge about symbolic differentiation, it might be better not to use it for now.

A drawback of symbolic differentiation in general is that it cannot handle open-formed expressions.. An advantage compared to finite difference is that it is exact, so it is less dependent on floating-point accuracy.

Still, it can be concluded that this might not be the best method, but what about automatic differentiation?

3.3.3. Automatic Differentiation

Automatic Differentiation (AD) computes derivatives with the same accuracy as symbolic derivatives. It is an extension of symbolic differentiation. Note that with automatic differentiation, we can also differentiate open-form expressions with loops, recursions (which symbolic differentiation can't). We have two kinds of automatic differentiation methods, namely forward mode and backward mode. Both of them use a DAG (Directed Acyclic Graph). The nodes contain small elementary functions, and the edges contain the variables. These methods are mainly used for Neural Networks.

Forward mode

For automatic differentiation, the forward mode is the simplest one. We can explain it with an example;

$$f(x, y) = \ln x_1 + x_1 x_2 - \sin x_2 \tag{3.28}$$

following the example of [1]. This function can be illustrated in the following picture:

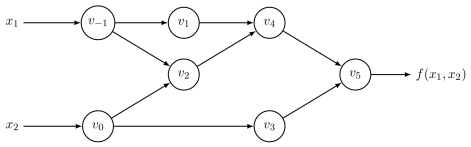
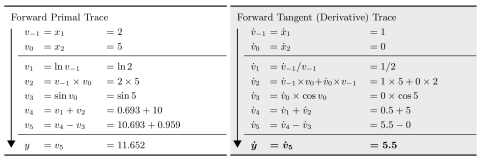


Figure 3.4: The function is illustrated as a graph, each vertex represents an operation. For example, v_2 means the product of the two incoming vertices ($x_1 x_2$), and v_1 represents the ln. operation [1]

The idea of forward mode is quite simple. Suppose we want the partial derivative with respect to x_1. Then we first define with each intermediate value a derivative:

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \tag{3.29}$$

Then we can apply the chain rule to each elementary operation. If we evaluate for every step the tangent $\dot{v}_i$ and the value (v_i), we get the required derivative. The process can be seen in the following table:



Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + v_0 \times \dot{v}_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

Figure 3.5: computing the partial derivative with respect to x_1 at x = (2,5), on the left hand side we see the process of evaluating at the different values and at the right hand side their tangent. [1]

This function was from $f : \mathbb{R}^2 \to \mathbb{R}$ and we need to go twice trough the graph, once for x_1 and once for x_2. But how does this work for a general function, $f : \mathbb{R}^n \to \mathbb{R}^m$ with input variables, $x_1 \dots x_n$ and output $y_1 \dots y_m$. Then the Jacobian looks as follows; J = $\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$ With every iteration trough the graph, we get a new column. Which means we need to go n times trough the network. This means that this approach is not ideal for the neural network, since $n \gg m$. But what about backpropagation then?

Backpropagation

Backpropagation is an algorithm used to compute gradients, primarily in neural networks. It calculates the overall error by applying the chain rule of calculus. The algorithm consists of two phases: the forward pass and the backward pass. To understand it better, we will revisit the neural network introduced at the beginning of this chapter, which predicted numbers written by humans. Before diving into the algorithm, we first take a look at the notation for the algorithm:

- a_j^l denotes the activation of neuron j in layer l. The vector of all activations in layer l is denoted by $\mathbf{a}^l \in \mathbb{R}^{n_l}$, where n_l is the number of neurons in that layer.
- w_{jk}^l is the weight connecting neuron k in layer $l-1$ to neuron j in layer l. The full weight matrix for layer l is $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$.
- b_j^l is the bias for neuron j in layer l, and $\mathbf{b}^l \in \mathbb{R}^{n_l}$ the bias vector for layer l.
- The input of neuron j in layer l is
$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l.$$
In vector form:
$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l.$$
- The activation function $\sigma(\cdot)$ is applied elementwise:
$$a_j^l = \sigma(z_j^l), \quad \text{or} \quad \mathbf{a}^l = \sigma(\mathbf{z}^l).$$

- The cost function is the mean-squared error:

$$C = \frac{1}{2}\|\mathbf{a}^L - \mathbf{y}\|^2,$$

where $\mathbf{a}^L$ is the network's output and $\mathbf{y}$ is the target output.

- The error term for neuron j in layer l is denoted by δ_j^l and is defined as:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l},$$

which represents how much the cost C changes with respect to the weighted input z_j^l of neuron j in layer l. In vector form:

$$\boldsymbol{\delta}^l = \frac{\partial C}{\partial \mathbf{z}^l} \in \mathbb{R}^{n_l},$$

The algorithm constructed is for a neural network with L layers. We will start with the forward pass. The forward pass starts at the input layer and goes all the way up to layer L. For each layer $l \in 1 \dots L$ it computes: z_j^l and from that $a^l = \sigma(z_j^l)$ for $j \in 1 \dots n_l$. All these values are stored. Then, when layer L is reached, the backward pass can start.
The backward pass starts from the loss at the output and applies the chain rule to compute the gradient of the loss with respect to each weight. First, the error term δ_j^L for each neuron j in the output layer L is computed by applying the chain rule to the cost function with respect to the activations a_j^L. For this cost function, this becomes:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = (a_j^L - y_j)\,\sigma'(z_j^L),$$

where y_j is the target output for neuron j. Note that since $\frac{\partial C}{\partial a_j^L}$ can be computed since the cost function is known and $\frac{\partial a_j^L}{\partial z_j^L}$ can be computed since z_j^L is known. Also note that σ, σ' are implemented. We can put this into vector form to obtain:

$$\boldsymbol{\delta}^L = (\mathbf{a}^L - \mathbf{y}) \odot \sigma'(\mathbf{z}^L),$$

Then for the resulting layers, l = L-1,L-2, .. 1, we get

$$\delta_j^l = \left(\sum_k w_{kj}^{l+1}\,\delta_k^{l+1}\right) \cdot \sigma'(z_j^l).$$

This comes from the fact that:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_i^l} = \frac{\partial C}{\partial a_j^l} \cdot \sigma'(z_j^l)$$

and by applying the chain rule twice:

$$\frac{\partial C}{\partial a_j^l} = \sum_k \frac{\partial C}{\partial a_k^{l+1}} \cdot \frac{\partial a_k^{l+1}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_j^l} = \sum_k \delta_k^{l+1}\, w_{kj}^{l+1}$$

with k $\in 1 \dots n_{l+1}$ Which we can write in matrix-vector form to obtain:

$$\boldsymbol{\delta}^l = \left((\mathbf{W}^{l+1})^\top \boldsymbol{\delta}^{l+1}\right) \odot \sigma'(\mathbf{z}^l).$$

Note that we know W^{l+1} from the forward phase, δ^{l+1} from the previous iteration in the backward pass, $\mathbf{z}^l$ from the feedforward phase, and σ' is stored since it is fixed.

We can then finally compute $\frac{\partial C}{\partial w_{ij}^l}$ and $\frac{\partial C}{\partial b_j^l}$ by the chain rule:

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial a_i^l} \cdot \frac{\partial a_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_{ij}^l} = \delta_i^l \, a_j^{l-1}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \, 1 = \delta_j^l \tag{3.30}$$

Which can be computed easily since δ_j^l is known from backpropagation for every $j \in 1 \dots L$ and a_j^{l-1} is known from the forward phase.
In matrix vector form, this becomes:

$$\frac{\partial C}{\partial \mathbf{W}^l} = \boldsymbol{\delta}^l (\mathbf{a}^{l-1})^\top, \quad \text{and} \quad \frac{\partial C}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l,$$

Now it becomes clear that backpropagation earns its name, since the error is computed backwards, starting at the output layer. It computes it for every input at once, which means it is faster for neural networks with a lot of inputs but a small number of outputs. Even if we have a large number of outputs, we always have a cost function, and thus the function for which we need gradients has only 1 output. This is why it is used for neural networks, since we only have to run through the entire neural network once (forward and backward), compared to the forward mode, which needs to run through the entire network for each input.

If we indeed work with automatic differentiation, we only need the loss function. This loss function can be implemented more efficiently in this case, which we will discuss in the following section:

3.4. Implementing the loss function

If we take a look back at the beginning of this chapter, we see that the loss function was defined by the following equation:

$$L(A,B) = \sum_{\alpha \in \mathbb{N}^n} (p_\alpha - (x^T \otimes x^T A^T B^T B x \otimes A x)_\alpha)^2 \tag{3.31}$$

However, finding the right terms for $x^T A^T B^T B x \otimes A x$ for each α is very time-consuming. Fortunately, in [7], they discovered a better way to do this. We will explore this method this section,

We can classify the degree 4 monomials, into the following 5 types:

- x_i^4
- $x_i^3 x_j$
- $x_i^2 x_j^2$
- $x_i^2 x_j x_k$
- $x_i x_j x_k x_l$

If we let $f = Bx \otimes Ax$ then the coefficient before the monomial x_i^4 can be rewritten as:

$$c(x_i^4) = \frac{1}{24} \frac{\partial^4}{\partial x_i^4} \left\| f \right\|^2 \tag{3.32}$$

Note that the $\frac{1}{24}$ comes from the exponent of the monomial when taking the derivative. Now, we can expand the right-hand side and apply the chain rule to the squared norm:

$$\begin{aligned}
\frac{1}{24} \frac{\partial^4}{\partial x_i^4} \|f\|^2 &= \frac{1}{24} \frac{\partial^4}{\partial x_i^3} \left(2\langle f, f_{x_i} \rangle\right) \\
&= \frac{1}{24} \frac{\partial^3}{\partial x_i^2} \left(2\left(\langle f_{x_i}, f_{x_i} \rangle + \langle f, f_{x_i x_i} \rangle\right)\right) \\
&= \frac{1}{24} \frac{\partial}{\partial x_i} \left(2\left(2\langle f_{x_i x_i}, f_{x_i} \rangle + \langle f_{x_i}, f_{x_i x_i} \rangle\right)\right) \\
&= 6\langle f_{x_i x_i}, f_{x_i x_i} \rangle
\end{aligned}$$

Since f only contains monomials of degree 2, every 3rd or higher partial derivative of f cancels. We can do a similar computation for the other monomials, this results to:

- $c(x_i^4) = \frac{1}{4}\langle f_{x_i x_i}, f_{x_i x_i}\rangle$
- $c(x_i^3 x_j) = \langle f_{x_i x_i}, f_{x_i x_j}\rangle$
- $c(x_i^2 x_j^2) = \langle f_{x_i x_j}, f_{x_i x_j}\rangle + \frac{1}{2}\langle f_{x_i x_i}, f_{x_j x_j}\rangle$
- $c(x_i^2 x_j x_k) = \langle f_{x_i x_i}, f_{x_j x_k}\rangle + 2\langle f_{x_i x_j}, f_{x_i x_k}\rangle$
- $c(x_i x_j x_k x_l) = 2\big(\langle f_{x_i x_j}, f_{x_k x_l}\rangle + \langle f_{x_i x_k}, f_{x_j x_l}\rangle + \langle f_{x_i x_l}, f_{x_j x_k}\rangle\big)$

Now it remains to compute $f_{x_i x_j}$, remember that : $f = Bx \otimes Ax$ which means:

$$\frac{\partial^2}{\partial x_i x_j} Bx \otimes Ax = B\frac{\partial^2}{\partial x_i x_j} x \otimes Ax = \frac{\partial}{\partial x_i}(Bx_{x_j} \otimes Ax + Bx \otimes Ax_{x_j}) = Bx_{x_j} \otimes Ax_{x_i} + Bx_{x_i} \otimes Ax_{x_j}) \tag{3.33}$$

In this computation we applied the product-rule. We can write this as a multiplication of a sub-block of B with a column of A, which reduces the computation time. We can use this for automatic differentiation. In order to be able to use it, we have to provide some begin state of A and B. By doing this, we also give their sizes (n,m,k), n must be equal to the number of variables but for k there exist some bounds on the minimal size it must have. We will explore this in the following section.

3.5. Bounds on k

As discussed earlier, the number of columns of B, denoted by k, determines the number of squares. However, if we choose k too small, then even though p is a sum-of-squares, it may not be representable by the neural network, making the problem infeasible.

Fortunately, there is an upper bound on the number of squares needed to write a sum-of-squares polynomial as a sum-of-squares, called the Pythagoras bound. Since we look at quartics for the Neural Network, we consider the Pythagoras bound for quartics:

$$\frac{1}{2\sqrt{3}(n^2+3n+1)} - \frac{1}{2} - o(1) \tag{3.34}$$

Now we have all the theory to implement the neural network. But how does this approach compare to other methods?

One of the main advantages is that we eliminate the PSD constraint, but a drawback is that it is possible to get stuck at spurious local minima. In [7], they found that overparametrizing A, B avoids spurious local minima. For the next chapter, we will look at the last method, called the Burer-Monteiro approach, which offers an alternative way to find a sum-of-squares optimization.

4

Burer Monteiro Approach

Another method to find a sum-of-squares decomposition is called the Burer-Monteiro approach. It is a powerful method for large semidefinite programs. It was first introduced by Samuel Burer and Renato Monteiro in [4]. The goal of the Burer-Monteiro approach is to reduce computational complexity. Even though this reformulation is non-convex, under certain conditions, local optima often correspond to global solutions. But how does it work? In the following section, we will first look at how this method finds a sum-of-squares factorization. Then we will take a look at some bounds for the rank of the program. After that, we will look at the special case for univariate polynomials. And, finally, at the end of the chapter, we will look at the gradient of this method. But first, what is the Burer-Monteiro approach?

4.1. The program for Burer-Monteiro

The Burer-Monteiro approach uses the Cholesky decomposition (see theorem 2.2.2). Instead of trying to find a positive semidefinite matrix, it finds the matrix in the decomposition. We can formalize this in the following theorem:

Theorem 4.1.1. $p \in\in \mathbb{R}[x_1,\dots,x_n]_{2d}$ is a SOS $\Leftrightarrow \exists P \in \mathbb{R}^{lxr}$ such that $p = m_d P P^t m_d^T$.

This theorem is similar to theorem 2.2.2, only now P is not necessarily lower triangular.

Proof. (Theorem 4.1.1) If p is a sum of squares (SOS), then there exists a matrix P such that $p = m_d P P^\top m_d^\top$. This follows directly from the Cholesky decomposition (theorem 2.2.2) Conversely, suppose $p = m_d P P^\top m_d^\top$ for some $P \in \mathbb{R}^{l\times r}$. Let $v = P^\top m_d^\top$. Then:

$$p = v^\top v = \sum_{i=1}^{r} v_i^2,$$

where each v_i is a polynomial in the variables $x_1,\dots,x_n$ (since v_i is a linear combination of the monomials in m_d). Thus, it can be concluded that p is a sum of squares polynomial. □

Using this decomposition, we get the following feasibility problem:

$$\begin{aligned} \min \sum_{\alpha\in\mathbb{N}^n s.t. \|\alpha\|=4} (p(x)_\alpha - (m_d P P^T m_d^T)_\alpha)^2 \\ s.t. P \in \mathbb{R}^{l\times r} \end{aligned} \tag{4.1}$$

We used the norm error squared here, also called the l_2 norm for polynomials squared. One of the main advantages of this formulation is that we got rid of the PSD constraint, but can we also reduce the computation by minimizing the rank of P?

4.2. Rank of P

The computation can be reduced if the rank of P is minimized. The choice of this rank influences the existence of spurious local minima. This raises an important question: how large must we choose r, the rank of P, to avoid spurious local minima?

To begin, the rank of B must be at least equal to the Pythagoras bound to ensure that the feasible region contains the solution. We can give an upper bound by taking B to be a square matrix, which means that r = l = $\binom{n+d-1}{d}$. We also know that the total number of constraints is equal to the number of monomials, which for homogeneous polynomials is equal to $\binom{n+2d-1}{2d}$.

In order to avoid spurious local minima at second-order critical points for the general Burer-Monteiro approach, this bound upper bound is strict. There do exist examples were for rank $r = l-1$ there are spurious local minima.

This means that in order to use a lower rank, extra conditions are necessary. If an SDP has a compact search space, it is known that the program admits a global optimum at rank at most $\sqrt{2m}$. [3]. With some additional compactness and smoothness conditions, the Burer-Monteiro approach does not contain spurious local minima for that rank for almost all cost matrices (the measure of the space that does have local minima is 0).

There is also a well-known result, the *Barvinok–Pataki bound*, which states that any semi definite program (SDP) often has an optimal solution of rank at most $O(\sqrt{m})$, where m is the number of affine constraints. [8]

Burer and Monteiro showed that if a linear objective is added to the SDP and its low-rank version the Burer-Monteiro approach, then as long as the rank $r \gtrsim \sqrt{m}$, any local minimum of the Burer-Monteiro approachis also a local minimum of the original SDP with an additional rank-r constraint. They further proved that such a local minimum is either an optimal extreme point, or lies in the relative interior of a face of the SDP feasible region where the objective is constant.

Later work showed that if the cost matrix C is generic, then all local minima of the Burer-Monteiro approach are global minima of the full SDP. These results generally require generic constraints and either the smoothness of the feasible set or a smoothed analysis framework.

However, if the rank r is smaller than $\sqrt{m}$, then local minima of the Burer-Monteiro approach are not guaranteed to be globally optimal.

Now we know all the bounds for the rank of P, but how do they all compare to each other in size? That can be illustrated in the following figure:

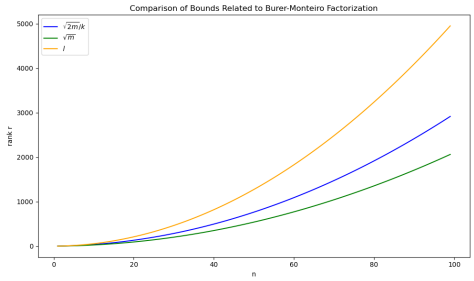


Figure 4.1: Figure: the rank r against the number of variables for the Pythagoras bound,$\sqrt{2m}$, $\sqrt{m}$ and a square matrix (l). It should be noted that $\sqrt{2m}$ is equal to the Pythagoras bound here.

For this case, if there are no constraints on the SDP problem, the rank of r grows quickly. However, there exists a special case where there are no extra constraints needed in order to avoid spurious local minima for the Pythagoras bound. These are the univariate polynomials and we will discuss them in the next section.

4.3. Univariate polynomials

In this section, we will take a look at the proof of the article [8], where they show that for univariate polynomials of $r = 2$, you are guaranteed that every second-order point (SOCP) is a global minimum. In order to have this statement, the objective must be equal to the norm squared error.

Before we can start with the proof, we must first formally define what an SOCP is and define which objective we want to minimize:

Definition 4.3.1. $u \in \mathbb{R}[x]_d^r$ is a second order critical point (SOCP) of $f_p(\mathbf{u})$ if $\nabla f_p(\mathbf{u}) = 0$ and $\nabla^2 f_p(u) \succeq 0$

We define the objective function as follows:

Definition 4.3.2. The objective function is denoted by $f_p(\mathbf{u})$ and given by: $f_p(\mathbf{u}) = \left\| \sum_{i=1}^r (u_i^2 - p) \right\|^2$, where $\mathbf{u} \in \mathbb{R}[x]_d^r$

Now we can finally state the theorem we want to prove

Theorem 4.3.1. For all nonnegative univariate polynomials $p(x) \in R[x]_{2d}$ and any r ≥ 2 if $\mathbf{u} \in R[x]_d^r$ is a SOCP then $f_p(\mathbf{u}) = 0$

We will prove the case for when u_1 and u_2 are co-prime, which they will be generically. In this case, it is enough for $\mathbf{u}$ to be a first-order critical point.

Suppose u_1 and u_2 are c-orpime and the gradient is equal to 0, then we know that this is equal to saying that$\nabla_p(\mathbf{u})(\mathbf{v}) = 0$ for all $\mathbf{v} = (v_1, v_2) \in R[x]^2$. If we compute the directional derivative using the Chain rule we get:

$$\nabla f_p(\mathbf{u})(\mathbf{v}) = 4\left\langle u_1 v_1 + u_2 v_2,\, u_1^2 + u_2^2 - p\right\rangle = 0 \tag{4.2}$$

But now something interesting happens, since u_1, u_2 are co-prime Bezouts identity for polynomials states that there exist v_1' and v_2' such that:

$$u_1 v_1' + u_2 v_2' = u_1^2 + u_2^2 - p \tag{4.3}$$

Since there exist other v_1'' and v_2''' such that the right-hand side is equal to 1. If we fill this right-hand side in for $u_1 v_1' + u_2 v_2'$ in the directional derivative, then we end up with:

$$\nabla f_p(\mathbf{u})(\mathbf{v}') = 4\left\langle u_1^2 + u_2^2 - p,\, u_1^2 + u_2^2 - p\right\rangle = 4 f_p(u) = 0 \tag{4.4}$$

Which finishes the proof. Note that in this case, we only need $\mathbf{u}$ to be an FOCP. However, in order to prove that there are non-spurious local minima for all cases, you will need $\mathbf{u}$ to be a second-order critical point.

Now we know the bounds for the rank of P for univariate polynomials and quartics using the Burer-Monteiro approach, so it is time to compute the gradient. Which we will do in the next section.

4.4. Deriving the gradient

For Burer-Monteiro, we only have to compute the gradient with respect to P. We will do the computation for the quartics. The computation for the gradient of univariate polynomials is similar. We will follow a similar method as for the neural network and write the gradient as a column vector:

$$\nabla_P L(P) = \left[\, \tfrac{\partial L(P)}{\partial P_{11}} \cdots \tfrac{\partial L(P)}{\partial P_{l1}} \cdots \tfrac{\partial L(P)}{P_{lr}} \,\right]^T \tag{4.5}$$

Since we know the loss function, we can now compute the gradient. Which is given by the following equation:

$$\nabla_P L(P) = \nabla_P \sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)^2 \tag{4.6}$$

We will first get rid of the outer square by applying the chain rule:

$$\nabla_P L(P) = -2\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)\nabla_P((m_d P P^T m_d^T)_\alpha) \tag{4.7}$$

We write $m_d P P^T m_d^T$ back to $\left\| P^T m_d^T \right\|$, in order to be able to apply the chain rule later:

$$\nabla_P L(P) = -2\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)\nabla_P(\left\| P^T m_d^T \right\|_\alpha^2) \tag{4.8}$$

Since we know that $\nabla_P L(P) = (D_P L(P))^T$, we can rewrite it to:

$$\nabla_P L(P) = -2\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)(D_P(\left\| P^T m_d^T \right\|_\alpha^2))^T \tag{4.9}$$

Then, after this we can use the Chain Rule to get:

$$\nabla_P L(P) = -4\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)(D_P(P^T m_d^T)^T (P^T m_d^T))_\alpha \tag{4.10}$$

The only unknown in this equation is $D_P(P^T m_d^T)$. So this will be determined first. This can be done by writing this out:

$$D_P(P^T m_d^T) = \begin{bmatrix} P_{11} & \cdots & P_{l1} \\ \vdots & \ddots & \vdots \\ P_{1p} & \cdots & P_{lp} \end{bmatrix} \begin{bmatrix} m_d[1] \\ \vdots \\ m_d[l] \end{bmatrix} = m_d \otimes I_p \tag{4.11}$$

If we put this formula $D_P(P^T m_d^T)$ in the formula of $\nabla_P L(P)$. This results to:

$$\nabla_P L(P) = -4\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)((m_d \otimes I_p)^T (P^T m_d^T))_\alpha \tag{4.12}$$

Working out the transpose results to:

$$\nabla_P L(P) = -4\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)((m_d^T \otimes I_p) P^T m_d^T)_\alpha \tag{4.13}$$

The goal is to write this into a better form:

$$((m_d \otimes I_p)^T (P^T m_d^T)) = (m_d P (m_d \otimes I_p))^T = \begin{bmatrix} m_d P e_1 m_d & \dots & m_d P e_p m_d \end{bmatrix}^T \tag{4.14}$$

Here e_i is just a column vector of only 0's accept for the i^{th} entry. Taking the transpose, this results in, where we write it in matrix form again:

$$((m_d \otimes I_p)^T (P^T m_d^T)) = \begin{bmatrix} (m_d P e_1 m_d)^T & \dots & (m_d P e_p m_d)^T \end{bmatrix} = m_d^T \otimes \begin{bmatrix} (e_1^T P^T m_d^T) & \dots & (e_p^T P^T m_d^T) \end{bmatrix} \tag{4.15}$$

Since these are scalars, the tranpose can be taken out, which results in:

$$m_d^T \otimes \begin{bmatrix} (e_1^T P^T m_d^T) & \dots & (e_p^T P^T m_d^T) \end{bmatrix} = ((m_d \otimes I_p)^T (P^T m_d^T)) = m_d^T \otimes \begin{bmatrix} (m_d P e_1) & \dots & (m_d P e_p) \end{bmatrix} \tag{4.16}$$

This results to:

$$((m_d \otimes I_p)^T (P^T m_d^T)) = m_d^T \otimes \begin{bmatrix} m_d P e_1 & \dots & m_d P e_p \end{bmatrix} = m_d^T \otimes m_d P \tag{4.17}$$

Putting everything together, this becomes:

$$\nabla_P L(P) = -4\sum_\alpha (p_\alpha - (m_d P P^T m_d^T)_\alpha)(m_d^T \otimes m_d)_\alpha P \tag{4.18}$$

The first part of this equation, $m_d P P^T m_d^T$, can be rewritten again, changing it to an inner product with only one part with only monomials and the other with coefficients. When implementing, this will save computations:

$$(m_d P P^T m_d^T) = Tr(m_d P P^T m_d^T) = Tr(m_d^T m_d P P^T) = \langle m_d^T m_d, P P^T \rangle \tag{4.19}$$

The first is a calculation rule, and then the second part is due to the fact that traces are invariant to cycles.

$$\nabla_P L(P) = -4\sum_\alpha (p_\alpha - (\langle m_d^T m_d, P P^T \rangle)_\alpha)(m_d^T \otimes m_d)_\alpha P \tag{4.20}$$

This is the final form for the gradient for Burer-Monteiro for quartics. For the univariate polynomials the gradient is similar with the only change that m_d gets replaced with $[1, x, \dots x^d]$.

We are now ready to implement the different methods in Julia. However, it may also be beneficial to use some form of automatic differentiation for Burer-Monteiro. Since we again have a lot of input variables and only one output variable, backpropagation is likely the most efficient choice.

5

Computational results

After some early testing, we found that the L-BFGS optimizer in Julia outperformed Adam, achieving faster convergence in runtime and number of iterations for target polynomials up to 10 variables. Therefore, all experiments use the L-BFGS optimizer. All the target polynomials were generated via the equation:

$$m_d B B^T m_d^T,$$

where B is a square matrix with entries sampled uniformly from the interval $[-1,1]$. We considered the convergence successful when the objective was less than $1e^{-7}$. During the early testing, some different kinds of automatic differentiation were tested, which showed that the autodifferentiation mode, `autodiff; Autozygote`, outperformed the others in time. Therefore, all experiments with automatic differentiation used this autodifferentiation method. We begin by presenting the results for univariate polynomials.

5.1. Univariate polynomials

The univariate polynomials were tested for $d \in [2,\cdot 20]$, considering only the even degrees (since odd-degree polynomials can never be sum-of-squares polynomials). The tests were performed using different ranks for the matrix B, specifically for $r = 1,2,4$. For each rank and d, we performed 20 runs, resulting in a total of 200 runs for each rank. This set of tests was performed with a maximum of 50 iterations as well as with a maximum of 1000 iterations. This experiment was performed with both automatic and non-automatic differentiation, but there was no notable difference in the number of iterations or in time. The results showed that the rank played a significant role in the number of times that the target polynomial was successfully found and the speed of convergence. These results are summarized in the following table.

r	50 iteraties	1000 iteraties
1	0 %	0 %
2	89.5 %	100 %
4	100 %	100 %

Table 5.1: Percentage of successful runs for $r = 1,2,4$ over 20 runs for $d = 2,\ldots,20$ for both 50 and 1000 iterations, using automatic differentiation.

As shown in the table, the convergence is fastest for $r = 4$, but for $r = 2$, all target polynomials are eventually found. However, $r = 1$ is not guaranteed to converge. The times it converged were all for $d = 2,4$. It might seem like the number of iterations is too low for $r = 1$; however, when the target polynomial was not found, the objective function was plotted. An example of such a plot is shown below:

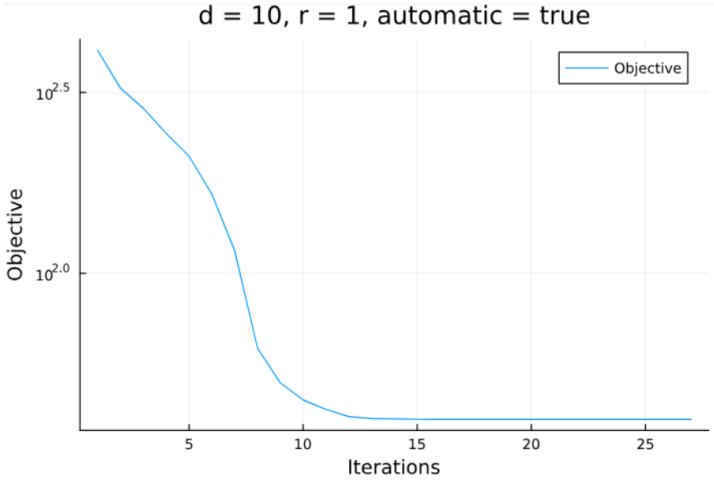


Figure 5.1: The objective for a univariate polynomial with $d = 10$ with the maximum number of iterations equal to 50, using automatic differentiation

As shown in the figure, the objective barely decreases after 10 iterarions. The plot stops early since the optimizer stops when the objective does not decrease anymore. The other plots for $r = 1$ showed a similar pattern.

The code for the univariate polynomials, followed by its tests, is provided in the appendix A. We proceed with the quartic case using the Burer–Monteiro method.

5.2. Burer-Monteiro for quartics

There were two tests performed for the quartics for Burer-Montero. The first evaluated the convergence in iterations using different ranks for P. The second one consisted of comparing automatic and manual differentiation in runtime. We begin by presenting the results for the rank comparison test.

5.2.1. Different ranks of P

The quartics were tested using both automatic and non-automatic differentiation for $n = 2 \dots 10$ with steps of 2. To evaluate convergence, each n was tested 20 times. We used different ranks r, specifically: full rank (l), $\sqrt{2m}$, $\sqrt{m}$, $\frac{3}{4}\sqrt{m}$, $\frac{1}{2}\sqrt{m}$ and n, resulting in 100 runs for each rank. There was no significant difference in the number of iterations required for convergence between automatic and non-automatic differentiation. This test was performed using different limits for the number of iterations: 25,50,75,100, and 1000. Both $r = \frac{1}{2}\sqrt{m}$ and $r = n$ were too low to converge; both of them converged only for $n = 2$ but never converged for larger n, which can be illustrated in the following figures:

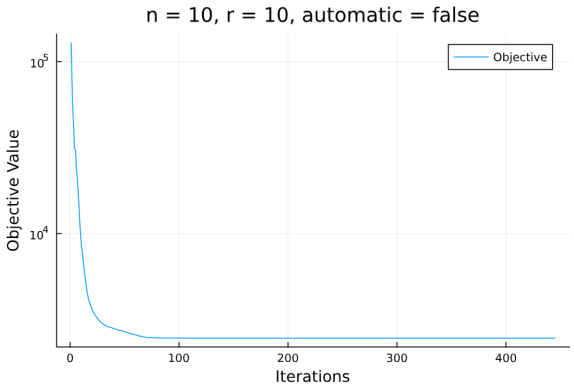


Figure 5.2: The objective for $n = 10$, using $r = 10$, with non-automatic differentiation over 1000 iterations.

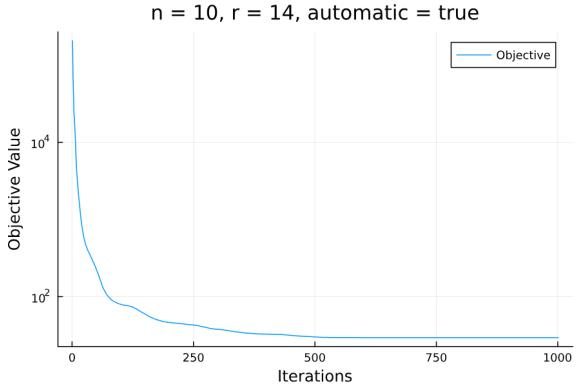


Figure 5.3: The objective for $n = 10$ with $r = \frac{1}{2}\sqrt{m}$, over 1000 iterations.

We can see that $r = n$ gives up earlier and reaches a higher objective value compared to $r = \frac{1}{2}\sqrt{m}$. For $r = n$ it is hard to say whether or not it is a spurious local minimum because it can also be that the global

optimum of the SDP is not in the feasible region, since the rank is below the Pythagoras number itself. The same holds for $r = \frac{1}{2}\sqrt{m}$. Since for $r = \frac{1}{2}\sqrt{m}$ the objective was still decreasing after 1000 iterations, we also tested for a higher number of iterations (2000), but this did not change the objective much.

The other ranks of r; l, $\sqrt{2m}$, $\sqrt{m}$ and $\frac{3}{4}\sqrt{m}$, all convergenced for every run within 100 iterations. However, larger values of r generally led to faster convergence. There was no notable difference in convergence for automatic vs manual differentiation. In the next section, we will explore whether the same can be said about the runtime between the two.

5.2.2. Runtime for automatic and manual differentiation

The second test compared the runtime of automatic and manual differentiation for $n = 10$, for r = (l), $\sqrt{2m}$, $\sqrt{m}$, $\frac{3}{4}\sqrt{m}$, $\frac{1}{2}\sqrt{m}$ and n. There was a notable difference in runtime, for example, for full rank it took 7 seconds for automatic differentiation and 31 for non-automatic differentiation, for example. Also, for the other ranks, automatic differentiation was almost 3x as fast. The code, including its test, is in Appendix B. Now we can take a look at the second approach, namely the neural network.

5.3. Neural network

The neural network followed a similar test to Burer-Monteiro. However, the code for automatic differentiation was too slow (it took over a minute to run the optimizer for $n = 5$ once). Even optimizing the code did not seem to work; the code can be seen in C. Which means the tests are only performed using automatic differentiation. Since we only use automatic differentiation, firstly, a test was performed to control whether or not the loss function was corrected. Following that test we performed two additional experiments; the first compared different values of k and m to check the convergence. The second experiment examined the runtime for automatic differentiation. We will first examine the test that compared the number of iterations.

5.3.1. Different values for k and m

We tested the convergence for different values of k (the Pythagoras number and l), and different values of m ($m = n$ or $m = 2n$). Initially, the idea was to test it with the same number of iterations as with Burer-Monteiro (that is 25,50,75,100 and 1000). However, it turned out that the convergence for the neural network was slower; after 400 iterations, around 70-90 percent had converged; larger values for k and m led to higher percentages. At 1000 iterations, almost all values have converged, which can be illustrated in the following table:

r	succesvolle runs
m = n, k = Pythagoras getal	98 %
m = n, k = l	99 %
m = 2n, k = Pythagoras getal	100 %
m = 2n, k = l	100 %

Table 5.2: Percentage of successful runs after 1000 iterations, for automatic differentiation for the neural network

As shown in the table, only three runs failed to converge successfully. To determine whether these runs got stuck at local minima, we examine their corresponding objective value plots. One of these plots is given below:

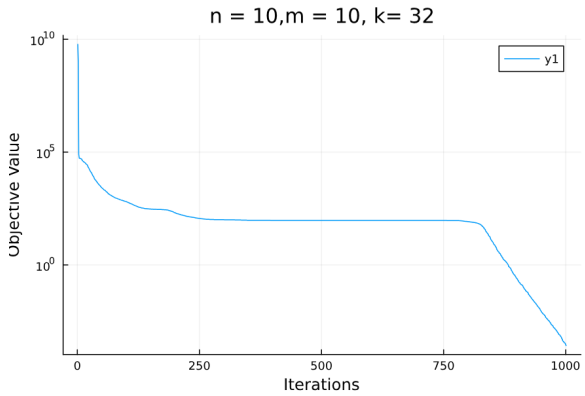


Figure 5.4: The objective for n = 10, m = n, k = Pythagoras number, for automatic differentiation of the neural network

It is interesting to see that the convergence stays for a large number of iterations, and then suddenly it starts dropping again. The other two cases had a similar pattern in their convergence plot. Which denotes that they might have been in a flat region, but in the end, it seems to converge to a global minimum.

5.3.2. The time of the neural network

The second experiment measured the runtime for automatic differentiation for $k = l$/Pythagoras number and $m = 2n/n$ for $n = 10$. This led to a runtime between 17 and 30 seconds (larger values of m/k led to a longer runtime). The code, including its tests, is in the appendix D (for iter = 1000). This is significantly slower than for the Burer-Monteiro approach with automatic differentiation.

6

Discussion and Conclusion

The experimental results support the theoretical expectations regarding the Burer-Monteiro approach for univariate polynomials, as convergence occurs when the rank of r is at least 2. We also saw that this bound was tight. Although $r = 2$ was sufficient, larger r led to faster convergence, suggesting a mild advantage of overparameterization.

In the theory for quartics, the rank $\sqrt{m}$ seems to be large enough for n up to 10, even $0.75\sqrt{m}$ is sufficient for n up to 10, which is lower than the bound we saw in theory. However, for lower values such as $0.5\sqrt{m}$ or $r = n$, convergence consistently fails. Overall, Burer-Monterio converged within 100 iterations, but again, overparameterization seemed to have a slight advantage. Automatic differentiation outperformed manual differentiation.

For the neural network, similar trends were observed; both parameterizing k and m led to faster convergence. Furthermore, using automatic differentiation outperformed manual gradient computations by far in speed. The number of iterations in order to converge was significantly larger than for Burer-Monteiro; some even seemed to get temporarily stuck at flat regions. Non-automatic differentiation failed completely for the neural network.

Overall, Burer-Monteiro seemed to outperform the neural network in both runtime as well as number of iterations. Automatic differentiation also outperformed manual differentiation. However, it is worth noting that LBFGS was used exclusively, and no parameter optimization was performed. This leaves open the possibility that alternative first or second-order methods might lead to different results. Additionally, tests for larger n were limited due to the time of the computations, which could affect the relative performance of the methods. For future work there could be looked at the following things:

- Investigate scalability by looking at larger values of n
- Apply the methods to higher-degree polynomials (creating a neural network for higher-degree polynomials or Burer-Monteiro for higher-degree polynomials).
- Compare against traditional semidefinite programming (SDP) using interior point methods
- Tune the hyperparameters
- Investigate different first-order and second-order methods

Bibliography

[1] Atilim Gunes Baydin et al. “Automatic differentiation in machine learning: a survey”. In: *Journal of machine learning research* 18.153 (2018), pp. 1–43.

[2] Grigoriy Blekherman. “Nonnegative polynomials and sums of squares”. In: *Journal of the American Mathematical Society* 25.3 (2012), pp. 617–635.

[3] Nicolas Boumal, Vlad Voroninski, and Afonso Bandeira. “The non-convex Burer-Monteiro approach works on smooth semidefinite programs”. In: *Advances in Neural Information Processing Systems* 29 (2016).

[4] Samuel Burer and Renato DC Monteiro. “A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization”. In: *Mathematical Programming* 95.2 (2003), pp. 329–357.

[5] Michel X Goemans and David P Williamson. “Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming”. In: *Journal of the ACM (JACM)* 42.6 (1995), pp. 1115–1145.

[6] Paul J Goulart and Sergei Chernyshenko. “Global stability analysis of fluid flows using sum-of-squares”. In: *Physica D: Nonlinear Phenomena* 241.6 (2012), pp. 692–704.

[7] Daniel Keren, Margarita Osadchy, and Roi Poranne. “A practical, fast method for solving sum-of-squares problems for very large polynomials”. In: *arXiv preprint arXiv:2410.19844* (2024).

[8] Benoıt Legat, Chenyang Yuan, and Pablo Parrilo. “Low-rank univariate sum of squares has no spurious local minima”. In: *SIAM Journal on Optimization* 33.3 (2023), pp. 2041–2061.

[9] Michael A. Nielsen. *Neural Networks and Deep Learning*. Online; accessed on July 3, 2025. Determination Press, 2015. URL: `http://neuralnetworksanddeeplearning.com/`.

[10] Victoria Powers. “Positive polynomials and sums of squares: a beginner’s guide”. In: *preprint* ().

A

Code for Burer-Monteiro for univariate Polynomials

A.1. Code for testing

```
using Optim, AbstractAlgebra, LinearAlgebra, SparseArrays, Zygote,
    DifferentiationInterface, Plots
include("Burer_Monteiro_univariate.jl") #include the code

#chapter 3: TESTS
#chapter 0: just some help for the tests
#create the d dictionaires
all_A_dicts = Dict{Int, Dict}()
for d in 1:10
    all_A_dicts[d] = dict_A(d)
end

# a helper function: it makes the plot if the err is too large (to check if it
    is indeed
# a local minimum or just too slow convergence compared to the number of
    iterations : ))
# can also be used to make a convergence plot by setting err_bound to 0 : )
function helper(d,l,r, automatic::Bool, opts, err_bound)
    initial_x = rand(l*r)
    p = creating_p(d)
    p_map = creating_p_map(p,d)
    f_wrapped = x -> f(x, d, l, r, p_map,all_A_dicts[d])
    if automatic
        found = Optim.optimize(f_wrapped, initial_x,LBFGS(),opts; autodiff=
            AutoZygote())
            if Optim.minimum(found)>err_bound
                obj_vals = [obj.value for obj in found.trace]
                display(plot(1:length(obj_vals), obj_vals;xlabel = "Iterations"
                    ,ylabel = "Objective",title = "d = $d, r = $r, automatic =
                    $automatic",yscale = :log10,
    label = "Objective"))
                return 1
            end
        return 0
    end
    #non-automatic part
    g_wrapped = (G,x) -> g!(G,x,d,l,r,p_map, all_A_dicts[d])
    found = Optim.optimize(f_wrapped,g_wrapped, initial_x,LBFGS(),opts)
    if Optim.minimum(found)>err_bound
```

```
        obj_vals = [obj.value for obj in found.trace]
        display(plot(1:length(obj_vals), obj_vals;xlabel = "Iterations",ylabel
            = "Objective",title = "d = $d, r = $r, automatic = $automatic",
            yscale = :log10,
    label = "Objective"))
        return 1
    end
    return 0
end

#chapter 1 tests
#1.1 TEST: how often does it convergece over x amount of runs ?
#1.1 TEST: automatic case
function test_convergence_auto()
    err_bound = 1e-7
    iter = 50
    opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most
        30 000

    #a too low rank
    non_convergence = 0
    for d in 1:10
        l = d+1
        r = 1
        for times in 1:20
            non_convergence+= helper(d,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
         r = 1")

    #for the minimum rank to have no spurious local minima
    non_convergence = 0
    for d in 1:10
        l = d+1
        r = 2
        for times in 1:20
            non_convergence+= helper(d,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
         r = 2")

    non_convergence = 0
    #a larger rank then necessary
    for d in 1:10
        l = d+1
        r = 4
        for times in 1:20
            non_convergence+= helper(d,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
         r = 4")
end



#1.1 TEST: non-automatic case
function test_convergence_non_auto()
    err_bound = 1e-7
```

```
    iter = 1000
    opts = Optim.Options(iterations = iter ,store_trace=true) #it runs at most
        30 000

    #first full rank convergence
    non_convergence = 0
    for d in 1:10
        l = d+1
        r = 1
        for times in 1:20
            non_convergence+= helper(d,l,r, false, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
        r = 1")

    #for the minimum rank to have no spurious local minima
    non_convergence = 0
    for d in 1:10
        l = d+1
        r = 2
        for times in 1:20
            non_convergence+= helper(d,l,r, false, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
        r = 2")

    #a larger rank then necessary
    non_convergence = 0
    for d in 1:10
        l = d+1
        r = 4
        for times in 1:20
            non_convergence+= helper(d,l,r, false, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 200 times, for
        r = 4")
end




println("automatic")
test_convergence_auto()
println("non automatic")
test_convergence_non_auto()
```

A.2. Code for gradient and objective

```
using AbstractAlgebra, LinearAlgebra, Optim

#creates for an n all matrices A, for correct alpha in a dictionary
function dict_A(d)
    #create the dictionary
    A_dict = Dict{Int, Matrix{Float64}}()
    l_dim = d+1

    for i=1:2*d+1
```

```
        #building M
        M = zeros(l_dim, l_dim)
        row = col = 1
        for row= 1:d+1, col = 1:d+1
            if row + col-1 == i
                M[row, col] = 1
            end
        end
        A_dict[i] = M
    end
    return A_dict
end


#creates a "random" polynomial
function creating_p(d)
    R, x_p = polynomial_ring(RDF, 1)
    md = [x_p[1]^i for i in 0:d]
    md_t = reshape(md,1,:)
    B = 2*rand(length(md), length(md)).-1
    p = md_t*B*transpose(B)*md
    return p[1,1]
end

#creating the p_map of the polynomial
function creating_p_map(p,d)
    p_map = Dict{Int, Float64}()
    for i in 1:2*d+1
       p_map[i] = coeff(p, i)
    end
    return p_map
end

#notes:
# in p_map correspondeert 1 met getal 1, en 2 met coeffcient voor x
# in A_dict correspondeert 1 met getal 1 matrix A en 2 met de coefficent voor x


#creates the loss function
function f(x,d,l,r,p_map, A_dict)
    result = 0.0
    P = reshape(x,l,r)
    PPT = P*P'
    for i=1:2*d+1
            result += (p_map[i] - dot(A_dict[i], PPT))^2
    end
    return result
end

#computes the gradient
function g!(G,x,d,l,r,p_map, A_dict)
    P = reshape(x,l,r)
    PPT = P*P'
    result = zeros(size(P)...)
    for i=1:2*d+1
            result += (p_map[i] - dot(A_dict[i], PPT)) * A_dict[i] * P
    end
    G[1:l*r] = -4 * reshape(result,:)
end
```

B

Code for Burer-Monteiro for quartics

B.1. Normal code

```
   sing AbstractAlgebra, LinearAlgebra, Optim

#creates for an n all matrices A, for correct alpha in a dictionary
function dict_A(n)
    #create the dictionary
    A_dict = Dict{NTuple{4, Int}, Matrix{Float64}}()
    l_dim = binomial(n+1, 2)

    for i=1:n, j=i:n, k=j:n, l=k:n
        alpha = zeros(Int, n)
        alpha[i] += 1
        alpha[j] += 1
        alpha[k] += 1
        alpha[l] += 1

        #building M
        M = zeros(l_dim, l_dim)
        row = col = 1
        for r1 = 1:n, r2 = r1:n
            beta = zeros(Int, n)
            beta[r1] += 1
            beta[r2] += 1
            col = 1
            for c1 = 1:n, c2 = c1:n
                gamma = zeros(Int, n)
                gamma[c1] += 1
                gamma[c2] += 1

                if beta + gamma == alpha
                    M[row, col] = 1
                end

                col += 1
            end
            row += 1

        #set it to the dict
        end
        #we don't want alpha but i,j,k,l
        A_dict[(i,j,k,l)] = M
    end
```

```
    return A_dict
end

#creates a "random" polynomial
function creating_p(n)
    R, x_p = polynomial_ring(RDF, n)
    md =  Vector{typeof(x_p[1])}(undef, Int(length(x_p)*(length(x_p)+1)    2))
        #column vector
    idx = 1
    for i in 1:length(x_p)
        for j in i:length(x_p)
            md[idx] = x_p[i]*x_p[j]
            idx += 1
        end
    end
    md_t = reshape(md,1,:)
    B = 2*rand(length(md), length(md)).-1
    p = md_t*B*transpose(B)*md
    return p[1,1]
end

#creating the p_map of the polynomial
function creating_p_map(p,n)
    p_map = Dict{NTuple{4, Int}, Float64}()
    alpha = zeros(Int, n)
    for i in 1:n, j in i:n, k in j:n, l in k:n
        fill!(alpha, 0) #it said this for less allocations
        alpha[i] += 1
        alpha[j] += 1
        alpha[k] += 1
        alpha[l] += 1
        p_map[(i, j, k, l)] = coeff(p, alpha)
    end
    return p_map
end


#creates the loss function
function f(x,n,l,r,p_map, A_dict)
    result = 0.0
    P = reshape(x,l,r)
    PPT = P*P'
    for i=1:n, j=i:n, k=j:n, l=k:n
            result += (p_map[i,j,k,l] - dot(A_dict[i,j,k,l], PPT))^2
    end
    return result
end

#computes the gradient

function g!(G,x,n,l,r,p_map, A_dict)
    P = reshape(x,l,r)
    PPT = P*P'
    result = zeros(size(P)...)
    for i=1:n, j=i:n, k=j:n, l=k:n
            result += (p_map[i,j,k,l] - dot(A_dict[i,j,k,l], PPT)) * A_dict[i,j
                ,k,l] * P
    end
    G[1:l*r] = -4 * reshape(result,:)
end
```

B.2. Code for testing

```
    using Optim, AbstractAlgebra, LinearAlgebra, SparseArrays, Zygote,
        DifferentiationInterface, Plots
include("Burer_Monteiro.jl") #include the code


#chapter 0: just some help for the tests
#create the alpha dictionaires
all_A_dicts = Dict{Int, Dict}()
for n in 1:10
    all_A_dicts[n] = dict_A(n)
end

# a helper function: it makes the plot if the err is too large (to check if it
    is indeed
# a local minimum or just too slow convergence compared to the number of
    iterations : ))
# can also be used to make a convergence plot by setting err_bound to 0 : )
function helper(n,l,r, automatic::Bool, opts, err_bound)
    initial_x = rand(l*r)
    p = creating_p(n)
    p_map = creating_p_map(p,n)
    f_wrapped = x -> f(x, n, l, r, p_map,all_A_dicts[n])
    if automatic
        found = Optim.optimize(f_wrapped, initial_x,LBFGS(),opts; autodiff=
            AutoZygote())
            if Optim.minimum(found)>err_bound
                obj_vals = [obj.value for obj in found.trace]
                display(plot(1:length(obj_vals), obj_vals;xlabel = "Iterations"
                    ,ylabel = "Objective Value",title = "n = $n, r = $r,
                    automatic = $automatic",yscale = :log10, label = "Objective
                    "))
                return 1
            end
        return 0
    end
    #hier for non automatic , cechk daarna de functie en daarna aanpssen in
        cdoe
    g_wrapped = (G,x) -> g!(G,x,n,l,r,p_map, all_A_dicts[n])
    found = Optim.optimize(f_wrapped,g_wrapped, initial_x,LBFGS(),opts)
    if Optim.minimum(found)>err_bound
        obj_vals = [obj.value for obj in found.trace]
        display(plot(1:length(obj_vals), obj_vals;xlabel = "Iterations",ylabel
            = "Objective Value",title = "n = $n, r = $r, automatic = $automatic
            ",yscale = :log10, label = "Objective"))
        return 1
    end
    return 0
end


#chapter 1 tests
#1.1 TEST: how often does it convergece over x amount of runs ?
#1.1 TEST: automatic case
function test_convergence_auto()
    err_bound = 1e-7
    iter = 1000
    opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most
        30 000

```

```
    #first full rank convergence
    non_convergence = 0
    for n in 2:2:10
        l = Int(n*(n+1)/2)
        r = l
        for times in 1:20
            non_convergence+= helper(n,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")

    #for the minimum rank to have generic cases convergence (sqrt(2m)/k)
    non_convergence = 0
    for n in 2:2:10
        l = Int(n*(n+1)/2)
        m = binomial(n+3, 4)
        r = Int(ceil(sqrt(2*m)))
        for times in 1:20
            non_convergence+= helper(n,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")

    #(sqrt(m))
    non_convergence = 0
    for n in 2:2:10
        l = Int(n*(n+1)/2)
        m = binomial(n+3, 4)
        r = Int(ceil(sqrt(m)))
        non_convergence = 0
        for times in 1:20
            non_convergence+= helper(n,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")

    #3/4(sqrt(m))
    non_convergence = 0
    for n in 2:2:10
        l = Int(n*(n+1)/2)
        m = binomial(n+3, 4)
        r = Int(ceil(0.75*sqrt(m)))
        for times in 1:20
            non_convergence+= helper(n,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")

    #far below the minimum rank to have generic cases convergence namely r = n
    non_convergence = 0
    for n in 2:2:10
        l = Int(n*(n+1)/2)
        r = n
        for times in 1:20
            non_convergence+= helper(n,l,r, true, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")
end
```

```
110
111
112 #1.1 TEST: non-automatic case
113 function test_convergence_non_auto()
114     err_bound = 1e-7
115     iter = 1000
116     opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most
        30 000
117
118     #first full rank convergence
119     non_convergence = 0
120     for n in 2:2:10
121         l = Int(n*(n+1)/2)
122         r = l
123         for times in 1:20
124             non_convergence+= helper(n,l,r, false, opts, err_bound)
125         end
126     end
127     println("The convergence failed ", non_convergence, " out of 100 times")
128
129     #(sqrt(2m))
130     non_convergence = 0
131     for n in 2:2:10
132         l = Int(n*(n+1)/2)
133         m = binomial(n+3, 4)
134         r = Int(ceil(sqrt(2*m)))
135         for times in 1:20
136             non_convergence+= helper(n,l,r, false, opts, err_bound)
137         end
138     end
139     println("The convergence failed ", non_convergence, " out of 100 times")
140
141     #(sqrt(m))
142     non_convergence = 0
143     for n in 2:2:10
144         l = Int(n*(n+1)/2)
145         m = binomial(n+3, 4)
146         r = Int(ceil(sqrt(m)))
147         for times in 1:20
148             non_convergence+= helper(n,l,r, false, opts, err_bound)
149         end
150     end
151     println("The convergence failed ", non_convergence, " out of 100 times")
152
153     #3/4(sqrt(m))
154     non_convergence = 0
155     for n in 2:2:10
156         l = Int(n*(n+1)/2)
157         m = binomial(n+3, 4)
158         r = Int(ceil(0.75*sqrt(m)))
159         for times in 1:20
160             non_convergence+= helper(n,l,r, false, opts, err_bound)
161         end
162     end
163     println("The convergence failed ", non_convergence, " out of 100 times")
164
165     #far below the minimum rank to have generic cases convergence, r = n
166     non_convergence = 0
167     for n in 2:2:10
168         l = Int(n*(n+1)/2)
169         r = n
```

```
        for times in 1:20
            non_convergence+= helper(n,l,r, false, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")
end


# a test for time for both automatic and non-automatic, but first we see if there is a difference in
#convergence. we test until n= 10 that is easier in terms of time consuming
#we take this m (do we still see convergence everywhere? then we take a larger one )
function test_time()
    n = 10
    m = binomial(n+3, 4)
    l = Int(n*(n+1)/2)
    err_bound = 1e-7
    iter = 1000
    opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most 30 000
    p = creating_p(n)
    p_map = creating_p_map(p,n)
    for r in [l, k, Int(ceil(sqrt(2*m))), Int(ceil(sqrt(m))),k = Int(3/4*sqrt(m)),n]
        initial_x = rand(l*r)
        f_wrapped = x -> f(x, n, l, r, p_map,all_A_dicts[n])
        g_wrapped = (G,x) -> g!(G,x,n,l,r,p_map, all_A_dicts[n])
        println("time of non-automatic is with rank: ", r)
        @time found = Optim.optimize(f_wrapped,g_wrapped, initial_x,LBFGS(), opts)
        println("time of automatic is with rank: ", r)
        @time found2 = Optim.optimize(f_wrapped, initial_x,LBFGS(),opts; autodiff=AutoZygote())
    end
end
```

C

Code for neural network using manual differentiation

C.1. Normal code

```
    using Optim, AbstractAlgebra, LinearAlgebra, SparseArrays, LineSearches


#create p
function creating_p(n)
    R, x_p = polynomial_ring(RDF, n)
    md =  Vector{typeof(x_p[1])}(undef, Int(length(x_p)*(length(x_p)+1)    2))
        #column vector
    idx = 1
    for i in 1:length(x_p)
        for j in i:length(x_p)
            md[idx] = x_p[i]*x_p[j]
            idx += 1
        end
    end
    md_t = reshape(md,1,:)
    B = 2*rand(length(md), length(md)).-1
    p = md_t*B*transpose(B)*md
    return p[1,1]
end


#loss functie
function f(x, p,m,n,k)
    R, x_p = polynomial_ring(RDF, n)
    result = 0.0
    A = transpose(reshape(x[1:m*n], n, m))
    B = transpose(reshape(x[m*n+1:(k+1)*m*n], m*n, k))


    v = kron(x_p, A*x_p)
    g = sum((B * v).^2)
    difference = p-g
    for c in coefficients(difference)
        result += c^2
    end
    return result
end

```

```
39
40
41
42
43 #gradient
44 function g!(G, x,p,m,n,k)
45     #we alter the G and we eveluate G at x
46     R, x_p = polynomial_ring(RDF, n)
47     iden = Matrix{Float64}(I, m, m)
48     A = transpose(reshape(x[1:m*n], n, m))
49     B = transpose(reshape(x[m*n+1:(k+1)*m*n], m*n, k))
50
51     Axp = A*x_p
52
53     q = kron(transpose(x_p), iden, x_p) * B' * B * kron(x_p, Axp)
54     r = kron(transpose(x_p), transpose(x_p)*transpose(A)) * B' * B * kron(x_p,
           Axp)
55
56     for i = 1:n*m
57         G[i] = 0.0 #first make it empty to add again
58         for alpha in exponent_vectors(q[i])
59             G[i] += -4*(coeff(p, alpha) - coeff(r, alpha)) * coeff(q[i], alpha)
60         end
61     end
62
63   #   q = kron(iden, x_p, A*x_p) * B * kron(x_p, A*x_p)
64     xpAxp = kron(x_p, Axp)
65     c = B * xpAxp
66     for i = 1:k*m*n
67         G[i+m*n] = 0.0 #first make it empty to add again
68         ik, imn = divrem(i-1, m*n)
69         ik += 1
70         imn += 1
71         qi = xpAxp[imn] * c[ik]
72         for alpha in exponent_vectors(qi)
73             G[i+m*n] += -4*(coeff(p, alpha) - coeff(r, alpha)) * coeff(qi,
                   alpha)
74         end
75     end
76 end
```

D

Code for neural network using automatic differentiation

D.1. Normal code

```
using Optim, AbstractAlgebra, LinearAlgebra, SparseArrays, Zygote,
    DifferentiationInterface

#deze functie maakt het polynoom aan
function creating_p(n)
    R, x_p = polynomial_ring(RDF, n)
    md =  Vector{typeof(x_p[1])}(undef, Int(length(x_p)*(length(x_p)+1)    2))
        #column vector
    idx = 1
    for i in 1:length(x_p)
        for j in i:length(x_p)
            md[idx] = x_p[i]*x_p[j]
            idx += 1
        end
    end
    md_t = reshape(md,1,:)
    B = 2*rand(length(md), length(md)).-1
    p = md_t*B*transpose(B)*md
    return p[1,1]
end

#deze functie zet de functies in een map
function creating_p_map(p,n)
    p_map = Dict{NTuple{4, Int}, Float64}()
    alpha = zeros(Int, n)
    for i in 1:n, j in i:n, k in j:n, l in k:n
        fill!(alpha, 0) #it said this for less allocations
        alpha[i] += 1
        alpha[j] += 1
        alpha[k] += 1
        alpha[l] += 1
        p_map[(i, j, k, l)] = coeff(p, alpha)
    end
    return p_map
end

#dit is gewoon de loss functie
function f(x, p_map, m,n,k)
    # dit maakt een A,B lijst aan, voor straks voor de f'jes
```

```
    result = zero(typeof(first(x)))
    A = transpose(reshape(x[1:m*n], n, m))
    B = transpose(reshape(x[m*n+1:(k+1)*m*n], m*n, k))
    A_list = [A[:, i] for i in 1:n]
    B_list = [B[:, (i-1)*m+1:i*m] for i in 1:n]

    #dit maakt een map aan met alle f'jes, in de hoop dat dit sneller was
    f_map = Dict{Tuple{Int, Int}, Vector{typeof(first(x))}}()
    for i in 1:n, j in i:n
        f_map[(i, j)] = B_list[i]*A_list[j]+B_list[j]*A_list[i]
    end

    #deze laat alle coefficienten voor de monomials uitrekenen (case splitten)
    for i in 1:n, j in i:n, k in j:n, l in k:n
        if i== l #4 case (note l>=k>=j>=i)
            result += (p_map[(i, j, k, l)] -1/4*dot(f_map[(i,i)], f_map[(i,i)])
                )^2
        elseif i == j && k == l #2,2 case (i 2x, k 2x)
            result += (p_map[(i, j, k, l)] -dot(f_map[(i,k)],f_map[(i,k)])-0.5*
                dot(f_map[(i,i)],f_map[(k,k)]))^2
        elseif i ==k #3,1 case (i 3x, l 1x)
            result += (p_map[(i, j, k, l)] -dot(f_map[(i,i)],f_map[(i,l)]))^2
        elseif j == l #3,1 case (i 1x, j 3x)
            result += (p_map[(i, j, k, l)] -dot(f_map[(j,j)], f_map[(i,j)]))^2
        elseif i ==j #2,1,1 case (i 2x, k 1x, l 1x)
            result += (p_map[(i, j, k, l)] -dot(f_map[(i,i)],f_map[(k,l)])-2*
                dot(f_map[(i,l)],f_map[(i,k)]))^2
        elseif j == k #2,1,1 case (i 1x, j 2x, l 1x)
            result += (p_map[(i, j, k, l)]-dot(f_map[(j,j)],f_map[(i,l)])-2*dot
                (f_map[(i,j)],f_map[(j,l)]) )^2
        elseif k == l  #case 1,1,2 (i 1x, j 1x, k 2x)
            result += (p_map[(i, j, k, l)]-dot(f_map[(k,k)],f_map[(i,j)])-2*dot
                (f_map[(j,k)],f_map[(i,k)]))^2
        else #1,1,1,1 case (i 1x, j 1x, k 1x, l 1x)
            result += (p_map[(i, j, k, l)]-2*(dot(f_map[(i,j)], f_map[(k,l)])+
                dot(f_map[(i,k)], f_map[(j,l)])+dot(f_map[(i,l)],f_map[(j,k)]))
                )^2
        end
    end
    return result
end
```

D.2. Tests

```
   using Optim, AbstractAlgebra, LinearAlgebra, SparseArrays, Zygote,
      DifferentiationInterface, Plots
include("Neural_Network_automatic.jl") #include the code

#TEST 0
#test die kijkt of je daadwerkelijk het goede polynoom terug krijgt
#de objective gevonden door de optmizer en de objective gevonden
# bij norm squared error uitrekenen uit de hand
function test_rightpolynomial()
    n = 2
    m = n
    k = Int(n*(n+1)/2)
    initial_x = rand(m*n+k*m*n)
    p = creating_p(n)
    p_map = creating_p_map(p,n)

```

```
    #de optimizer uitvoeren
    found = Optim.optimize(x -> f(x, p_map, m, n,k), initial_x, Adam())
    optimal_x = Optim.minimizer(found)
    A = transpose(reshape(optimal_x[1:m*n], n, m))
    B = transpose(reshape(optimal_x[m*n+1:(k+1)*m*n], m*n, k))
    R, x_p = polynomial_ring(RDF, n)
    p_found =transpose(B*kron(x_p,A*x_p))*B*kron(x_p,A*x_p)
    #opzetten van de check
    g = p-p_found
    count = 0
    for alpha in exponent_vectors(g)
        count +=coeff(g, alpha)^2
    end

    #uitvoeren van de check
    objective_value = f(optimal_x, p_map, m, n,k)
    println("Final objective value: ", objective_value)
    print("The final objective check = ", count)
end

# a helper function: it makes the plot if the err is too large (to check if it is indeed
# a local minimum or just too slow convergence compared to the number of iterations : ))
# can also be used to make a convergence plot by setting err_bound to 0 : )
function helper(n,m,k,opts, err_bound)
    initial_x = rand(m*n+k*m*n)
    p = creating_p(n)
    p_map = creating_p_map(p,n)
    f_wrapped = x -> f(x, p_map, m,n,k)
    found = Optim.optimize(f_wrapped, initial_x,LBFGS(),opts; autodiff=AutoZygote())
        if Optim.minimum(found)>err_bound
            obj_vals = [obj.value for obj in found.trace]
            display(plot(1:length(obj_vals), obj_vals;xlabel = "Iterations",
                ylabel = "Objective Value",title = "n = $n,m = $m, k= $k",
                yscale = :log10))
            return 1
        end
    return 0
end


#chapter 1 tests
#1.1 TEST: how often does it convergece over x amount of runs ?
#1.1 TEST: automatic case
function test_convergence_auto()
    err_bound = 1e-7
    iter = 1000
    opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most 30 000

    #m =n, k pythagoras number
    non_convergence = 0
    for n in 2:2:10
        m = n
        k = Int(ceil(1/(2*sqrt(3))*(n^2+n+1)-0.5))
        for times in 1:20
            non_convergence+= helper(n,m,k, opts, err_bound)
        end
    end
```

```
    println("The convergence failed ", non_convergence, " out of 100 times")


    #m =n, k = l
    non_convergence = 0
    for n in 2:2:10
        m = n
        k = Int(n*(n+1)/2)
        for times in 1:20
            non_convergence+= helper(n,m,k, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")


    #m = 2n, k = pythagoras number
    non_convergence = 0
    for n in 2:2:10
        m = 2*n
        k = Int(ceil(1/(2*sqrt(3))*(n^2+n+1)-0.5))
        for times in 1:20
            non_convergence+= helper(n,m,k, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")

    #m = 2n, k = l
    non_convergence = 0
    for n in 2:2:10
        m = 2*n
        k = Int(n*(n+1)/2)
        for times in 1:20
            non_convergence+= helper(n,m,k, opts, err_bound)
        end
    end
    println("The convergence failed ", non_convergence, " out of 100 times")
end



function test_time()
    n = 10
    l = Int(n*(n+1)/2)
    err_bound = 1e-7
    iter = 1000
    opts = Optim.Options(iterations = iter,store_trace=true) #it runs at most
        30 000
    p = creating_p(n)
    p_map = creating_p_map(p,n)
    for k in [Int(ceil(1/(2*sqrt(3))*(n^2 + 3*n + 1) - 0.5)), l]
        for m in [n, 2*n]
            initial_x = rand(m*n+k*m*n)
            f_wrapped = x -> f(x, p_map, m,n,k)
            println("time of automatic is with k = : ", " ", k, " and m is ","
                ", m)
            @time found = Optim.optimize(f_wrapped, initial_x,LBFGS(),opts;
                autodiff=AutoZygote())
        end
    end
end

```

```
129 test_time()
```