universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Copositivity Testing: A novel decomposition procedure for arbitrary matrices and an investigation of gradient-based search algorithms for finding violating vectors“

verfasst von / submitted by

Johannes Zischg, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 977
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Business Analytics
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Immanuel Bomze

Abstract

1 Abstract

This thesis explores the possibility of enhancing existing copositivity tests by decomposing a given matrix in a novel way. The resulting algorithm can process any given matrix and offers a sufficient condition for copositivity. Also, this paper investigates gradient-based methods for a quick and efficient search for a violating vector that can be applied to any symmetric matrix of arbitrary order without needing any preprocessing steps.

2 Kurzfassung

Diese Arbeit befasst sich mit der Problemstellung, existierende Copositivititätstests mittels eines neuen Zerlegungsverfahrens besser anwendbar zu machen. Der resultierende Algorithmus ist in der Lage, jede beliebige Matrix zu zerlegen und bietet eine hinreichende Bedingung für den Nachweis von Copositivität einer Matrix. Darüberhinaus beschäftigt sich diese Arbeit mit Gradientensuchverfahren zur schnellen und unkomplizierten Identifizierung von Vektoren, die die Copositivitätseigenschaft einer Matrix widerlegen. Diese Verfahren können auf jede symmetrische Matrix beliebiger Ordnung angewandt werden, ohne Vorbearbeitungsschritte durchführen zu müssen.

Contents

List of Tables

List of Figures

Listings

1 Copositivity

1.1 Motivation

Copositivity is a well-studied research field, dating back to 1952 when it came up in [Mot52]. From then on, numerous results have been achieved in this field, from general results to concrete algorithms to prove this essential property to links to several other fields, stressing the importance of copositivity even more. It has numerous applications, such as optimization theory, max-clique determination, theoretical economics, etc. An overview of the various application fields can be found, e.g., in [HS10].
It has been shown that testing a matrix for copositivity is a co-NP-complete problem [Dic19]. Hence, as long as P≠NP holds, no general fast algorithm can give a definitive answer to this question in polynomial time for an arbitrary matrix. Due to this fact, it is an active field of research to find good algorithms for specific types of matrices (like for tri-[Bom00] or pentadiagonal [Bre22] and acyclic matrices [Ikr02]).
Another branch of research is concerned with trying to find faster methods for the general case. While there are existing algorithms that can be applied to any matrix of arbitrary order, even the fastest algorithms (one of which is described in [Kap00]) become less and less valuable if the order of the matrix increases [ZD11].
The question that naturally arises is whether there are fast, easily deployable solutions that can be applied even to larger matrices in a reasonable time or if there is an option to decrease the matrix size, hence reducing computational complexity in the process.

1.2 Introduction

To increase the speed of checking for copositivity of a matrix, there are essentially two ways to go:
One can either develop a faster algorithm to derive a result in less time or reduce the size of the matrix in question, thereby 'overcoming' the exponential increase in run time that comes along with (co)-NP-hard problems. This thesis aims at doing both. First, a novel approach for splitting up a given matrix in a way that multiple but smaller matrices have to be verified to assess copositivity of a matrix is introduced.
Since, to the author's surprise, no detailed study of the use of gradient-based methods for determining copositivity (or, more precisely, to refute the same) are available, different approaches for such algorithms are presented together with experimental results on different kinds of matrices.
This thesis is organized in the following way:
In the beginning, a theoretical foundation about general results for copositivity is laid,

then the main result of this paper in the form of the matrix decomposition algorithm is presented, and the different cases that can arise in this process are treated both from a theoretical as well as from a practical perspective. After that, several gradient-based approaches are discussed with their up- and downsides.
In the experiments section, numerical trials using the gradient-based algorithms are discussed, carried out on random matrices and the DIMACS data set.
Lastly, the results of this paper are summarized, and an outlook for potential future work is given.

1.3 Copositivity - a theoretical foundation

Before diving into the properties of copositive matrices, we start by defining what copositivity even means:

Definition 1.3.1. *[Mot52][Dia62][HN63]: Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix. A is considered copositive if its associated quadratic form $x^T Ax$, $x \in \mathbb{R}^n$ takes only nonnegative values on the nonnegative orthant $\mathbb{R}^n_+$.*

This can also be rephrased to the following :

Definition 1.3.2. *[HS10]: For a $n \times n$ matrix A define*

$$\mu(A) \coloneqq \min_{x\geq 0, \|x\|=1} x^T Ax$$

where $\|.\|$ is any norm.

Proposition 1.3.3. *[HS10] Let A be a real, symmetric matrix. Then the following are equivalent:*

- *A is copositive*
- *$\mu(A) \geq 0$*

Proof. The proof is very similar to that of Lemma 1.3.5 below. □

Definition 1.3.4. *A vector $x \in \mathbb{R}^n_+$ is called a violating vector for the copositivity of a matrix $A \in \mathbb{R}^{n\times n}$ if*

$$x^T Ax < 0$$

i.e., the matrix A is not copositive because the vector x is a violating counterexample.

One difference that immediately becomes apparent is that in Definition 1.3.2 and in Proposition 1.3.3, only vectors x with unit length are considered. The fact that this is indeed a valid equivalence becomes immediately clear with the proof of the next result.

Lemma 1.3.5. *If a vector $x \in \mathbb{R}^n_+$ is a violating vector, so is every multiple of x.*

Proof. If $x^T Ax < 0$ for some x and $\lambda \in \mathbb{R}$ then we have

$$x^T Ax < 0 \Leftrightarrow$$

$$(\lambda^2) x^T Ax < 0 \Leftrightarrow$$

$$(\lambda x)^T A(\lambda x) < 0.$$

□

Remark 1.3.6. *Note that if $\lambda < 0$, the resulting vector x will lie in $\mathbb{R}^n_-$.*

Since multiplication with a constant does not change anything about the fact that a violating vector stays a violating vector, the same holds for normalizing a vector to unit length (which is also just a multiplication with a constant).

One of the essential properties of copositive matrices is that copositivity also holds for the principal submatrices - the following result will be used repeatedly in later chapters and is, therefore, especially important.

Theorem 1.3.7. *[BSU12]: If $A \in \mathbb{R}^{n\times n}$ is a copositive, symmetric matrix, then also every principal submatrix and every permutation similar matrix $P^T AP$ (with P a permutation matrix) is again copositive.*

Remark 1.3.8. *A principal submatrix is part of an original matrix, where arbitrary column/row indices are chosen (column and row indices have to be the same). Only the values at these indices are used for the submatrix.*

Another view on the principal-submatrix-property is given in the following proposition.

Proposition 1.3.9. *[HS10]: If $A \in \mathbb{R}^{n\times n}$ is a copositive matrix, then each principal submatrix of order $n-1$ is also copositive.*

Remark 1.3.10. *It is important to note that this property can be applied recursively - meaning that if a matrix A is copositive, every principal submatrix of any order $< n$ is copositive as well.*

The other direction does not hold, so it is insufficient to only check all principal submatrices for copositivity.

Theorem 1.3.11. *[HS10]: Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix. If every principal submatrix of A of order $n-1$ is copositive, the following two equivalences hold:*

$$A \text{ is not copositive} \Leftrightarrow A^{-1} \text{ exists and is nonpositive elementwise}$$

and

$$A \text{ is copositive} \Leftrightarrow \det A \geq 0 \text{ or } adjA \text{ contains a negative entry.}$$

Proof. A proof of the first equivalence can be found in [Had83]. For the second one, consider [CHL67],[CHL70]. □

Remark 1.3.12. *The adjugate matrix adjA is defined as the transpose of the matrix of cofactors of A, which can be computed from determinants of submatrices of A. This matter is not discussed further in detail in this thesis. The interested reader can find more information in [HJ13].*

Lastly, a result that will come in handy later is the following proposition about nonnegative matrices in the context of verifying copositivity of a given matrix.

Proposition 1.3.13. *If $A \in \mathbb{R}^{n\times n}$ is a nonnegative, symmetric matrix, it is also copositive.*

Proof. The proof is trivial since x is always a nonnegative vector in the context of copositivity, and A is also nonnegative. Hence $x^T Ax$ has to be nonnegative in any case as well. □

These are some first results that give a hint of why checking for copositivity is such a hard problem. To verify that a matrix is copositive, naively, one would have to check every single principal submatrix, and even then, copositivity might not be given if the determinant of A was negative.

A fascinating aspect that will also pop up again later while conducting the numerical experiments is the link to the max-clique problem.

In [KP02] it has been shown that the clique-number $\gamma(G)$ for a graph $G(V, E)$ with a set of edges E and a set of vertices V can be determined by calculating

$$\gamma(G) = \min\{\lambda \in \mathbb{N} : \lambda(I - B) - I \text{ is copositive}\}$$

where B is the adjacency matrix of G and I is the all-one matrix (i.e. every entry is 1) of fitting size. If exact calculation is not possible - since many instances of very dense graphs cause numerical problems while evaluating copositivity - it is still possible to derive upper and lower bounds for the size of the maximum clique.

1.4 Proving copositivity for matrices of order $n \leq 7$

Proving that a matrix is indeed copositive becomes very hard in larger dimensions - after all, this is a co-NP-complete problem. But there are closed-form solutions for some kinds of matrices.

Proposition 1.4.1. *[Had83]: Let $A \in \mathbb{R}^{2\times 2}$ be a symmetric matrix. It is copositive if and only if*

$$a_{1,1} \geq 0, a_{2,2} \geq 0$$

and

$$a_{1,2} + \sqrt{a_{1,1}a_{2,2}} \geq 0.$$

Proposition 1.4.2. *[Had83]: Let $A \in \mathbb{R}^{3\times 3}$ be a symmetric matrix. It is copositive if and only if*

$$a_{1,1} \geq 0, a_{2,2} \geq 0, a_{3,3} \geq 0$$

$$\bar{a}_{1,2} = a_{1,2} + \sqrt{a_{1,1}a_{2,2}} \geq 0$$

$$\bar{a}_{1,3} = a_{1,3} + \sqrt{a_{1,1}a_{3,3}} \geq 0$$

$$\bar{a}_{2,3} = a_{2,3} + \sqrt{a_{2,2}a_{3,3}} \geq 0$$

and

$$\sqrt{a_{1,1}a_{2,2}a_{3,3}} + a_{1,2}\sqrt{a_{3,3}} + a_{1,3}\sqrt{a_{2,2}} + a_{2,3}\sqrt{a_{1,1}} + \sqrt{2\bar{a}_{1,2}\bar{a}_{1,3}\bar{a}_{2,3}} \geq 0.$$

Further investigations on an algorithmic determination whether a given matrix of order $n \leq 7$ is copositive or not can be found in [YL09] - beyond that, there are no algorithmic solutions to the author's knowledge that don't either focus on an investigation on single rows/columns or require some recursive procedure.

1.5 Necessary and sufficient conditions for copositivity

For matrices larger than order 7, finding a general algorithm or a sufficient, easy-to-check condition to prove copositivity of any given matrix A becomes increasingly. Instead, over time many results have been found that allow a statement about the copositivity of A in (more or less) specific cases. This section collects some of these results and aims to indicate conditions that can be checked to verify or refute copositivity.

Corollary 1.5.1. *[Mur88]: If all off-diagonal elements of a real, symmetric matrix A are nonpositive, then A is copositive if and only if A is positive semidefinite.*

It is clear that a positive semidefinite matrix is always copositive, so one way of checking for copositivity is to verify whether a matrix A is positive semidefinite or not - while stressing that it is a sufficient but not a necessary condition for copositivity! One way of approaching this is to look at the eigenvalues of A.

Proposition 1.5.2. *[HS10]: Let A be a real, symmetric matrix.*

- *If A is copositive, at least one of the eigenvalues of A is nonnegative and the sum of the eigenvalues of A (counting multiplicity) is also nonnegative.*

- *If all eigenvalues of A are nonnegative, A is positive semidefinite, and therefore also copositive.*

It does not stop there. Copositive matrices have additional properties concerning their eigenvalues - and their spectral radii.

Definition 1.5.3. *[Bom08]: The spectral radius $\rho(A)$ of a matrix A is the maximum of the absolute values of all eigenvalues of A.*

Theorem 1.5.4. *[Bom08]: If A is a copositive matrix, then $\rho(A)$ is an eigenvalue of A.*

Remark 1.5.5. *This means that the eigenvalue of A with the largest absolute value is always positive if A is copositive.*

Another approach is to look at the values of A itself. Some necessary conditions can be identified:

Theorem 1.5.6. *[Vli11]: If A is a copositive $n \times n$ matrix, then*

- $A_{i,j} \geq -\frac{1}{2}(A_{i,i} + A_{j,j})$ *for all* $i \neq j$

- $\sum_{i \neq j} A_{i,j} \geq -\sum_i A_{i,i}$

- $A_{i,j} \geq -\sqrt{A_{i,i}A_{j,j}}$ *for all* $i \neq j$.

Since positive diagonal values can always be rescaled to 1 (Proposition 1.7.1) , the more specific case can be considered instead.

Theorem 1.5.7. *[Vli11]: If A is a copositive $n \times n$ matrix with $A_{i,i} = 1$ for all i,*

- $A_{i,j} \geq -1$ *for all* $i \neq j$

- $\sum_{i \neq j} A_{i,j} \geq -n$

In addition to that, there are plenty more certificates for copositivity. What is very interesting, especially if one tries to reduce the dimension of the matrix that has to be checked, is the following theorem that explains in which cases specific columns/rows can be omitted when checking the rest of the matrix for copositivity.

Theorem 1.5.8. *Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix with the partitioning:*

$$\begin{pmatrix} \alpha & \beta \\ \beta^T & B \end{pmatrix} \tag{1.1}$$

where B is an $(n-1)\times(n-1)$ (symmetric) matrix, $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}^{n-1}$. Then the following statements hold:

a) If $\alpha < 0$ then A is not copositive.

b) If $\beta \in \mathbb{R}^{n-1}_+$ and $\alpha \geq 0$ then A is copositive if and only if B is copositive

c) If $\beta \in \mathbb{R}^{n-1}_-$ and $\alpha > 0$ then A is copositive if and only if

$$\alpha B - \beta\beta^T$$

is copositive [Bom00] .

d) If $\alpha = 0$ and one entry of b is negative, then A is not copositive [BE10].

Remark 1.5.9. *The decomposition in this theorem can be accomplished for any row/-column of A since every permuted version of A shares the same copositivity property as A.*

1.6 Inverse A^{-1} of a copositive matrix A

Another interesting aspect to consider when dealing with a copositive matrix A is whether this property also translates to its inverse A^{-1}. Unfortunately, this is not the case. One cannot make a general statement about the copositivity of the inverse of a copositive matrix, as can be seen in these two examples [HS10]:

$$A = \begin{pmatrix} 1 & \sqrt{2} \\ \sqrt{2} & 1 \end{pmatrix} \qquad A^{-1} = \begin{pmatrix} -1 & \sqrt{2} \\ \sqrt{2} & -1 \end{pmatrix}$$

Figure 1.1: A is copositive (symmetric, nonnegative), however A^{-1} fails to inherit this property

$$A = \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix} \qquad A^{-1} \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}$$

Figure 1.2: A is copositive (symmetric, nonnegative) and so is A^{-1}

Even though one cannot be sure that A^{-1} will be copositive or not, some results still hold in any case.

Proposition 1.6.1. *[HS10]: If A is a nonsingular and copositive matrix, each column of A^{-1} contains a positive entry.*

Lemma 1.6.2. *[Jac76]: For a nonsingular, symmetric matrix A, the following statements are equivalent:*

- $\{x \in \mathbb{R}^n : x^T Ax \geq 0\} \subset \{x \in \mathbb{R}^n : x^T A^{-1} x \geq 0\}$
- *There is a scalar $r \geq 0$ such that $A - rA^3$ is positive semidefinite.*

But investigating the inverse can also be used in the other direction, as can be seen in the following theorem.

Theorem 1.6.3. *[Väl86]: A symmetric matrix A is not copositive if and only if it contains a nonsingular principal submatrix B such that a column of B^{-1} is nonpositive.*

1.7 Maintaining copositivity while manipulating a matrix

As already hinted by Theorem 1.5.8, it can be beneficial to look at certain matrix decompositions - but even more operations can be applied to matrices that preserve copositivity.

Proposition 1.7.1. *[JR09]: A real, symmetric matrix A with only positive diagonal elements is a copositive matrix if and only if the rescaled matrix where all diagonal elements are 1 is copositive.*

This is indeed an important proposition, as it allows us to only consider matrices with ones on the main diagonal.
Another 'manipulation' comes with the following proposition, which potentially allows changing a matrix such that it gets easier to check it for copositivity.

Proposition 1.7.2. *[BSU12]: If A is a copositive matrix and B is a rectangular matrix of matching order with no negative entries, then $B^T AB$ is again a copositive matrix.*

Remark 1.7.3. *Even though the other direction is not valid (i.e., non-copositivity is not necessarily preserved by the multiplication of such a matrix), applying this proposition can still be useful. If the resulting matrix can be proven to be not copositive, also the original matrix cannot be copositive.*

But what about the combination of two matrices? There are some results concerning this as well.

Proposition 1.7.4. *[Yua90]: Let $A, B \in \mathbb{R}^{n \times n}$ be symmetric matrices and C, D be two closed sets in $\mathbb{R}^n$ such that $C \cup D = \mathbb{R}^n$.*
If

$$\forall x \in C : x^T Ax \geq 0 \text{ and } \forall x \in D : x^T Bx \geq 0$$

then

$$\exists t \in [0,1] : tA + (1-t)B \text{ is positive semidefinite.}$$

Proposition 1.7.5. *[CMS95]: Let $A, B \in \mathbb{R}^{n\times n}$ be symmetric matrices. Then*

- $\exists t \in [0,1] : tA + (1-t)B$ *is copositive*
- $\forall u, v \in \mathbb{R}^n : \max\{u^TAu + v^TAv, u^TBu + v^TBv\} \geq 0$

are equivalent.

The summation of copositive matrices always results in another copositive matrix. The other direction does not hold.

Theorem 1.7.6. *If $A, B \in \mathbb{R}^{n\times n}$ are copositive, symmetric matrices and $A + B = C$, then C is again copositive and symmetric.*

Proof. We have

$$\forall x \in \mathbb{R}^n : x^TAx \geq 0,\ x^TBx \geq 0$$

and because of the distributive property, we get

$$x^TCx = x^T(A+B)x = x^TAx + x^TBx \geq 0.$$

□

But now, let us come back to the scenario of applying certain decompositions to a given matrix.

Theorem 1.7.7. *[Bom08][Bom96]: Let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix with the decomposition*

$$\begin{pmatrix} B_{dec} & D_{dec} \\ D_{dec}^T & C_{dec} \end{pmatrix} \tag{1.2}$$

where $B_{dec} \in \mathbb{R}^{m\times m}$, $C_{dec} \in \mathbb{R}^{l\times l}$ and $D_{dec} \in \mathbb{R}^{m\times l}$ with $m + l = n$. Let B_{dec} be positive definite and $B_{dec}^{-1}D_{dec}$ be nonpositive. Then A is copositive if and only if $C_{dec} - D_{dec}^TB_{dec}^{-1}D_{dec}$ is copositive.

Remark 1.7.8. *If we assume B_{dec} only copositive instead, this result does not hold any longer - consider the following example:*

$$B_{dec} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix},\ D_{dec} = \begin{pmatrix} -1 \\ -1 \end{pmatrix},\ C_{dec} = \begin{pmatrix} 0.7 \end{pmatrix}$$

Then we have that

$$B_{dec}^{-1} D = \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{3} \\ -\frac{1}{3} \end{pmatrix}$$

is nonpositive and

$$C_{dec} - D_{dec}^T B_{dec}^{-1} D_{dec} = 0.7 - \begin{pmatrix} -1 \\ -1 \end{pmatrix}^T \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} \begin{pmatrix} -1 \\ -1 \end{pmatrix} = 0.7 - \frac{2}{3} > 0$$

is copositive. But the original matrix

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 1 & -1 \\ -1 & -1 & 0.7 \end{pmatrix}$$

is not copositive, one violating vector is, e.g., $x = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$ *with* $x^T A x = -\frac{3}{10} < 0$.

The following is another result with the same decomposition - but different conditions.

Proposition 1.7.9. *If $A \in \mathbb{R}^{n \times n}$ is a symmetric matrix with a decomposition*

$$\begin{pmatrix} B_{dec} & D_{dec} \\ D_{dec}^T & C_{dec} \end{pmatrix} \tag{1.3}$$

such that $B_{dec} \in \mathbb{R}^{m \times m}$ and $C_{dec} \in \mathbb{R}^{l \times l}$ are copositive (the symmetry follows directly from the symmetry of A) and $D_{dec} \in \mathbb{R}^{m \times l}$ is nonnegative with $m + l = n$, then A is copositive.

Proof. Proving this is trivial - in Theorem 1.7.6 it was shown that the sum of copositive matrices is always copositive. (And a copositive matrix can always be padded with zero rows/columns to have arbitrary order while preserving copositivity) □

This means that if a matrix is brought in a form where nonnegative values accumulate in the upper right and lower left corner and hence forming ‘rectangles‘ in the corners such that the decomposition from Proposition 1.7.9 can be applied, only B and C have to be checked for copositivity to verify the same for A. This is the main idea of the copositivity checking procedure explained in the following chapters.

2 Matrix reordering

Now that a theoretical understanding of the term ‘copositivity‘ and the consequences that come along with this property are established, we can try to use it to speed up the process of identifying a matrix as copositive (or not). The main idea is to treat a symmetric, real matrix A as an adjacency matrix, where all the negative entries represent a 1 and all non-negative entries correspond to a 0. The first step of the algorithm is then to reorder this adjacency matrix such that connected components can easily be identified - and more importantly, components that are not connected can be split up into individual parts easily. These components are then checked for copositivity.
First, the theoretical foundation for the algorithm will be laid in the following. After that, the algorithm will be explained by showing the respective Python code and concrete examples to illustrate the process. Also, the time complexity of the entire procedure will be discussed.

The first step is to investigate ways to reorder a matrix such that a decomposition like in Proposition 1.7.9 can be applied - one has to note that this is by no means always possible, but rather only holds for a special kind of matrices that will be discussed in this chapter.

2.1 Theoretical aspects of the reordering algorithm

Definition 2.1.1. *For a matrix $A \in \mathbb{R}^{n\times n}$ define the corresponding negative-based adjacency matrix A^{d^*} as*

$$A^{d^*}_{i,j} = \begin{cases} 0, \text{ for } A_{i,j} \geq 0, i \neq j, \\ 1, \text{ for } A_{i,j} < 0, i \neq j, \\ 1, \text{ for } A_{i,i} \end{cases}$$

.

Remark 2.1.2. *The main diagonal elements are set to* 1 *for the sole purpose of more convenient use in later algorithms - this operation does not affect any outcome presented in this thesis, it just simplifies implementation.*

$$\begin{pmatrix} 1 & -2 & 7 & 9 & 5 \\ -2 & 1 & -1 & 1 & 4 \\ 7 & -1 & 1 & 5 & 2 \\ 9 & 1 & 5 & 1 & -3 \\ 5 & 4 & 2 & -3 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 2.1: Left: original matrix A; Right: respective matrix A^{d^*}

Corollary 2.1.3. *If A is a symmetric matrix, then A^{d^*} is also symmetric.*

Proof. This follows immediately from the definition of A^{d^*}. □

The consequence of this result is that A^{d^*} (if A is a real, symmetric matrix) represents an undirected graph. Hence, any reordering algorithm applicable to undirected graphs and suited for grouping connected components using an appropriate permutation of the rows and columns of A^{d^*} can be used to bring the matrix in the desired form. Suppose there are disconnected components in A^{d^*}. In that case, the wanted outcome is a permuted version of A^{d^*} where a decomposition as described in Proposition 1.7.9 is possible such that D is a zero block matrix and the matrices B and C contain both ones and zeros.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

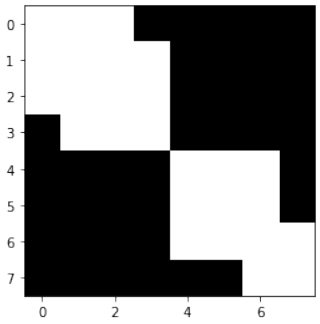

Figure 2.2: Decomposition as in Proposition 1.7.9 is possible since the reordered matrix consists of block matrices with the desired properties

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

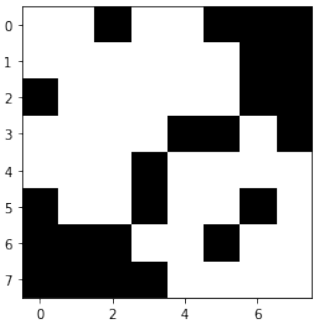

Figure 2.3: Case where a decomposition as in Proposition 1.7.9 is not possible

For the algorithm described in this thesis, the (reversed) Cuthill-McKee algorithm, as first mentioned in [CM69] is used to find the appropriate permutation. A description of the inner workings of this algorithm, together with an example, can be found at [Dos]. It is important to note that the reversed and the regular Cuthill-McKee algorithm are identical except for reversed index order, but since the reversed variant is superior to the original Cuthill-McKee algorithm in terms of computational complexity [CG80]. Because a fast implementation in Python using the 'Scipy' module [Vir+20] is available, this variant, rather than the original one, is used from now on.

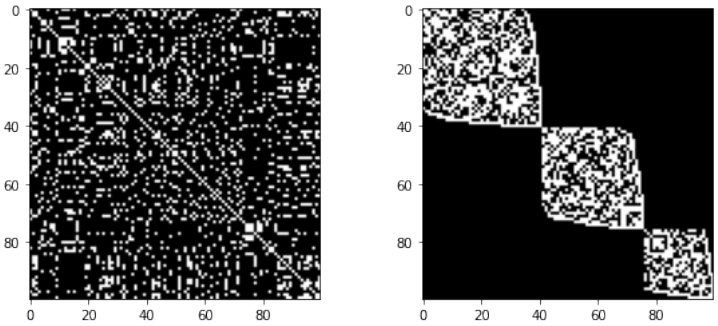

Figure 2.4: Left: unordered 'raw' adjacency matrix A^{d^*}; Right: the output of the reversed Cuthill-McKee algorithm

The result is that starting from the lower right corner, connected points are kept together while separate components form their own clusters. If the graph has separated components, a clearly visible structure indicating the split point between the components will arrive in the final matrix as described in Figure 2.2.

Once the desired permutation of A^{d^*} is reached, A gets permuted in the same way.

$$\begin{pmatrix} 1 & 4 & 1 & 9 & 7 & -8 & 3 & 1 \\ 4 & 1 & 8 & 3 & 3 & 3 & 0 & 1 \\ 1 & 8 & 1 & 8 & 3 & -8 & -5 & 7 \\ 9 & 3 & 8 & 1 & -10 & 2 & 4 & 6 \\ 7 & 3 & 3 & -10 & 1 & 3 & 3 & -5 \\ -8 & 3 & -8 & 2 & 3 & 1 & 8 & 8 \\ 3 & 0 & -5 & 4 & 3 & 8 & 1 & 9 \\ 1 & 1 & 7 & 6 & -5 & 8 & 9 & 1 \end{pmatrix}$$

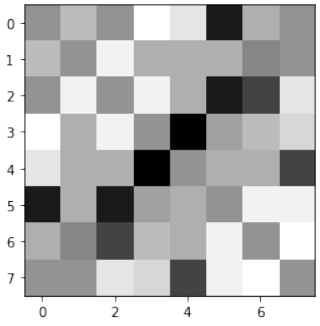

Figure 2.5: Original matrix without reordering, the graphic on the right side shows where negative/positive values are (the 'more' negative, the darker)

$$\begin{pmatrix} 1 & -5 & 6 & 9 & 7 & 8 & 1 & 1 \\ -5 & 1 & -10 & 3 & 3 & 3 & 7 & 3 \\ 6 & -10 & 1 & 4 & 8 & 2 & 9 & 3 \\ 9 & 3 & 4 & 1 & -5 & 8 & 3 & 0 \\ 7 & 3 & 8 & -5 & 1 & -8 & 1 & 8 \\ 8 & 3 & 2 & 8 & -8 & 1 & -8 & 3 \\ 1 & 7 & 9 & 3 & 1 & -8 & 1 & 4 \\ 1 & 3 & 3 & 0 & 8 & 3 & 4 & 1 \end{pmatrix}$$

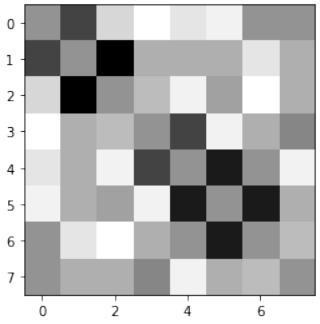

Figure 2.6: Reordered matrix according to the respective adjacency matrix reordering. One can spot a light rectangle in the top right and bottom left that does not contain a single negative value

2.2 Case distinction based on the reordering outcome

Given the result of the reversed Cuthill-McKee algorithm, there are two possible outcomes:

- There are two or more separated components in the graph
- The entire graph is one large connected component.

The following section will discuss the first case (2.3). The second case will be investigated in Chapter 3.

2.3 Case 1: separated components in the result of the Cuthill-McKee algorithm applied to A^{d^*}

Here we investigate the case that the resulting permuted version of A has separated components of negative values in the matrix.

$$\begin{pmatrix} 1 & -4 & -2 & 5 & 7 & 2 & 3 & 9 \\ -4 & 1 & -2 & -7 & 8 & 5 & 6 & 6 \\ -2 & -2 & 1 & -6 & -9 & 7 & 4 & 4 \\ 5 & -7 & -6 & 1 & -6 & 6 & 1 & 4 \\ 7 & 8 & -9 & -6 & 1 & 2 & 3 & 0 \\ 2 & 5 & 7 & 6 & 2 & 1 & -10 & 5 \\ 3 & 6 & 4 & 1 & 3 & -10 & 1 & -9 \\ 9 & 6 & 4 & 4 & 0 & 5 & -9 & 1 \end{pmatrix}$$

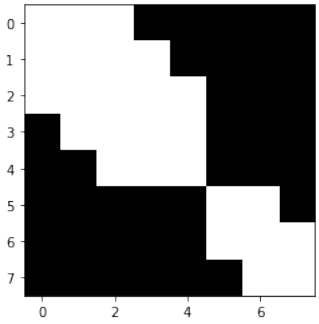

Figure 2.7: Left: matrix A; Right: respective matrix A^{d^*}

In this case, a variant of the decomposition discussed in Proposition 1.7.9 can be applied to the given matrix.

Theorem 2.3.1. *If $A \in \mathbb{R}^{n\times n}$ is a symmetric matrix such that A^{d^*} is a block diagonal matrix where the off-diagonal blocks are zero-matrices, i.e.,*

$$A^{d^*} = \begin{pmatrix} A_1^{d^*} & 0 & 0 & 0 \\ 0 & A_2^{d^*} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & A_m^{d^*} \end{pmatrix} \tag{2.1}$$

- where the number of separated components in A^{d^*} is m - then A is copositive if and only if the respective matrices $A_1, \dots A_n$ are copositive.

Proof. Follows immediately from repeated application of Proposition 1.7.9. □

This means that instead of checking the entire matrix for copositivity, it is sufficient to consider only these block matrices, which reduces computational complexity drastically!

2.4 Practical implementation

This section describes a practical implementation of the idea presented before. For implementation, the coding language Python was used, together with the packages "Numpy" [Har+20] and "Scipy" [Vir+20].

```
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import reverse_cuthill_mckee
```

Listing 2.1: Necessary imports

Remark 2.4.1. *These are all the imports needed to run the entire code in this chapter.*

As a first step, some prepossessing operations should be carried out on the given matrix A, namely, the deletion of rows/columns if one of the criteria given in Theorem 1.5.8 is satisfied since this reduces the computational complexity of any copositivity test without any downsides. This also means that after the preprocessing step, all diagonal elements of the processed matrix will be positive (i.e., $A_{i,i} > 0$) - because a negative value would cause the matrix to be immediately flagged as not copositive and zero values lead to a deletion of the respective row according to the constraints given in Theorem 1.5.8. But before introducing such a preprocessing function, two helper functions are discussed.

As already discussed in Proposition 1.7.1, the main diagonal of A can always be scaled to 1 while keeping the copositivity property of A. Since this form of normalization allows a more general approach to determining copositivity, it is also applied to every matrix before applying subsequent algorithms.

```
def normalize_matrix(matrix: np.ndarray) -> np.ndarray:
    """normalizes passed matrix such that the main diagonal only
    consists of ones

    :param matrix: real valued, square matrix
    :type matrix: np.ndarray
    :return: rescaled matrix with ones on the main diagonal
    :rtype: np.ndarray
    """
    # Extract the diagonal elements of A
    diagonal = np.diag(matrix)

    # Calculate the outer product of the diagonal elements
    diagonal_product = np.outer(diagonal, diagonal)

    # Divide each element in A by the corresponding element in the
    diagonal product
    result = matrix / np.sqrt(diagonal_product)
    return result
```

Listing 2.2: 'Normalization' of matrix A

$$\begin{pmatrix} 1 & -7 & 1 & 3 & -2 & -3 \\ -2 & 8 & 0 & 7 & -5 & -8 \\ 4 & -1 & 2 & 13 & -3 & 1 \\ 10 & 3 & 7 & 6 & -4 & -3 \\ -5 & -1 & -3 & -5 & 5 & 15 \\ -3 & -8 & 1 & 4 & 12 & 7 \end{pmatrix} \qquad \begin{pmatrix} 1.0 & -2.47 & 0.71 & 1.22 & -0.89 & -1.13 \\ -0.71 & 1.0 & 0.0 & 1.01 & -0.79 & -1.07 \\ 2.83 & -0.25 & 1.0 & 3.75 & -0.95 & 0.27 \\ 4.08 & 0.43 & 2.02 & 1.0 & -0.73 & -0.46 \\ -2.24 & -0.16 & -0.95 & -0.91 & 1.0 & 2.54 \\ -1.13 & -1.07 & 0.27 & 0.62 & 2.03 & 1.0 \end{pmatrix}$$

Figure 2.8: Left: original matrix; Right: rescaled matrix with ones on main diagonal

So from now on we can assume that the diagonal entries of A are always equal to 1 (i.e. $A_{i,i} = 1$). As a next step, the function to calculate A^{d^*} can be introduced.

```
def matrix_to_sign(matrix: np.ndarray) -> np.ndarray:
    """calculates A^d^* from a matrix A

    :param matrix: real valued, square matrix
    :type matrix: np.ndarray
    :return: the sign matrix of A (A^d^*)
    :rtype: np.ndarray
    """
    matrix_signs = np.sign(matrix)
    matrix_signs[matrix_signs == 1] = 0
    matrix_signs[matrix_signs == -1] = 1
    matrix_signs += np.eye(matrix_signs.shape[0]).astype("int")
    return matrix_signs
```

Listing 2.3: Function to compute A^{d^*}

The procedure is the following: After taking the sign for every entry of the input matrix, the positive signed values are set to 0 and the negative entries to 1. Then the main diagonal is set to 1 to simplify the following calculations. This has no effect on the reordering of the matrix later on.

$$\begin{pmatrix} 1 & 3 & -2 & 8 & 2 & -5 \\ 3 & 1 & 8 & -6 & 2 & -10 \\ -2 & 8 & 1 & -9 & -4 & -6 \\ 8 & -6 & -9 & 1 & -2 & -1 \\ 2 & 2 & -4 & -2 & 1 & 3 \\ -5 & -10 & -6 & -1 & 3 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 2.9: Left: original matrix A; Right: respective matrix A^{d^*}

Now everything is in place to start the preprocessing of the ‘raw‘ matrix A.

```
def copositivity_check_diagonal(matrix: np.ndarray) -> bool:
    """checks if a matrix is not copositive based on its diagonal
    elements
    A matrix is flagged as not copositive if
    3. one diagonal element is negative
    4. diagonal element is 0 and there is a negative off-diagonal element

    :param matrix: real valued, square matrix
    :type matrix: np.ndarray
    :return: returns a boolean indicating if the matrix is not copositive
    :rtype: bool
    """
    copositive_bool = True
    # 3.
    if np.any(np.diag(matrix) < 0):
        copositive_bool = False
    # 4.
    if np.any(np.diag(matrix) == 0):
        for n, i in enumerate(np.diag(matrix)):
            if i == 0:
                if np.any(matrix[:, n] < 0):
                    copositive_bool = False
                    break
    return copositive_bool
```

Listing 2.4: Helper function to check non copositivity based on diagonal values

This function serves as a helper for the preprocessing function discussed next. It checks the diagonal values of a given matrix (and the respective rows/columns) for criteria that refute copositivity according to Theorem 1.5.8. First, it checks whether there are any negative elements on the main diagonal. The next step is to check whether, for a zero element, the respective row/column contains any negative values - both cases guarantee that the matrix cannot be copositive.

```
def preprocessing_matrix(matrix: np.ndarray) -> Tuple[np.ndarray, bool]:
    """perform the following preprocessing steps:
    1. delete rows/columns that only consist of nonnegative values
    2. adjust matrix if one row only consists of negative off-diagonal
    values

    :param matrix: real valued, square matrix
    :type matrix: np.ndarray
    :return: returns a tuple of the preprocessed matrix and a boolean
    indicating if the matrix is not copositive
    False means that it was flagged not copositive since at least one
    criterion was met
    True means that it was not flagged not copositive
    :rtype: np.ndarray
    """

    copositive_bool = True
    while True:
        flag1 = False
```

```
17          flag2 = False
18          # 1.
19          ones = np.where(np.sum(matrix_to_sign(matrix), axis=0) == 1)[0].
        tolist()[::-1]
20          if len(ones) == 0:
21              flag1 = True
22          else:
23              for z in ones:
24                  matrix = np.delete(matrix, z, axis=0)
25                  matrix = np.delete(matrix, z, axis=1)
26          if len(matrix) == 0:
27              flag1 = True
28              flag2 = True
29              break
30          # 2.
31          res = np.sum(matrix_to_sign(matrix), axis=0)
32          r = res == matrix.shape[0]
33          if np.any(r):
34              for n, i in enumerate(r):
35                  if i:
36                      gamma = matrix[n, n]
37                      b = matrix[:, n]
38                      b = np.delete(b, n)
39                      R = matrix.copy()
40                      R = np.delete(R, n, axis=0)
41                      R = np.delete(R, n, axis=1)
42                      matrix = gamma * R - np.outer(b, b)
43                      copositive_bool = copositivity_check_diagonal(matrix)
44                      if not copositive_bool:
45                          return matrix, copositive_bool
46                      matrix = normalize_matrix(matrix)
47                      break
48          else:
49              flag2 = True
50              break
51          if len(matrix) == 0:
52              flag1 = True
53              flag2 = True
54              break
55          if flag1 and flag2:
56              break
57
58      copositive_bool = copositivity_check_diagonal(matrix)
59
60      return matrix, copositive_bool
```

Listing 2.5: Preprocessing routine

The main goal of this preprocessing function is to reduce rows/columns that can be considered irrelevant when checking a matrix for copositivity. While doing that, also the other statements from Theorem 1.5.8 are assessed, resulting in a processed matrix and a boolean value indicating whether a violating condition was detected or not - ‘False‘ meaning that the matrix was flagged ‘not copositive‘.

First, all purely nonnegative rows/columns are deleted. Following this, rows/columns with negative off-diagonal values are excluded, and the remaining matrix is brought in the form given in Theorem 1.5.8. What is interesting to note in the code is that after each calculation of the new matrix in the second case, the resulting matrix is checked for copositivity using the 'copositivity_check_diagonal' function. This is because the newly calculated matrix could have negative values on the diagonal, a clear sign for non-copositivity - and also something that would break the normalization process (square root of negative values). Also note that the matrix will be processed as long as one or both cases are met - ensured by the 'While'-loop that only breaks if both flags are set to True, indicating that neither case 1 nor case 2 can be applied anymore.

$$\begin{pmatrix} 1 & 0 & 0 & 6 & 7 & 7 \\ 0 & 1 & 0 & -2 & -3 & -4 \\ 0 & 0 & 1 & 2 & -10 & 1 \\ 6 & -2 & 2 & 1 & 6 & -10 \\ 7 & -3 & -10 & 6 & 1 & 4 \\ 7 & -4 & 1 & -10 & 4 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & -2 & -3 & -4 \\ 0 & 1 & 2 & -10 & 1 \\ -2 & 2 & 1 & 6 & -10 \\ -3 & -10 & 6 & 1 & 4 \\ -4 & 1 & -10 & 4 & 1 \end{pmatrix}$$

Figure 2.10: Case 1: first row/column only consists of nonnegative values. Hence it is discarded

$$\begin{pmatrix} 1.0 & -0.57 & -0.38 & -0.49 & -0.35 \\ -0.57 & 1.0 & 0.64 & -0.49 & -0.73 \\ -0.38 & 0.64 & 1.0 & 0.27 & 0.72 \\ -0.49 & -0.49 & 0.27 & 1.0 & 0.57 \\ -0.35 & -0.73 & 0.72 & 0.57 & 1.0 \end{pmatrix} \qquad \begin{pmatrix} 1.0 & 0.56 & -1.07 & -1.21 \\ 0.56 & 1.0 & 0.1 & 0.68 \\ -1.07 & 0.1 & 1.0 & 0.49 \\ -1.21 & 0.68 & 0.49 & 1.0 \end{pmatrix}$$

Figure 2.11: Case 2: first row/column only consists of negative off-diagonal values. Hence it is discarded, and the respective new matrix is calculated based on the dropped row/column and the remaining matrix

The next step is already concerned with reordering the matrix - since there is a fast implementation of the reversed Cuthill-McKee algorithm already available for Python, this one is used instead of implementing an own (probably slower) version.

```
def sort_matrix_idx(matrix: np.ndarray) -> np.ndarray:
    """function to reorder an adjacency matrix using the reverse
    Cuthill-McKee algorithm

    :param matrix: symmetric adjacency matrix
    :type matrix: np.ndarray
    :return: indices for reordering (1-dimensional array)
    :rtype: np.ndarray
    """
    # matrix to sparse format with
    graph = csr_matrix(matrix)

    # now reordering with reverse_cuthill_mckee algorithm
    res = reverse_cuthill_mckee(graph)
    return res
```

Listing 2.6: Computing permutation of matrix A^{d^*}

Remark 2.4.2. *The function 'sort_ matrix_ idx' does not return the reordered matrix itself but rather the permutation array that is necessary to create it, i.e., it returns a list of indices that can be used to reorder the matrix* A^{d^*} *(and more importantly the original matrix A). Note that indices start with* 0*.*

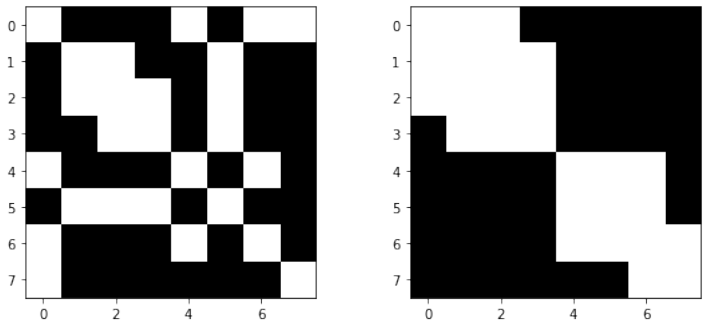

Figure 2.12: Left: unordered 'raw' adjacency matrix A^{d^*}; Right: ordered version when applying the permutation array: [3 5 2 1 6 4 0 7]

The next step is reordering the matrix A according to the permutation that was returned from the previous function computed for A^{d^*}.

```
def rearrange_matrix(matrix: np.array, idx: np.array) -> np.array:
    """reorders a matrix according to a passed permutation (idx)

    :param matrix: real, symmetric matrix
    :type matrix: np.array
    :param idx: 1-dimensional array holing the indices for permutation
    :type idx: np.array
    :return: return the permuted matrix
    :rtype: np.array
    """
    matrix[:, :] = matrix[idx, :]
    matrix[:, :] = matrix[:, idx]
    return matrix
```

Listing 2.7: Reordering of matrix A

Now, if there are separated components in A^{d*}, the resulting block matrices should be clearly visible by looking at the returned matrix and the respective, new matrix A^{d*} of the permuted matrix A - as shown for example in Figures 2.13 and 2.14.
How to handle cases where no such block matrices can be identified will be discussed in Chapter 3. But in this chapter we assume the case where two or more separated components are present.

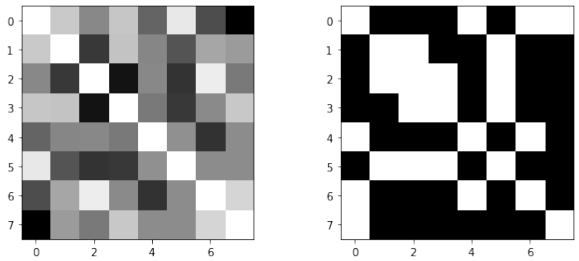

Figure 2.13: Unordered matrix A, Left: values colored from high (bright) to low, e.g. negative values (dark); Right: respective matrix A^{d*}

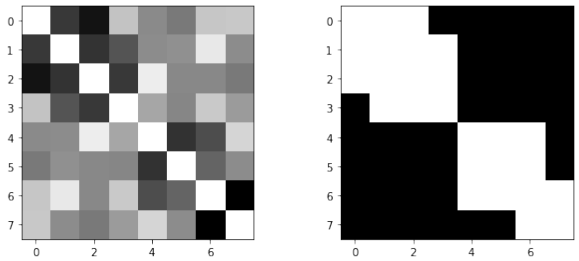

Figure 2.14: Reordered matrix A, Left: values colored from high (bright) to low, e.g. negative values (dark); Right: respective matrix A^{d*}

Once the reordering of A is done, the only thing that is left to do is to split the matrix at the right spots. These spots can be determined by traversing the matrix from the bottom right towards the upper left corner. First, a clearly defined route from one corner to the other is found using function described in Listing 2.8.

```
def traverse_matrix(matrix: np.ndarray) -> np.ndarray:
    """from bottom right corner walk the graph up towards upper left
    corner
    and find split-points, keeping track how far
    we are away from the diagonal at every step!

    :param matrix: adjacency matrix
    :type matrix: np.ndarray
    :return: _description_
    :rtype: np.ndarray
    """
    cur_point_x = matrix.shape[0] - 1
    cur_point_y = matrix.shape[0] - 1
    points = []
    while cur_point_y > 0:
        points.append([cur_point_y, cur_point_x])
        if matrix[cur_point_y, cur_point_x - 1] == 1 and cur_point_x - 1
    >= 0:
            cur_point_x -= 1
        elif cur_point_x == cur_point_y:
            cur_point_x -= 1
            cur_point_y -= 1
        elif (
            matrix[cur_point_y - 1, cur_point_x - 1] == 1
            and cur_point_y - cur_point_x == 1
        ):
            cur_point_x -= 1
            cur_point_y -= 1
        else:
            cur_point_y -= 1
    points.append([cur_point_y, cur_point_x])

    point_array = np.array(points)
    return point_array
```

Listing 2.8: Traverse reordered matrix A

The algorithm itself is just a very simple implementation following the idea of jumping from fields with a 1 to fields with a 1 if possible (either straight ahead or diagonally). Otherwise, it moves upwards towards the main diagonal. It essentially tries to walk on the ‘boundary‘ of the components visible in the graph. Note that it is impossible that the route ever crosses the main diagonal. Also, due to the nature of the reordering algorithm, there is no need to ever move ‘down‘ or ‘right.‘

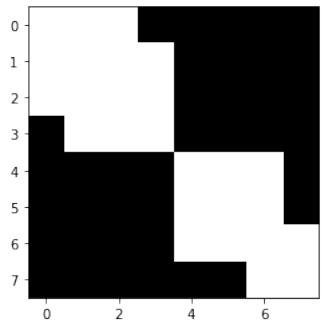

Figure 2.15: Reordered matrix A^{d*}, the respective traverse route for this matrix is (row,column):
(7, 7), (7, 6), (6, 5), (6, 4), (5, 4), (4, 4), (3, 3), (3, 2), (3, 1), (2, 1), (2, 0), (1, 0), (0, 0)

From this traverse route, now the points for splitting up the matrix can be decided very easily - they are simply the points where the row index is equal to the column index. Using the ‘coordinates‘ from the route the algorithm has taken, it can easily be checked whether at some point the x and y coordinates are identical - implying that one component is ‘complete‘ and the next, separate one will be entered in the next step.

```
def split_sorted_matrix_exact(traverse_route: np.ndarray, matrix: np.ndarray) -> list:
    """this function splits a matrix based on the traverse-route calculated for the respective sign matrix

    :param traverse_route: the traverse route calculated for the sign matrix
    :type traverse_route: np.ndarray
    :param matrix: original matrix for which the sign matrix was calculated
    :type matrix: np.ndarray
    :return: list of matrices split up according to the traverse route
    :rtype: list
    """
    diff = traverse_route[:, 0] - traverse_route[:, 1]
    coord = np.where(diff == 0)[0]
    coord = coord.reshape(-1, 2)
    split_up_matrices = []
    for c in coord:
        start = traverse_route[c[1]][0]
        end = traverse_route[c[0]][0]
        split_up_matrices.append(matrix[start : end + 1, start : end + 1])
    return split_up_matrices
```

Listing 2.9: Split matrix according to diagonal block

This exact idea is performed in the last function of this section - Listing 2.9 - where first the difference between the x and y coordinates is calculated, which is used to split the matrix up in the individual block matrices that can then further be checked for copositivity according to Proposition 1.7.9.

As already discussed, first, the function looks for zero values in the (elementwise) difference between the x and y coordinates - note that these zeros always come in pairs (i.e., if there is just one large component, there will be 2 zeros in this difference array, if there are two components then it is 4 zeros, etc.). Finally, take the matrices enclosed by the respective two crossings of the traverse route with the main diagonal and extract the matrix enclosed by them. The result is a list containing the separated block matrices according to the connected components in the graph A^{d^*} that can now individually - and in parallel - be checked for copositivity with any copositivity check. Once either one of the matrices is proven to be not-copositive (by finding a violating vector) or all of them are proven to be copositive, the copositivity property for the original matrix A can be derived accordingly.

All of these code parts can now be put together, resulting in a routine with the potential to reduce the time necessary to check for copositivity drastically - given the graph consists of separated components.

```
1  def main_work(matrix: np.ndarray) -> list:
2      """this function acts as an orchestrator for the introduced functions
3      and calculates the split up matrices according to the connected
       components
4      of the respective adjacency matrix
5
6      :param matrix: real valued, symmetric matrix
7      :type matrix: np.ndarray
8      :return: list of matrices according to the connected components
9      :rtype: list
10     """
11     m_signs = matrix_to_sign(matrix)
12     res = sort_matrix_idx(m_signs)
13     m = rearrange_matrix(matrix, res)
14     traverse_route = traverse_matrix(matrix_to_sign(matrix))
15     split_matrices = split_sorted_matrix_exact(traverse_route, m)
16     return split_matrices
```

Listing 2.10: Complete workflow

At this point, this function should be self-explanatory. What remains to show is a final example of the resulting output.

2.5 Example of entire algorithm

First, we introduce the original matrix, which is 11×11 in this case.

$$\begin{pmatrix}
1.0 & 0.28 & 0.79 & 0.4 & 0.3 & -0.54 & 0.55 & -0.89 & 0.48 & 0.56 & -0.57 \\
0.28 & 1.0 & 0.08 & 0.5 & 0.99 & 0.66 & 0.2 & 0.83 & -0.79 & 0.83 & 0.58 \\
0.79 & 0.08 & 1.0 & -0.18 & -0.73 & 0.13 & 0.58 & 0.16 & 0.06 & -0.97 & 0.06 \\
0.4 & 0.5 & -0.18 & 1.0 & 0.69 & 0.89 & 0.07 & 0.54 & 0.73 & -0.63 & 0.95 \\
0.3 & 0.99 & -0.73 & 0.69 & 1.0 & 0.72 & 0.13 & 0.57 & 0.22 & 0.06 & 0.04 \\
-0.54 & 0.66 & 0.13 & 0.89 & 0.72 & 1.0 & 0.8 & -0.32 & 0.96 & 0.74 & 0.58 \\
0.55 & 0.2 & 0.58 & 0.07 & 0.13 & 0.8 & 1.0 & 0.72 & -0.01 & 0.98 & 0.79 \\
-0.89 & 0.83 & 0.16 & 0.54 & 0.57 & -0.32 & 0.72 & 1.0 & 0.2 & 0.08 & 0.59 \\
0.48 & -0.79 & 0.06 & 0.73 & 0.22 & 0.96 & -0.01 & 0.2 & 1.0 & 0.25 & 0.64 \\
0.56 & 0.83 & -0.97 & -0.63 & 0.06 & 0.74 & 0.98 & 0.08 & 0.25 & 1.0 & 0.36 \\
-0.57 & 0.58 & 0.06 & 0.95 & 0.04 & 0.58 & 0.79 & 0.59 & 0.64 & 0.36 & 1.0
\end{pmatrix}$$

Figure 2.16: Large example matrix A

Obviously, the algorithm's main step is the reordering step.

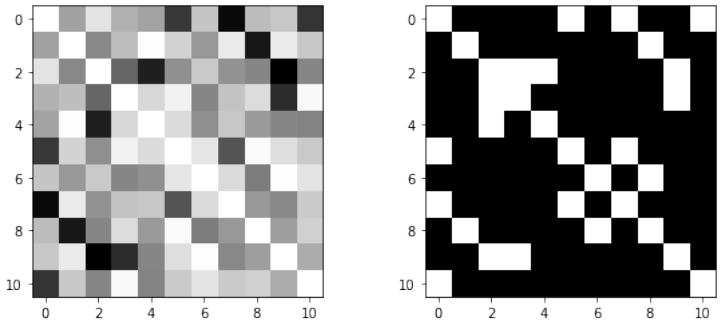

Figure 2.17: Left: matrix A depicted as image; Right: respective matrix A^{d^*}

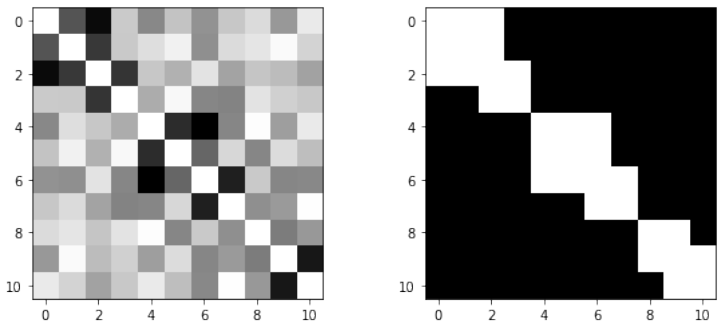

Figure 2.18: Left: reordered matrix A; Right: respective matrix A^{d^*}

And from this, the other matrices can be derived already. As can be seen in Figure 2.18, there will be 3 parts in the end - of sizes 4, 4, and 3.

$$\begin{pmatrix} 1.0 & -0.32 & -0.54 & 0.58 \\ -0.32 & 1.0 & -0.89 & 0.59 \\ -0.54 & -0.89 & 1.0 & -0.57 \\ 0.58 & 0.59 & -0.57 & 1.0 \end{pmatrix}$$

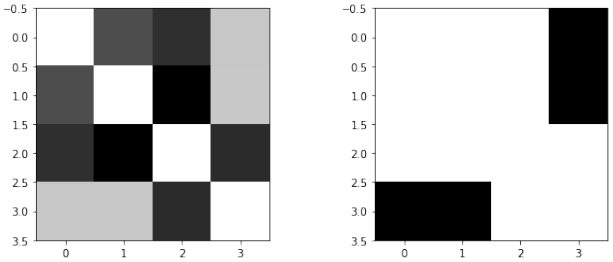

Figure 2.19: First separated component isolated

$$\begin{pmatrix} 1.0 & -0.63 & -0.18 & 0.69 \\ -0.63 & 1.0 & -0.97 & 0.06 \\ -0.18 & -0.97 & 1.0 & -0.73 \\ 0.69 & 0.06 & -0.73 & 1.0 \end{pmatrix}$$

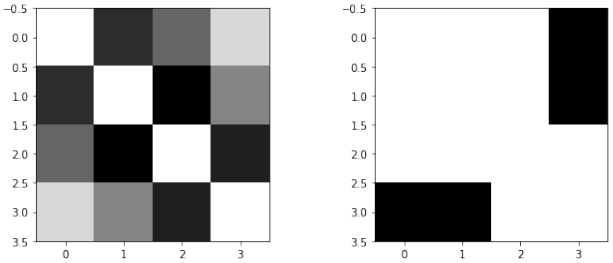

Figure 2.20: Second separated component isolated

$$\begin{pmatrix} 1.0 & -0.79 & 0.2 \\ -0.79 & 1.0 & -0.01 \\ 0.2 & -0.01 & 1.0 \end{pmatrix}$$

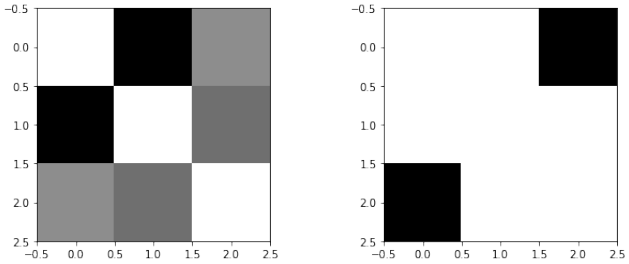

Figure 2.21: Third separated component isolated

Now, to verify or refute copositivity, only these three matrices have to be checked, which is significantly easier than checking the original matrix!

We continue the example by investigating the first isolated matrix shown in Figure 2.19.

$$\begin{pmatrix} 1.0 & -0.32 & -0.54 & 0.58 \\ -0.32 & 1.0 & -0.89 & 0.59 \\ -0.54 & -0.89 & 1.0 & -0.57 \\ 0.58 & 0.59 & -0.57 & 1.0 \end{pmatrix} \quad \begin{pmatrix} 1.0 & -2.09 & 0.39 \\ -2.09 & 1.0 & 0.22 \\ 0.39 & 0.22 & 1.0 \end{pmatrix} \quad \begin{pmatrix} 1.0 & -2.09 \\ -2.09 & 1.0 \end{pmatrix}$$

Figure 2.22: Left: the first isolated matrix; Center: the second column/row was removed since it only contained negative off-diagonal values and the matrix was adjusted accordingly; Right: the third column was removed since now it only contained nonnegative values; the resulting 2×2 matrix is obviously not copositive, a violating vector is, for example, $(1, 1)$, hence the original 11×11 matrix is also not copositive.

Since the first matrix investigated already refuted copositivity of A, there is no need to check any further matrices. If, on the other hand, this first isolated matrix was copositive, we would have continued with the second and the third matrix in the same way until either all of them are proven copositive, or we find at least one matrix which is not.

2.6 Time complexity of the algorithm

In this section, we will have a brief look at a rough estimate of the computational complexity of this algorithm. But note that the here described complexity estimate only holds for the case presented in this chapter and is not valid for the second case, which will be described in Chapter 3.

Let us investigate the individual functions step by step, starting with the function 'matrix_to_sign' 2.3 - here every element is replaced by another value, hence it is $\mathcal{O}(n^2)$.

For normalization (2.2), retrieving the diagonal can be done in linear time, and the outer product calculation, as well as the elementwise division and the elementwise square root operation, can be done in $\mathcal{O}(n^2)$.

Checking the diagonal values as part of a copositivity check introduced in Section 2.4 consists of the mentioned two cases. The first case, checking for negative diagonal values, can be done in linear time. The second case requires checking every element in the upper triangular part of the matrix, which in the worst case (if all main diagonal elements are zero and there is only 1 negative value in this subset of the matrix) leads again to $\mathcal{O}(n^2)$

The preprocessing part 2.5 is a bit trickier to evaluate. Summing a matrix along the columns needs n^2 summations, and checking the resulting array for occurrences of the

value 1 happens in linear time. Deleting rows/columns is, in the worst case, a $\mathcal{O}(n^2)$ operation. For the second check, calculating the individual parts for generating the new matrix and the calculation itself can also be summarized as $\mathcal{O}(n^2)$. So, overall the complexity for one iteration is $\mathcal{O}(n^2)$.

Remark 2.6.1. *There is a chance that the process could restart from the beginning with only one row removed in each of the previous steps until the matrix gets very small. Given that a matrix exists that would trigger this behavior, the overall complexity would result in* $\mathcal{O}(n^3)$.

The sorting algorithm 2.6 according to [CG80] is linear in the number of non-zero entries of the matrix, hence the complexity is $\mathcal{O}(n^2)$ in the worst case. The same holds for rearranging the matrix according to the given indices.

Traversing the matrix after that in 2.8 is only a $\mathcal{O}(n)$ operation. The algorithm responsible for splitting the matrix, in the end, is $\mathcal{O}(n^2)$ in the worst case.

Overall, for one step of the algorithm, the complexity of the algorithm can therefore be described with $\mathcal{O}(n^2)$.
In the very unlikely 'special case' described in Remark 2.6.1, the entire matrix can be checked for copositivity in $\mathcal{O}(n^3)$ operations, which is incredibly fast (considering this is otherwise an NP-complete problem).

3 Case 2 overlapping matrices - the general case

In Proposition 1.7.9 the case with the decomposition

$$A = \begin{pmatrix} B_{dec} & D_{dec} \\ D_{dec}^T & C_{dec} \end{pmatrix}$$

for a symmetric matrix A (after preprocessing steps and reordering) was discussed.

Unfortunately, this has to be considered a special case. In most situations, it is unlikely that matrices can be decomposed as described in Theorem 2.3.1. This means, in order to build a practically useful algorithm, a more general approach that can be applied in any case has to be found.

Theorem 3.0.1. *Let A be a real, symmetric matrix where all preprocessing steps were applied and the matrix was reordered according to the procedure described in Section 2.4. Then the following decomposition of A is always possible.*

$$B = \begin{pmatrix} B_{dec} & 0 \\ 0 & 0 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

with $B_{dec} \in \mathbb{R}^{m \times m}$

$$C = \begin{pmatrix} 0 & 0 \\ 0 & C_{dec} \end{pmatrix} \in \mathbb{R}^{n \times n}$$

with $C_{dec} \in \mathbb{R}^{l \times l}$

$$D = \begin{pmatrix} 0 & 0 & D_{dec} \\ 0 & 0 & 0 \\ D_{dec}^T & 0 & 0 \end{pmatrix} \in \mathbb{R}_+^{n \times n}$$

with $D_{dec} \in \mathbb{R}_+^{k \times h}$

$$E = \begin{pmatrix} 0 & 0 & 0 \\ 0 & E_{dec} & 0 \\ 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

with $E_{dec} \in \mathbb{R}^{u \times u}$

such that E_{dec} is the overlapping part of the matrices B_{dec} and C_{dec}. This means that $m + l - u = n$, $m + h = n$ and $l + k = n$ and $A = B + C + D - E$.

Proof. Assume A to be a matrix as described. Then there must exist at least one nonnegative entry in each row/column of the matrix (otherwise the row would have been deleted in the preprocessing steps of Theorem 1.5.8). Hence, there always exists a permutation such that there is a nonnegative entry in the upper right corner of the matrix. This means, B_{dec} can be set to only consist of this one entry and the other matrices accordingly, which is a valid (but not very useful) decomposition according to the theorem. □

The matrix E represents the overlapping part between the matrices B and C - i.e. where their respective adjacency matrices could have values $\neq 0$. In the case described in the previous chapter where the decomposition of Proposition 1.7.9 could be applied, there was no overlapping part, hence E was just a matrix of order 0.
The matrix D consists of nonnegative values that accumulate in the corners of the matrix A after reordering. Lastly, note

$$A - D = (B - E) + (C - E) + E$$

for later use.

Given this decomposition of the matrix A, we can specify a test for copositivity based on the following idea: Since the sum of copositive matrices is again copositive, it is sufficient to prove copositivity of summands of a matrix to conclude copositivity of the entire matrix.

Before a concrete algorithm is outlined, the individual parts that make this idea work are discussed. First, have a look at two obvious steps:

Corollary 3.0.2. *If A is a real, symmetric matrix with the decomposition as discussed in Theorem 3.0.1, then the resulting matrix D is always copositive.*

Proof. This is trivial since the matrix D only consists of nonnegative elements, hence it is copositive. □

Corollary 3.0.3. *If A is a real, symmetric matrix with the decomposition as discussed in Theorem 3.0.1, then A is not copositive if the matrix E_{dec} is not copositive (and hence E is not copositive).*

Proof. E_{dec} is by definition a principal submatrix of A. For A to be copositive, all principal submatrices have to be copositive as well (as introduced in Theorem 1.3.7). Hence, if E is not copositive, A is not either. □

The easy part is done, now one has to investigate the matrices B and C that result from the decomposition of A. The problem is that B and C are overlapping. Therefore it is not immediately clear how to check B and C for copositivity. The reason for this is, that it is necessary but not sufficient for B and C to be copositive matrices for A to be the same. If summed together, the E part would be doubled (because it appears in both matrices), hence we would have to account for that and subtract the E part again to end up with the original matrix A. And this subtraction is the violating part, as one cannot

make a general assumption about whether a matrix is still copositive or not after another (copositive) matrix is subtracted.
This circumstance shows that checking for copositivity this way will not produce a reliable result.

Hence, a better way has to be found.

Theorem 3.0.4. *If A is a real, symmetric matrix with the decomposition as discussed in Theorem 3.0.1, then A is copositive if the matrices E, $(B-E)$ and $(C-E)$ are copositive.*

Proof. The statement follows immediately from the fact that the sum of copositive matrices is again copositive and D is always copositive. □

Remark 3.0.5. *In other words, to ensure copositivity of A it is sufficient to prove copositivity of the matrix $(A-D)$.*

Remark 3.0.6. *A can still be copositive even if $A-D$ is not copositive! To the knowledge of the author, there is no immediate way of telling whether A is copositive/not copositive given that $A-D$ is not copositive - except if a principal submatrix of A was not copositive.*

So now the goal is to check $(A-D)$, to approach this, the matrix E will be divided in two parts which are added to the matrices $(B-E)$ and $(C-E)$.

Theorem 3.0.7. *If A is a real, symmetric matrix with the decomposition as discussed in Theorem 3.0.1 , then A is copositive if there is a $\lambda \in [0,1]$ such that both*

$$(B-E)+\lambda E$$

and

$$(C-E)+(1-\lambda)E$$

are copositive.

Proof. If a λ exists such that both $(B-E)+\lambda E$ and $(C-E)+(1-\lambda)E$ are copositive, then $(B-E)+\lambda E+(C-E)+(1-\lambda)E = (B-E)+(C-E)+E = (A-D)$ is copositive, hence according to Remark 3.0.5 A is copositive. □

The main task is now to find such a λ. In the following, some cases that can occur while searching for this will be discussed.

Theorem 3.0.8. *Let A be a real, symmetric matrix that was preprocessed and reordered with a decomposition as in Theorem 3.0.1 and assume E, B, and C are copositive. Then there is always a $\lambda \in [0,1]$ such that either*

$$\textit{both } (B-E)+\lambda E \textit{ and } (C-E)+(1-\lambda)E \textit{ are copositive}$$

or

$$\textit{both } (B-E)+\lambda E \textit{ and } (C-E)+(1-\lambda)E \textit{ are not copositive.}$$

If there is a λ_1 such that both matrices are copositive, then there is no λ_2 such that both of them are not copositive and vice versa.

Before the proof for this theorem is shown, some explanation of the individual cases is necessary to understand the need for this theorem: While looking for a fitting λ to prove copositivity, it is possible that only one of both resulting matrices is copositive while the other one is not. But such a mixed result yields no valuable insight since no conclusion about overall copositivity can be made. Hence, one has to keep looking for a 'decisive' λ - the theorem ensures, that this search will always terminate in one of the cases presented.

Proof. Define the functions

$$f(\lambda) = \min\{x^T((B - (1-\lambda)E)x,\ x \in \mathbb{R}^n_+,\ \|x\| = 1\}$$

and

$$g(\lambda) = \min\{x^T((C - \lambda E)x,\ x \in \mathbb{R}^n_+,\ \|x\| = 1\}$$

for some norm $\|.\|$.

It is obvious, that both functions are continuous in the variable λ. Also, since the matrix E is assumed to be copositive, both functions are monotone - f is monotonically increasing, g is monotonically decreasing.
If for some λ either f or g is negative, the respective matrix is not copositive - on the other hand, if the function takes on a nonnegative value, the respective matrix is copositive.

If for some λ both functions evaluate to a negative result case 2 is satisfied, and if both functions evaluate to a nonnegative result, case 1 is satisfied.

Now the focus lies on showing that one of both cases is always satisfied. Without loss of generality, we say that for $0 \leq \lambda_1 < 1$ we have that $f(\lambda_1) < 0$ and $g(\lambda_1) \geq 0$ (e.g. we know that for $\lambda = 0,\ g(0) \geq 0$ since C is copositive).
Now, evaluate $f(1)$. If the result was negative, then B would be not copositive which contradicts the assumptions made on B! (In this case, the entire matrix A would not be copositive, since B is a principal submatrix).
This means that according to the intermediate value theorem

$$\exists \lambda \in (\lambda_1, 1] : f(\lambda) = 0,$$

we therefore choose

$$\lambda_2 = \min\{\lambda \in (\lambda_1, 1] : f(\lambda) = 0\}$$

Next, evaluate $g(\lambda_2)$, if the result is nonnegative, both functions are nonnegative with the same λ_2, and we are done since $f(\lambda_2) = 0$. If $g(\lambda_2) < 0$ on the other hand, there exists an $\epsilon > 0$ (continuity of f and g) such that $g(\lambda_2 - \epsilon) < 0$ and $f(\lambda_2 - \epsilon) < 0$ which completes this part of the proof.

What is left to check is that there can never co-exist $\lambda_1, \lambda_2 \in [0, 1]$ with

$$g(\lambda_1) < 0 \text{ and } f(\lambda_1) < 0$$

and

$$g(\lambda_2) \geq 0 \text{ and } f(\lambda_2) \geq 0.$$

Assume such λ's existed. Then by monotonicity of f and g

$$g(\lambda_1) < 0 \leq g(\lambda_2) \Rightarrow \lambda_2 < \lambda_1$$

and at the same time

$$f(\lambda_1) < 0 \leq f(\lambda_2) \Rightarrow \lambda_2 > \lambda_1$$

which are contradicting statements!

□

One way of determining the correct λ in a concrete implementation would be to apply some sort of bisection method, i.e., starting at $\lambda = 0.5$ and evaluating both matrices - if one is copositive and the other is not, continue with $\lambda = 0.75$ or $\lambda = 0.25$ respectively, etc. until both matrices are copositive or both are not.

Remark 3.0.9. *Even if it is not possible - to the author's knowledge at the time of writing this thesis - to gain a definitive answer about the copositivity/non-copositivity of A if $(A - D)$ is not copositive in a general case, it is still worth investigating some properties that are connected to the described procedure.*

Theorem 3.0.10. *If there is a λ such that both $B - (1 - \lambda)E$ and $C - \lambda E$ are not copositive, then $(A - D)$ is not copositive if the intersection of*

$$B_y^\lambda := \left\{ \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix} \in \mathbb{R}_+^n : \; y \in \mathbb{R}_+^u, \; \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}^T (B - (1-\lambda)E) \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} < 0, \text{ for some } x \in \mathbb{R}_+^v, \; \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \in \mathbb{R}_+^n \right\}$$

and

$$C_y^\lambda := \left\{ \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix} \in \mathbb{R}_+^n : \; y \in \mathbb{R}_+^u, \begin{pmatrix} 0 \\ y \\ z \end{pmatrix}^T (C - \lambda E) \begin{pmatrix} 0 \\ y \\ z \end{pmatrix} < 0, \text{ for some } z \in \mathbb{R}_+^w, \; \begin{pmatrix} 0 \\ y \\ z \end{pmatrix} \in \mathbb{R}_+^n \right\}$$

(where n is the order of the original matrix A and $u + v + w = n$), is not empty, i.e.,

$$B_y^\lambda \cap C_y^\lambda \neq \emptyset.$$

Proof. If the intersection of these two sets is not empty, then there exists a vector $\bar{y} \in B_y^\lambda \cap C_y^\lambda$. Also, based on the definition of both sets, there are vectors $\bar{x} \in B_y^\lambda$, $\bar{z} \in C_y^\lambda$ that satisfy

$$\bar{x}^T (B - (1 - \lambda)E)\bar{x} < 0$$

and

$$\bar{z}^T (C - \lambda E)\bar{z} < 0$$

with

$$\bar{x} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}, \; \bar{y} = \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix}, \; \bar{z} = \begin{pmatrix} 0 \\ y \\ z \end{pmatrix}, \; \bar{x}, \bar{y}, \bar{z} \in \mathbb{R}_+^n$$

such that both contain the $\bar{y}$ vector. Then we have that

$$0 > \bar{x}^T((B-E)+\lambda E)\bar{x} + \bar{z}^T((C-E)+(1-\lambda)E)\bar{z} =$$

$$(\bar{x}+\bar{z}-\bar{y})^T((B-E)+\lambda E)(\bar{x}+\bar{z}-\bar{y}) + (\bar{x}+\bar{z}-\bar{y})^T((C-E)+(1-\lambda)E)(\bar{x}+\bar{z}-\bar{y}) =$$

$$(\bar{x}+\bar{z}-\bar{y})^T(((B-E)+\lambda E)+((C-E)+(1-\lambda)E))(\bar{x}+\bar{z}-\bar{y}) = (\bar{x}+\bar{z}-\bar{y})^T(A-D)(\bar{x}+\bar{z}-\bar{y}) < 0$$

and $\bar{x}+\bar{z}-\bar{y} \in \mathbb{R}^n_+$ since the $\bar{y}$ part occurred in both vectors $\bar{x}$ and $\bar{z}$. The expansion of the vectors is valid due to the structure of the matrices B, C, and E. □

In summary, the approach is the following:
Once the decomposition Theorem 3.0.1 has been applied, the following steps have to be taken:

1. Check if E is a copositive matrix.
   - 1a) If E is copositive ⇒ continue with the next step
   - 1b) Else: A is not copositive.
2. Find a λ for the matrices B and C as described in Theorem 3.0.8
   - 2a) If both are copositive ⇒ A is copositive
   - 2b) If for one function there is no λ such that it becomes nonnegative ⇒ A is not copositive
   - 2c) Otherwise A is flagged 'not determined' - so no statement can be made whether A is copositive or not

Obviously, the algorithm can be applied recursively, splitting up very large matrices into smaller, easier-to-check parts.

3.1 Computational complexity

In this section, we have a look at the time complexity of this procedure. As far as for the reordering part, nothing has changed compared to Section 2.6. Equal to the decomposition in the first case, also, in this case, the overall complexity can be given by $\mathcal{O}(n^2)$ for one iteration (one iteration meaning the process of decomposing one matrix into the matrices B, C, D and E which can then be further analyzed).
Due to the nature of the algorithm, possibly multiple attempts have to be made in order to find a fitting λ to split up E accordingly - this is indeed quite a big, computational problem. If one assumes, that on average it takes μ steps to find the correct λ, this is equivalent to not just producing the aforementioned four matrices but instead

$$1(D) + 1(E) + \mu(B_1, B_2, \cdots, B_\mu) + \mu(C_1, C_2, \cdots C_\mu) = 2 + 2\mu$$

matrices - B_1 e.g. refers to $B - (1-\lambda_1)E$. From all of these matrices, $1+2\mu$ matrices would then have to be checked for copositivity (assuming again, that it takes μ steps to

find the correct λ), since D does not have to be checked.

In the worst case only a decomposition as described in the proof of Theorem 3.0.1 is possible, resulting in matrices E of size $(n-1)\times(n-1)$ and B and C of size $(n-1)\times(n-1)$. B and C can be further reduced to size $(n-2)\times(n-2)$ using the preprocessing technique, which is for sure applicable due to the fact that in the respective row, only one nonnegative value was present, meaning that the row without this value consists of only negative values, hence it can be removed following Theorem 1.5.8.

3.2 Practical implementation

The implementation part is only concerned with the actual decomposition of the matrix rather than performing a copositivity test on any of the matrices. Also, the produced outcome cannot be used for validating a matrix as copositive immediately, but rather gives an idea of how many matrices would have to be checked. Therefore, some code parts only act as a "delay" which should simulate behavior in reality.

First, a simple helper function is introduced.

```
1  def calc_size(matrix: np.ndarray) -> int:
2      """calculates the size of the matrix embedded in a zero matrix
3      by counting the number of ones on the diagonal of the matrix
4
5      :param matrix: matrix embedded in a zero matrix
6      :type matrix: np.ndarray
7      :return: size of the embedded matrix
8      :rtype: int
9      """
10     return np.sum(np.diag(matrix)).astype(int)
```

Listing 3.1: Calculating size of zero-padded matrix

This function comes in handy to understand how large a certain matrix really is because, in the next function, matrices will be embedded in zero matrices - as established, zero rows/columns can be ignored when checking for copositivity.

In the following function - Listing 3.2 - some familiar elements come to our attention, one important part is that we now do not have these clear splitting points as in the first case. This means that we have to figure out a way to find the best split possible, this is again done using the traverse path described in Listing 2.8 and finding the largest rectangles (i.e. the one with the largest ‘area‘) in the upper right and lower left corners which contain only zeros (our matrix D), that also defines the sizes of the matrices B and C. Then the matrix is simply split according to Theorem 3.0.1.

```
def split_connected(matrix: np.ndarray) -> list:
    """function that splits up a matrix according to the general
    decomposition

    :param matrix: real valued, symmetric matrix
    :type matrix: np.ndarray
    :return: list of the individual components
    :rtype: list
    """
    # now split up a graph if connected
    traverse = traverse_matrix(matrix_to_sign(matrix))
    # find the largest rectangle with only nonnegative values
    diff = np.zeros_like(traverse)
    diff[:, 0] = traverse[0, 0] - traverse[:, 0]
    diff[:, 1] = traverse[:, 1]
    diff_mult = diff[:, 0] * diff[:, 1]
    res = traverse[np.argmax(diff_mult)]

    matrix_B_E = np.zeros_like(matrix)
    tmp = matrix[0 : res[0] + 1, 0 : res[0] + 1].copy()
    tmp[res[1] :, res[1] :] = 0
    matrix_B_E[0 : res[0] + 1, 0 : res[0] + 1] = tmp

    matrix_E = np.zeros_like(matrix)
    tmp = matrix[res[1] : res[0] + 1, res[1] : res[0] + 1]
    matrix_E[res[1] : res[0] + 1, res[1] : res[0] + 1] = tmp

    matrix_C_E = np.zeros_like(matrix)
    tmp = matrix[res[1] : traverse[0, 0] + 1, res[1] : traverse[0, 0] +
    1].copy()
    tmp[: res[0] - res[1] + 1, : res[0] - res[1] + 1] = 0
    matrix_C_E[res[1] : traverse[0, 0] + 1, res[1] : traverse[0, 0] + 1]
    = tmp

    return matrix_E, matrix_B_E, matrix_C_E
```

Listing 3.2: Splitting up a matrix according to the general decomposition procedure

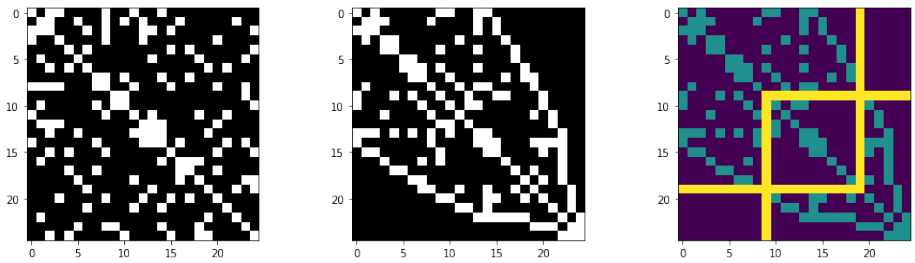

Figure 3.1: Left: original matrix; Center: reordered version; Right: found decomposition with the matrices B, C and E defined by the matrix D in the corners

Remark 3.2.1. *If there are multiple possible matrices D, i.e. several rectangles with the same, large area, the first rectangle that was found is chosen for the matrix D.*

This algorithm then has to be applied recursively on the individual parts. This can be done the following way:

```
def recursion(
    matrix: np.ndarray, matrices: list = [], min_matrix_len=7, lambd=3
) -> None:
    """This is the main recursion matrix

    :param matrix: real, symmetric matrix
    :type matrix: np.ndarray
    :param matrices: list object that is passed down the recursion to
    collect all found matrices, defaults to []
    :type matrices: list, optional
    """
    matrix_size = calc_size(matrix)
    if matrix_size <= min_matrix_len:
        matrices.append(matrix)
        return
    else:
        preprocessing_result = preprocessing_matrix(matrix)
        if preprocessing_result[0].shape[0] <= min_matrix_len:
            matrices.append(preprocessing_result[0])
            return
        # preprocessing
        preprocessed_matrix = preprocessing_result[0]

        # reordering
        m_signs = matrix_to_sign(preprocessed_matrix)
        res = sort_matrix_idx(m_signs)
        m = rearrange_matrix(preprocessed_matrix, res)

        # split matrix up
        res = split_connected(m, display=False)

        # now run the same for E
        recursion(res[0], matrices)
        for i in range(lambd):
            # next goes B + E
            recursion(
                res[1] + res[0], matrices, min_matrix_len=min_matrix_len,
    lambd=lambd
            )
            # next goes C + E
            recursion(
                res[2] + res[0], matrices, min_matrix_len=min_matrix_len,
    lambd=lambd
            )
```

Listing 3.3: Recursion for splitting a matrix into small matrices that can be checked easily for copositivity

When the function is called for the first time, it preprocesses the matrix and then splits it up using the aforementioned function. This procedure will be continued until a certain matrix size is achieved - the standard parameter is set to 7 since up to this order fast algorithms for checking copositive exist in any case ([YL09]). To mimic the search for the correct λ in each split, the number of matrices that are used to again call the function is increased artificially based on some (integer) value indicating the expected amount of steps necessary to find the correct λ (on average per step), i.e. simply repeating the respective function calls μ times.
The function is initialized with a list called 'matrices' that stores all the found matrices which can be used to count the total number of matrices after termination.

Remark 3.2.2. *It has to be noted that there is a lot of redundancy in the matrices that are investigated, submatrices that lie in the overlapping part of two matrices will be revisited two or more times. This implementation is by no means optimized to be applied to large matrices yet. Potential next steps on this algorithm will be outlined in the 'Future work' Section 6.1.*

4 Using gradient descent for copositivity testing

While there are lots of copositivity tests, not too many work out of the box for a wide range of matrices. Most are limited to a certain size of matrices (e.g. only work for matrices up to a certain order), have specific requirements on the matrix itself, i.e., expect the matrix to be of a certain form or are very hard to implement.
What holds for almost all tests is, that it is usually much easier (and faster) to find a violating vector for a matrix and therefore prove that it is not copositive than proving copositivity itself. While this algorithm is not an exception to this rule of thumb, it excels in terms of ease of use and execution speed. In an ideal case, the procedure can terminate and return a violating vector (given that such a vector exists) in a very short time.
The downside to this great property is, that it is not guaranteed to find one of the violating vectors - meaning that even if the algorithm terminates after a chosen amount of iterations without finding a violating vector, this is not a certificate for copositivity of the matrix. Hence, it should be seen as a chance to quickly identify non-copositive matrices with a very general algorithm that is easy to implement and which is applicable to matrices of any order and type, rather than a certificate for copositivity.

The 'heart' of the algorithm is the gradient-based search for a violating vector. The idea of using the gradient as a search direction to optimize for a certain target has been used in numerous fields and one cannot count the papers referencing such a search - the most popular application nowadays is most probably the procedure used to train neural networks (during the backward pass of a learning iteration, the gradients with respect to some loss function are calculated and the weights of the network are updated respectively). Also in the field of standard quadratic optimization problems, this approach has been used, for example in [BP05].
But, to the author's knowledge, using this technique for refuting copositivity of matrices has not yet been dealt with in the available literature. This section aims to discuss different approaches when it comes to the implementation and utilization of several gradient-based methods.

4.1 Problem Formulations

4.1.1 Problem formulation 1: $x^T Ax$

The original problem formulation for copositivity is to verify that

$$\forall x \in \mathbb{R}^n_+ : x^T Ax \geq 0$$

for a given matrix A for it to be copositive, which is equivalent to showing

$$\forall x \in \mathbb{R}^n_+ : x^T Ax \geq 0, \|x\| = 1,$$

where $\|.\|$ is any norm, i.e. only looking at x's with unit length in the process. We will use this restriction of x in the following algorithms, as this avoids exploding values in x and at the same time removes the danger that vectors found by the algorithm slowly approach the 0 vector.
So, assume one chooses a random vector $x \in \mathbb{R}^n_+$ and evaluates $x^T Ax$. If the result is negative, a violating vector has been found in x, else we want to take a step closer towards a violating vector. The question is, in what 'direction' we should take this step. The answer is simply the negative gradient.

Lemma 4.1.1. *The gradient of $x^T Ax$ with respect to x is*

$$\nabla_x x^T Ax = 2Ax \in \mathbb{R}^n.$$

Since we are only interested in the direction to move in and the proportion of the gradient's individual elements, we can ignore the factor 2 as this does not provide any additional information.
While this is a satisfactory result, it would work even better in the scenario of checking for positive semidefiniteness, i.e.,

$$\forall x \in \mathbb{R}^n : x^T Ax \geq 0$$

- in contrast to this, checking for copositivity requires only looking at vectors on the nonnegative orthant. Since the (negative) gradient (obviously) also points in a negative direction for some dimensions, the resulting vector will take on negative values to find a violating vector if not restricted. To avoid this, some countermeasure has to be introduced.

Cut-off values at 0 to avoid negative entries in x

In order to avoid negative values in x, the idea is to use a cut-off function

$$f(x) = \begin{cases} 0, \text{ for } x \leq 0 \\ x, \text{ for } x > 0 \end{cases}$$

that is applied elementwise on x after every update step. While this does indeed keep the vector x nonnegative, the function f is not differentiable in 0 and also restricts the 'movement' of the vector quite drastically.

Normalization of x after update step

After the update step, which is discussed a bit later in this chapter, the vector usually does not fulfill $\|x\| = 1$ anymore, hence it has to be normalized to unit length again before continuing the search for another vector. Therefore, we calculate

$$x_{norm} = \frac{x}{\|x\|}$$

and use x_{norm} as the new vector for further calculations.

4.1.2 Problem formulation 2: $(x^2)^T A(x^2)$

To avoid the necessity of a cut-off function, one idea is to square the elements of x elementwise. This way, negative values are allowed in the original vector x since they become positive anyways in the vector $x^2 = (x \circ x)$ where $\circ$ denotes the Hadamard product, i.e. the pointwise multiplication of two vectors - or in this case the elementwise square of each element of x.

Due to the additional complexity in the formulation, also the calculation of the gradient changes.

Lemma 4.1.2. *The gradient of $(x^2)^T A(x^2)$ with respect to x is*

$$\nabla_x (x^2)^T A(x^2) = 4A(x^2) \circ x \in \mathbb{R}^n$$

Again, as before, the factor 4 can be ignored as it does not contribute additional information in this context.

The big advantage of this problem formulation is, that there is no need for any cut-off function and the vector can move more "freely", but it also comes with some disadvantages of different levels of severity:

- The gradient is more complex than the one of problem formulation 1
- If the steps during the update step are too large, this could lead to oscillations of the vectors, since the overshooting over the 0 point could lead back to the same point again.
- It is not immediately clear how to normalize the vector x.

Especially the third point is tricky. If x has unit length, this usually does not hold for x^2 anymore. While there are several possible workarounds for this issue, all of them seem to have their downsides.

Of course, it is still possible to scale x to unit length - this will lead to smaller values in x^2 but does not break the main functionality of finding a violating vector.

A different approach is the reformulation of the problem to

$$(\frac{x^2}{\|x^2\|})^T A(\frac{x^2}{\|x^2\|})$$

which solves the problem at the cost of a complex gradient.

But: This formulation looks already very similar to a function well known for its important properties in the field of deep learning - the Softmax function.

4.1.3 Problem formulation 3: $(softmax(x))^T A(softmax(x))$

The two things that mainly bother us regarding the overall problem of finding a violating vector are

- The resulting violating vector has to be nonnegative in all entries
- We would like the violating vector to be of unit length

While the second condition was quite easy to achieve in the first formulation, the second case was ensured in the second formulation, but so far one of both conditions was always lacking behind.

One possible solution to this is to use the Softmax function on x instead of squaring it.

Definition 4.1.3. *The Softmax function is defined as*

$$softmax : \mathbb{R}^n \to \mathbb{R}^n : softmax(x) = \frac{e^x}{\|e^x\|_m}$$

where

$$\|.\|_m$$

is the Manhattan norm, i.e. the (absolute) sum of all the entries of some vector.

The consequences are the following:

- $softmax(x)$ will always produce a nonnegative vector since e_i^x only takes on nonnegative values (in fact, only positive values) for any $x_i \in \mathbb{R}$
- Every vector the Softmax function is applied to will have unit length - with respect to the Manhattan norm.

This means, that neither a cut-off function nor some artificial normalization procedure is necessary. The gradient on the other hand is still quite complex for this formulation. But one can avoid the calculation of the analytical gradient by using the stochastic gradient instead.

Remark 4.1.4. *Even though this formulation seems to solve some of the issues, it comes at the cost of a complex gradient and the exponential scaling might hinder the algorithm to converge in the desired manner. Whether this indeed holds true or not will be discussed in Section 5.*

4.2 Update step

So far, the calculation of the gradient was discussed together with the normalization of a vector after the update. This section aims at shedding some light on this update step. The overall idea is the following: Once the gradient for some vector x is determined, a step vector is calculated and the vector x is updated according to this step vector.

In the following, several approaches to calculate this step vector are discussed.

4.2.1 Learning rate

The parameter that determines how large one step per iteration should be, is in the field of deep learning usually called the learning rate, as it determines how 'much' a model learns from one sample - or to put it in other words, how much a model reacts to one sample (or a batch of samples). A high learning rate means that a goal vector can be reached fast, but there is always the risk of overstepping an optimal solution. A too large learning rate will very often not converge to a desired result.
On the other hand, a small learning rate means only taking very small steps in each iteration and then reassessing in what direction to go next. This means that the algorithm converges very slowly, but will eventually arrive at some (local) optimum. But if the learning rate is too small, it is very likely that the optimum will only be a local rather than a global solution.

Fixed learning rate

The most basic idea is to use a constant learning rate, i.e. a learning rate that is the same for all iterations. If chosen properly, one can have the advantages of a high and a

small learning rate at the same time: Convergence in reasonable time together with a certain resistance with respect to local solutions and overshooting a global solution.
The problem is to choose the learning rate properly - but with some rules of thumb and keeping in mind the size of the values that should be optimized one should be able to come to a more or less reasonable result.

Varying learning rate

Instead of having a fixed learning rate for all iterations, one can also vary the learning rate over the course of the iteration. Two heuristics that come to mind are for example

- decrease the learning rate with every iteration a bit more - this way it should be ensured that the model converges to some solution instead of jumping from one promising spot to another.
- Every time the evaluation of $x_{new}^T A x_{new}$ is larger than $x_{old}^T A x_{old}$ (or the respective other formulations), i.e.
$$x_{new}^T A x_{new} > x_{old}^T A x_{old}$$
where x_{new} is the newest vector and x_{old} is the second-newest vector (i.e. x_{new} is an updated version of x_{old}), the learning rate is reduced - halving it could be a good heuristic. This means, that once the algorithm runs out of options to optimize with the old learning rate, the rate is adjusted and the algorithm uses finer steps to approach some solution.

This is by far no exhaustive list of possibilities but just collects two ideas that will be investigated during the experiments.

Update step

Once the learning rate stands for some iteration, the step vector s is calculated. With the step vector at hand, we then have

$$x_{new} = x_{old} - s$$

The reason for the minus in front of s is, that in order to minimize $x^T A x$ (or the respective other formulations), we have to look at the negative gradient instead of the positive one.
What is left to do before calculating x_{new} is to first calculate the step vector s from the gradient and the chosen learning rate.

Straight-forward learning step

The most simple approach is to simply scale the gradient vector,

$$\zeta = \nabla_x x^T Ax$$

(or the respective other formulations) by the learning rate, i.e.

$$s = \zeta * \mu,$$

where μ denotes the learning rate. Since the step vector is only the scaled gradient, this can mean that the step taken in one iteration can be very large or almost zero in many cases - solely depending on the gradient. In many cases this is the desired outcome - close to an optimum the gradient will be very small and we only want to take very small steps, further away from the optimum the gradient will (hopefully!) be larger which will result in a larger step. But sometimes, for example, at a saddle point, this might be misleading for the model and it could end up staying at some local optimum.

'Normalized' gradient vector

Another approach is to 'normalize' the gradient such that its length is equal to the learning rate. This can be accomplished by calculating

$$s = \mu \cdot \frac{\zeta}{\|\zeta\|}$$

Using this step vector, the length of each step is completely determined by the learning rate, the gradient only determines the proportion and direction in which to move for every dimension. The advantage is, that one can very precisely determine how the solution develops over time, the issue on the other hand is that all the power now lies in the learning rate - if a bad rate is chosen, it is unlikely that the algorithm is able to find a good solution.

This concludes the description of all ingredients necessary to build a gradient-based optimization algorithm. In the experiments section 5 numerical results for the different problem formulations together with combinations of the presented methods for choosing the learning rate and calculating the step vector are presented.

An implementation of the different learning rate computations, step vector calculations and the gradient search itself with the three problem formulations can be found in the Appendix.

5 Experiments

In this section, experiments concerned with gradient-based methods can be found. It is organized in the following manner:
First, we will explore the capabilities of the different gradient-based methods on random matrices. After that, experiments on the famous 'Second DIMACS challenge' data set [Dim][JT96] concerned with max-clique detection will be used to see which configuration performs best on the instances of this data set.

5.1 Gradient descent approach evaluation on random matrices

This section is concerned with comparing the different gradient descent approaches together with the several ideas presented in Chapter 4 - in total, 18 different variants are evaluated:

	Problem formulation	Step size method	Step vector calculation method
1	Standard	fixed	simple
2	Square	fixed	simple
3	Softmax	fixed	simple
4	Standard	decay	simple
5	Square	decay	simple
6	Softmax	decay	simple
7	Standard	halving	simple
8	Square	halving	simple
9	Softmax	halving	simple
10	Standard	fixed	norm
11	Square	fixed	norm
12	Softmax	fixed	norm
13	Standard	decay	norm
14	Square	decay	norm
15	Softmax	decay	norm
16	Standard	halving	norm
17	Square	halving	norm
18	Softmax	halving	norm

The reasoning behind this order is to keep different problem formulations using the same step size and the same step vector calculation method together.

Step size ‘fixed‘ refers to a static learning rate, ‘decay‘ means that after each iteration, the learning rate will be decreased by $x\%$ (in this case, by 1%), and ‘halving‘ refers to the case where the learning rate is halved if no improvement in the last step was possible. In the ‘simple‘ step vector calculation, the gradient is multiplied by the learning rate, in the ‘normal‘ case the step vector is the gradient normalized to the length of the learning rate.

The inspiration for the experiment setup (first random matrices, then the evaluation on the DIMACS data set) was drawn from [ZD11]. For a desired matrix of order n, random values are drawn from a uniform distribution over $[-1, 1]$, then the matrix is mirrored to end up with a symmetric matrix, then the diagonal is filled with ones. In the setup of this thesis, the proportion of nonpositive to nonnegative values can be adjusted by skewing the distribution and rescaling afterward. This is necessary since it becomes less and less probable for matrices of this type with growing order to be copositive as discussed in [ZD11]. In the context of this experiment, this means that for each order k of matrices a total of 1.000 instances are generated and evaluated by all 18 gradient methods. The first of these matrices will always have a $1:1$ proportion of negative to positive values, whereas the last matrix will have a $10:1$ proportion to make it more likely to produce more copositive matrices. Also, no preprocessing steps will be applied: The gradient methods have to find a violating vector and hence refute copositivity using only the raw matrices without any ’help’.
We will investigate random matrices of order

$$k \in \{8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 50, 100, 250, 500, 1000\}.$$

Remark 5.1.1. *Even though these countermeasures were taken to avoid all matrices of a certain order k to be not copositive, this did only work until order* 20*. Until this order, the search for a violating vector was made at least ‘tricky‘ enough such that not for all of them a violating vector could be found in the experiment.*

The reason why no matrices of order 7 or less are considered lies in the fact, that [YL09] contains algorithms specifically designed to handle those matrices, hence there is just no need to use this procedure instead. The sequence is cut off at order 1.000 solely due to computational limitations - it simply takes too long to generate and evaluate multiple thousands of matrices of order 10.000 or greater due to the large number of tested algorithms - but the algorithm works fine with any order of matrices, experiments up to order 100.000 were carried out during implementation, showing a similar behavior as on smaller matrices. For reference, it has to be noted that the execution of, e.g., algorithm 11 on a matrix of order 10.000 with a $1:1$ proportion of positive to negative values on an Apple MacBook Pro 2022 (M1) takes on average around 0.3 seconds.

Every method is instantiated with the very same starting vector (which changes with the matrices) and the same matrix will be evaluated by all algorithms to ensure a fair comparison. Also, every method only has one chance - i.e. one random start vector and then 1.000 iterations (i.e. update steps) - in this experiment to find a violating vector. If one of the methods is able to find a violating vector, the matrix is flagged as

not copositive. The individual algorithms are then compared in terms of how many of these not-copositive matrices they were able to identify and what the average number of iterations until finding a solution was - if a violating vector was found. This analysis allows additional insights into how the algorithms would perform if faced with even larger matrices.
In the following, we will have a look at the numerical results for the individual algorithms. To improve visibility, the algorithms are clustered by the problem formulation, which results in displaying 6 algorithms next to each other. The numerical results for the shown plots can be found in the Appendix 7.

5.2 Numerical results

5.2.1 'Standard' problem formulation

We start this section off with the 'Standard' problem formulation - including algorithms $1, 4, 7, 10, 13, 16$.

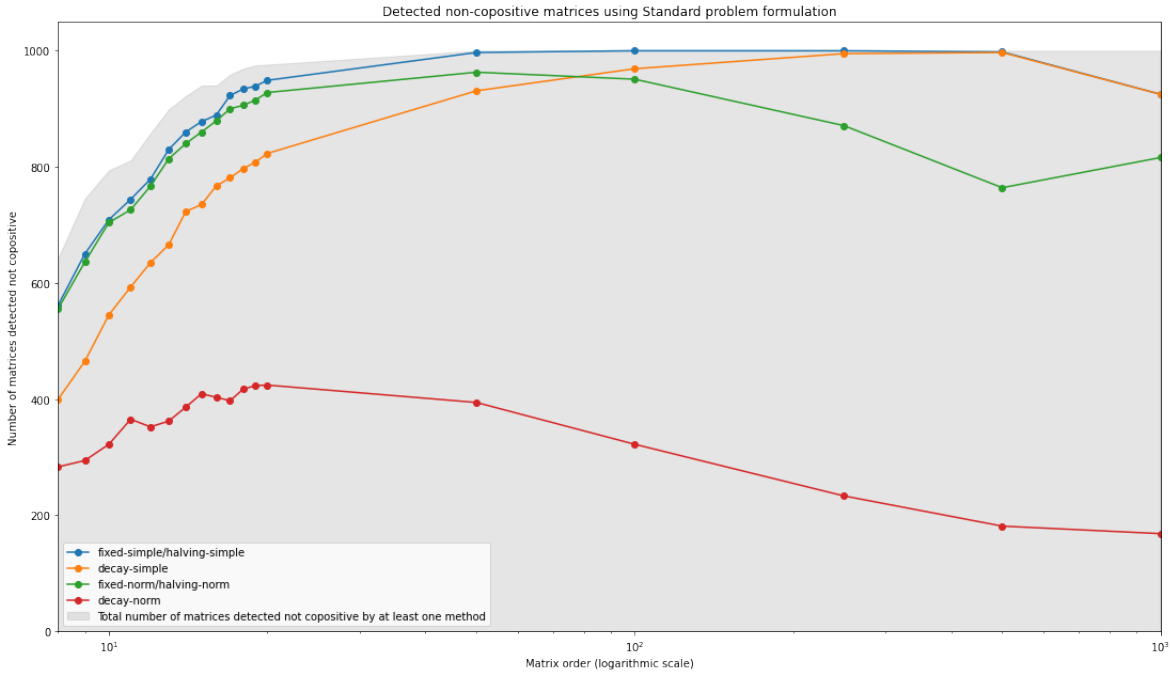

Figure 5.1: Results for experiments with Standard formulation on how many non-copositive matrices were identified

As can be seen, there is no difference between the 'fixed-norm' and 'halving-norm', 'fixed-simple' and 'halving-simple' versions respectively. This is the expected behavior, as the halving of the learning rate would only occur if no improvement with the old learning rate is possible - something that is more likely to occur with 'harder' examples than random matrices. While 'fixed-norm/halving-norm' performs well for smaller matrices, the capabilities seem to drop for matrices of larger order. Version 'decay-simple' on the other hand lacks behind in the beginning and approaches the best variants 'fixed-simple/halving-simple' continuously. These two approaches dominate the others for almost any matrix dimension. The worst variant is 'decay-norm' - which can be explained by the fact that the step size is limited and decreases continuously with every iteration, making it very hard to reach the goal - finding a violating vector - in time if the starting vector is not already close enough to it.

But how many iterations were needed to find the respective violating vectors?

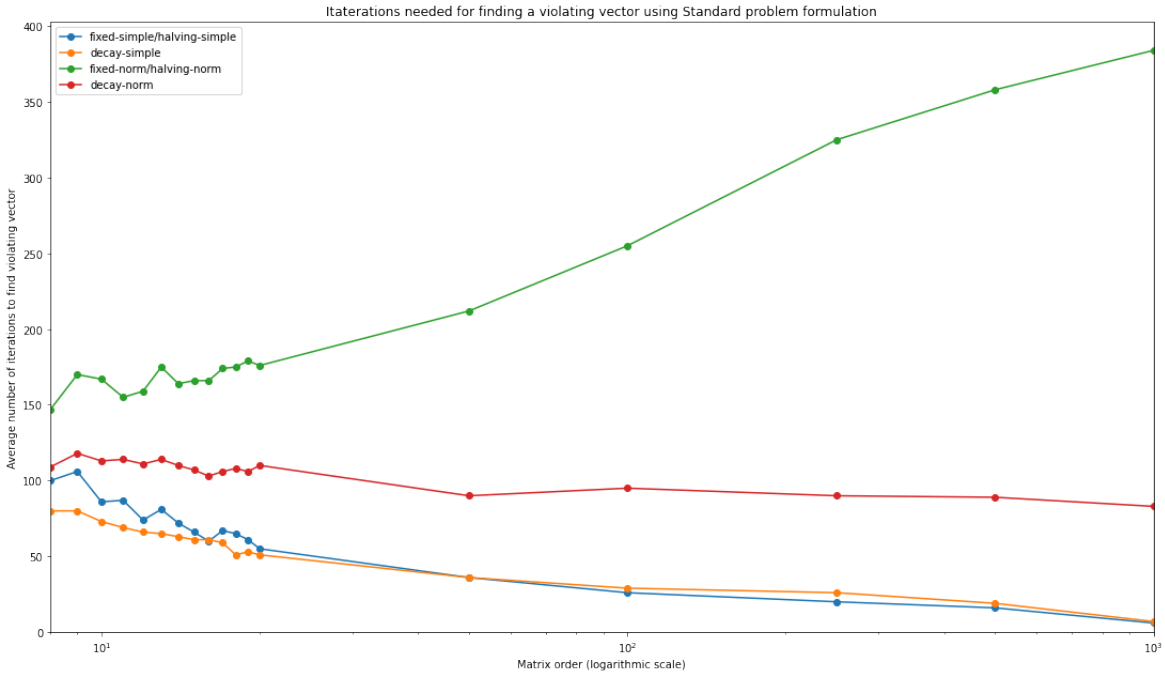

Figure 5.2: Results for experiments with Standard formulation on how many iterations were needed to find a violating vector

By first looking at Figure 5.2 the results might seem counterintuitive - for the variants ‘fixed-simple/halving-simple‘ and ‘decay-simple‘ the average number of iterations actually decreases with larger matrix orders. The reason for this can most likely be found in the fact how the matrices look like. Since the values are sampled uniformly, it simply becomes more and more likely that not just the matrix itself is not copositive but also the number of possible violating vectors increases. And it appears like these variants do a particularly good job of utilizing a good initialization (i.e. a ‘good‘ starting vector) by moving more directly into the promising direction with the right amount of momentum. The comparably poor performance of both versions containing ‘norm‘ can be simply explained by the fact that they are forced to take small(er) steps in each iteration than the other algorithms - with increasing matrix dimension, this leads to the fact that movement in a single direction is massively limited - hence it takes longer to converge overall.

5.2.2 'Square' problem formulation

In Chapter 4 problems with the 'Standard' problem formulation from a theoretical standpoint that could have a negative influence on the overall results were outlined, in the following we have a look at the 'Square' formulation results used in the algorithms $1, 4, 7, 10, 13$ and 16.

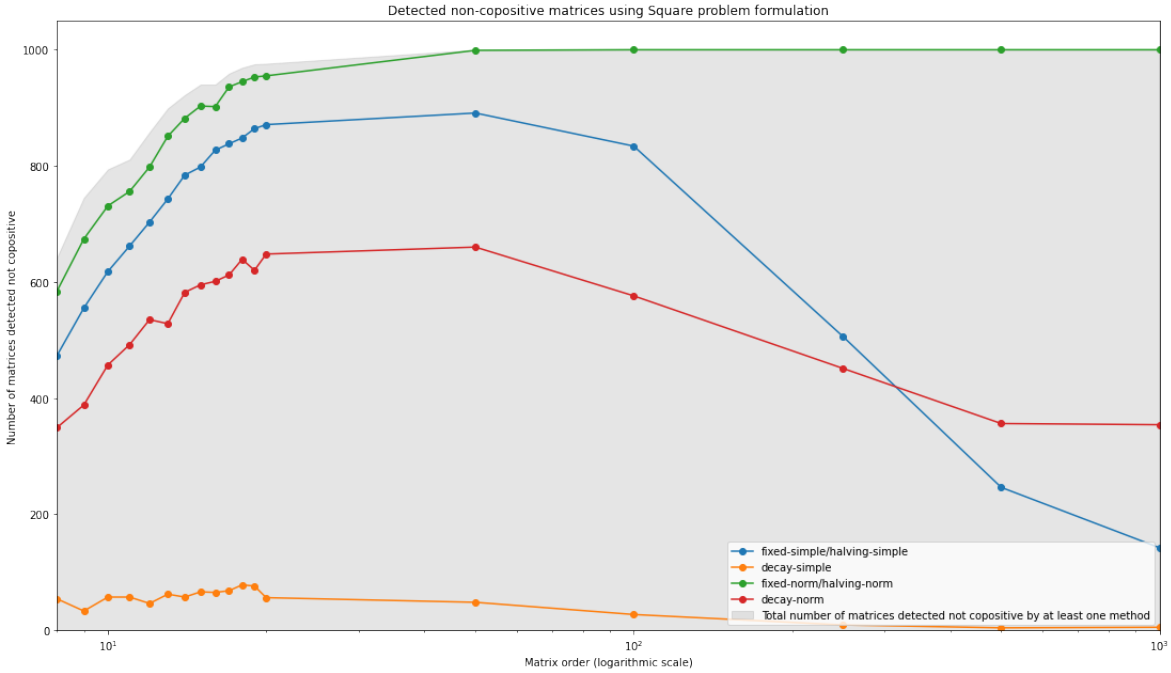

Figure 5.3: Results for experiments with Square formulation on how many non-copositive matrices were identified

In contrast to the 'Simple' formulation, Figure 5.3 offers a very clear 'winner' of the different variants: the 'fixed-norm/halving-norm' variant is clearly the best version for this problem formulation and was able to identify almost every matrix as not copositive from order 50 onwards. For other approaches, performance is either poor from the beginning or drops significantly with larger orders. One reason for this might be that the normalization aspect of the step vector hinders the algorithm from oscillating - a problem that otherwise arises if the step taken during one iteration is too large. It was already mentioned in Section 4.1.2 that this could cause problems in a numerical example.

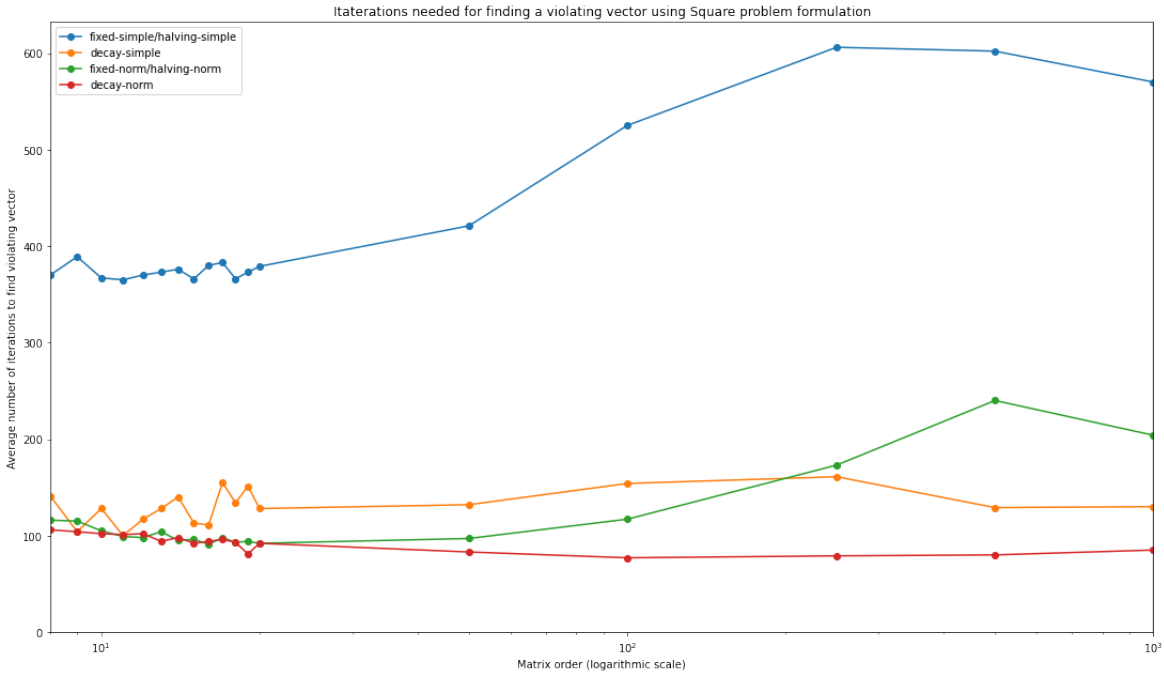

Figure 5.4: Results for experiments with Square formulation on how many iterations were needed to find a violating vector

Talking about the average number of iterations needed to come to a conclusion, there are no massive upsets - all algorithms stay in a certain range of steps needed, where ‘decay-norm‘ is arguably the most stable. But also for the other variants no clear trend that would indicate a drastic increase in average iterations with growing matrix dimensions can be identified. The number of iterations necessary seems to grow linearly with methods ‘fixed-norm/halving-norm‘ (mind that the x-axis has a logarithmic scale) except for the final instances of order 1.000. This could be considered an outlier but it is hard to say for sure since only 1.000 matrices were evaluated each, limiting the overall reliability of the data. But even with a linear increase (with the order of matrices) the algorithm can still be considered ‘good enough‘ considering the general difficulty of this problem.

5.2.3 'Softmax' problem formulation

The third and final problem formulation is explored in this section - from a mathematical perspective, some problems seem to have been solved with this variant, but the trials on actual data are devastating, especially for matrices of larger orders.

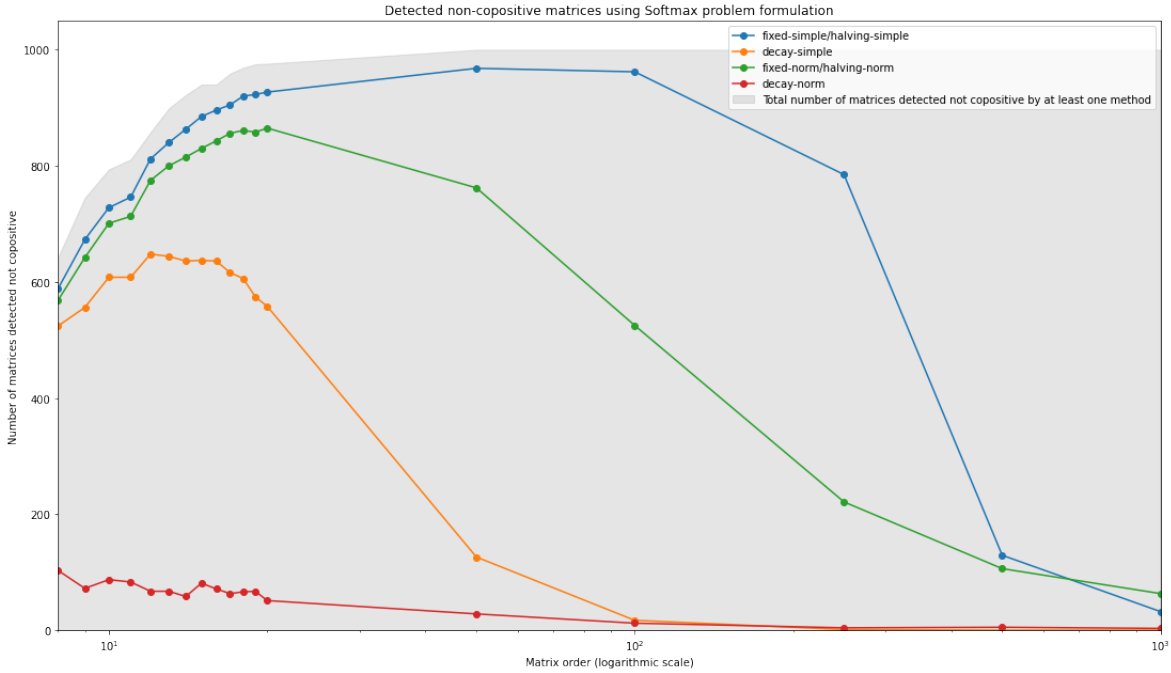

Figure 5.5: Results for experiments with Softmax formulation on how many non-copositive matrices were identified

While in the beginning there are still some variants that are able to perform quite well, starting from order 50 performance starts declining for all of them. For matrices of order 1.000, almost no violating vectors could be found anymore by any variant, rendering this problem formulation useless for general applicability. One explanation why it performs somewhat reasonably until order 20 for most of the presented variants (for 'fixed-simple/halving-simple' even up to 100) but lacks behind for the larger dimensions could lie in the Softmax function itself. Since the exponential function converges very, very slowly towards 0 with decreasing exponential term, the values in the produced vectors are almost 'bound' from below - restricting the 'movement' of the vector towards a potential solution drastically. The problem lies in the fact that with a growing number of dimensions also the solution vector has more and more entries - due to the 'built-in' normalization, the method is only able to converge at extremely slow speed - not being able to identify a violating vector in most cases.
Since for the average iteration count only 'successful' rounds (i.e. those where a violating vector was found) are taken into consideration, the behavior of the individual models

depicted in Figure 5.6 stay almost the same for the different matrix orders.

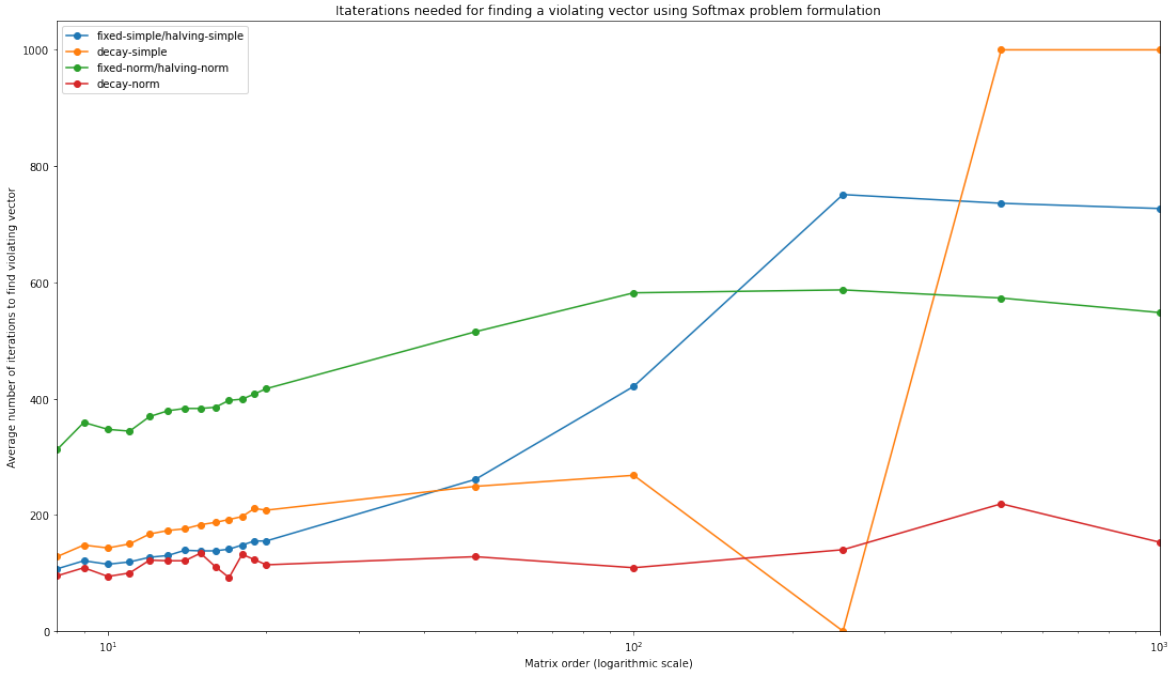

Figure 5.6: Results for experiments with Softmax formulation on how many iterations were needed to find a violating vector

For ‘decay-simple‘ the average number of iterations was set to 1.000 for the orders 500 and 1.000 since no violating vectors were found at all, making it impossible to calculate a mean as well.

In summary, the Softmax variant is definitely inferior to both other formulations in this experiment - most likely an additional regularization term would be necessary to free up the movement of the vector a bit and thereby enhance the searching procedure. The claim still holds that this problem formulation should be capable of finding violating vectors with the same or better performance given the correct step function, an appropriate adjustment of the learning rate, and a fitting regularization term - making this an interesting object to investigate in future work as also discussed in Section 6.1.

5.3 Gradient descent approach evaluation on DIMACS data set

Now that we have established the quality of the algorithms on random instances and proven that several variants of this algorithm are capable of finding violating vectors in the desired time, it remains to be checked how this kind of algorithm performs on the famous DIMACS data set [Dim]. This data set consists of a total of 80 problem instances - the goal is, to use the link between max clique detection and copositivity testing which was introduced at the end of Section 1.3, to approximate the clique number. The experiment is conducted to understand if there is a difference in the different approaches concerning this approximation. This means, that the already introduced 18 algorithms will be competing against each other in the quest to find the correct clique number, i.e. the largest value for which the graph is still provable non-copositive. Every method has up to 100 trials (meaning 100 starting points and again a maximum of 1.000 iterations per run) to find a violating vector for a given matrix. If such a vector is found, the matrix is tested again with the next larger supposed clique number until the method is not able to find a violating vector anymore. The largest found number is then the result for the method on the respective problem instance.

Remark 5.3.1. *The best known estimates for this data set were taken from [PH11]. Strangely enough, for 'C2000.0' [Dim] states the best known result with* 80 *instead of* 78*, so the higher estimate was used - this introduces some doubts about the correctness of the other estimates as well, but other researched paper (which only contain parts of the data set) show smaller or equal estimates for the same instances.*

In the following, the results of the different algorithms per data instance are shown in one plot each to enhance the visibility of how the 18 different variants perform compared to each other. The red line indicates the best known results, the numbering of the algorithms is the same as outlined at the beginning of this chapter. The bar(s) for the best algorithm(s) per data instance are colored in orange to make them more distinguishable from the others.

Figure 5.7: Results for experiments on the DIMACS instances $1 - 16$

From the first 16 plots shown in Figure 5.7, we can already see a pattern in which variants perform well - a trend that will continue in the next figures. Overall, there are only a handful of algorithms that are outstanding in terms of their performance. In some instances like ‘C125.9‘ or ‘MANN_a9‘ the best known result could be achieved by several methods, on the other hand, there are instances like ‘C4000.5‘ where none of the provided algorithms perform particularly well.

Figure 5.8: Results for experiments on the DIMACS instances 17 – 32

Figure 5.8 shows instances with quite promising outcomes: the discrepancy between performances of the procedures 11 and 17 - 'Square' problem formulation with 'fixed' or 'halving' step size method and with 'norm' step vector calculation method - and the others in the instances in the last row are quite drastic. As a reminder: These variants were also the most dominating ones in the random matrix experiments.

Figure 5.9: Results for experiments on the DIMACS instances 33 – 48

The trend continues in Figure 5.9 with the same procedures on top. Instance ‘hamming10-2‘ is especially interesting in this context, since again variants 11 and 17 again outperform the rest of the algorithms. But here the found clique numbers differ by 100 between those two and the next best result.

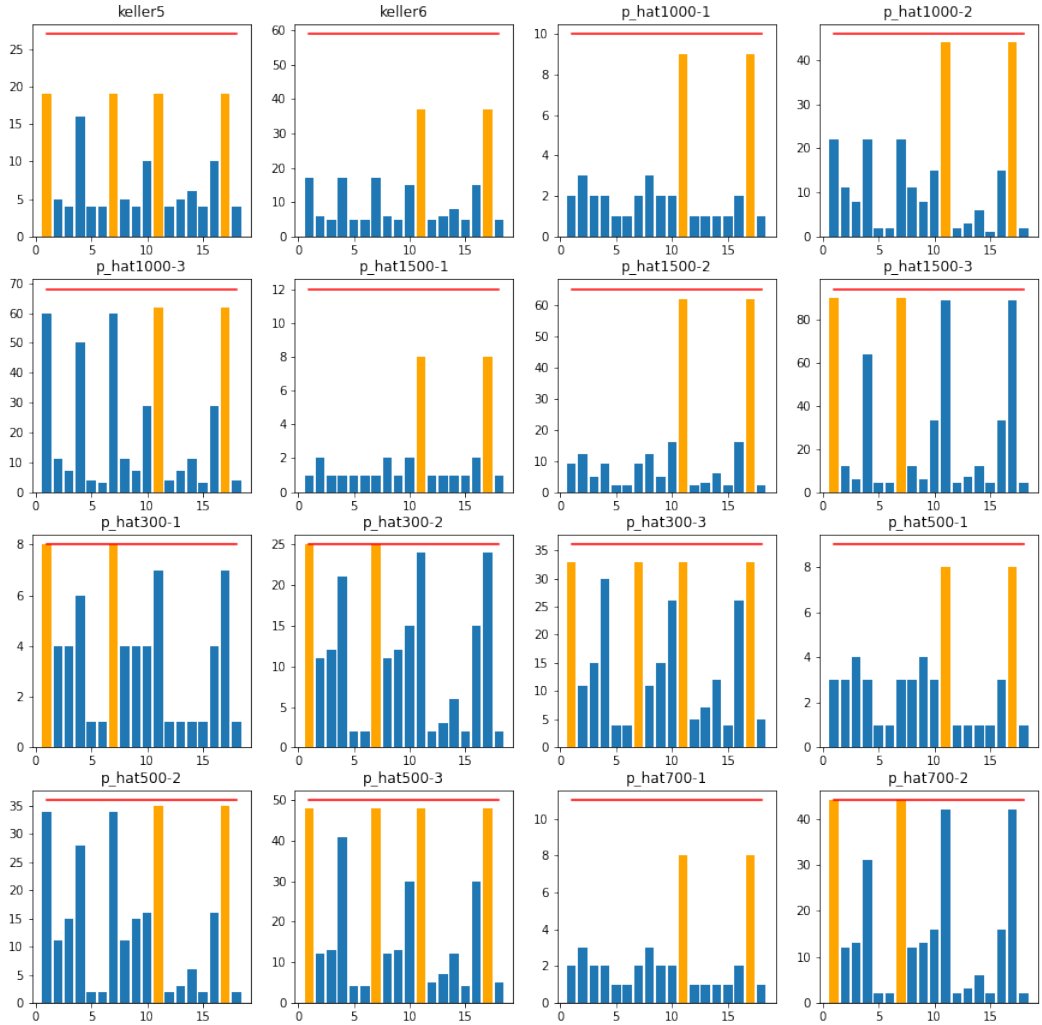

Figure 5.10: Results for experiments on the DIMACS instances 49 – 64

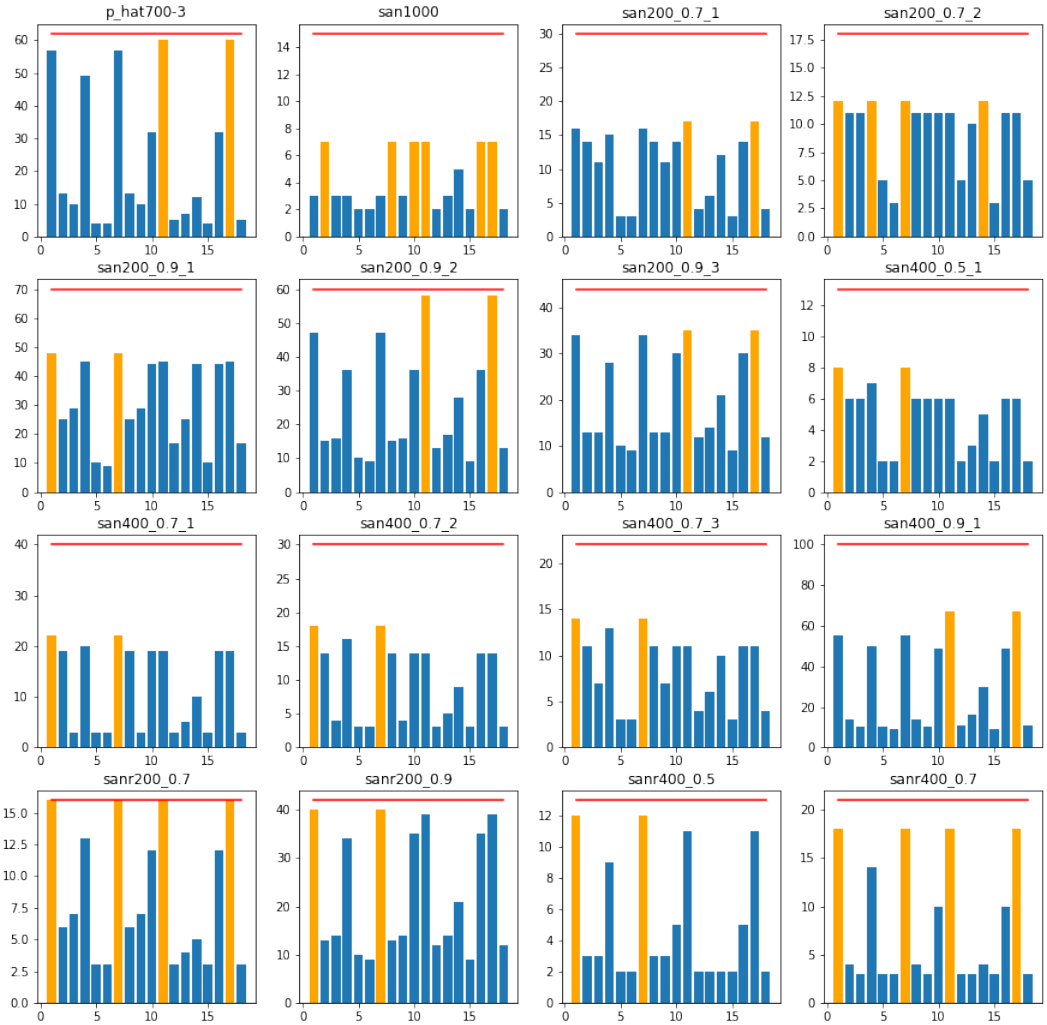

Figure 5.11: Results for experiments on the DIMACS instances $54 - 80$

It is worth mentioning that the versions with the ‘Softmax‘ problem formulation stand no chance against the ‘Standard‘ or the ‘Square‘ formulations - which goes hand in hand with what was observable on the random matrices.

What is left to do is aggregate the findings from these experiments to understand which methods are truly superior.

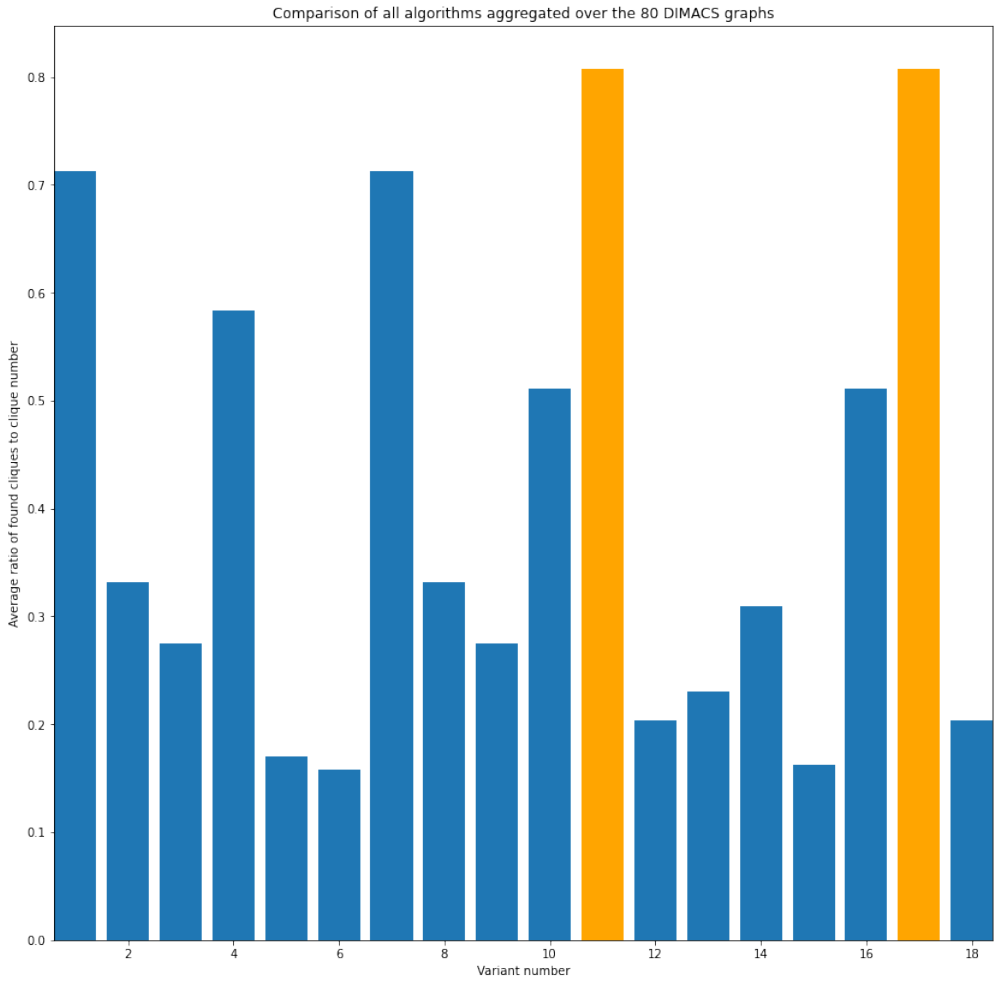

Figure 5.12: Aggregated results over all instances to find best performing methods

As already indicated by the figures above, the best performing methods are indeed numbers 11 and 17 with an average ratio of 80.7% - in fact, there was no difference between the ‘fixed‘ and ‘halving‘ method there, meaning that the learning rate never had to be adjusted to find the optimum. This ratio should be interpreted in the way that the found clique number was on average 20% smaller than the best known result. The best ‘Standard‘ procedure still arrives a bit above 70%, the ‘Softmax‘ procedures stand no chance in the direct comparison.
The result is overall surprisingly good, in 12 of the 80 instances the best known clique number was identified, and also in most other instances at least a handful of the algorithms performed quite well.
The numerical results for all problem instances can be found in the Appendix.

6 Conclusion and future work

6.1 Future Work

The algorithms discussed in this thesis show great potential to speed up copositivity testing on different levels. As already mentioned in Remark 3.2.2, the implementation of the general decomposition algorithm can be optimized in different directions, first and foremost it is crucial for speeding up the algorithm to keep track of which 'parts' of the matrix have already been proven to be copositive and ignoring those in subsequent calculations in order to avoid duplicate operations. Also, the way how the rectangles forming the matrices D_{dec} are chosen and the reordering algorithm that is applied in the case where the entire matrix A^{d^*} is just one large connected component might be reevaluated since changes on this end could lead to overall easier to evaluate matrices after the decomposition step.
One further interesting aspect that should be investigated more in-depth is the case where $(A - D)$ is not copositive, as noted in Remark 3.0.9. If a more general result than the ones presented in this thesis that circle around the case where $B - (1 - \lambda)E$ and $C - \lambda E$ are not copositive for some λ can be found, potentially a conclusion about whether A itself is copositive or not (of course incorporating matrix D as well) could be drawn as well. This would increase the usefulness of the proposed algorithm further.
Concerning the gradient descent approach, the 'Softmax' variant should be investigated more deeply, as from a theoretical standpoint this problem formulation should excel in terms of robustness, even though the experiment results for the current implementation do not reflect this. The next step on this end might be to find a better initialization method to start with, an appropriate step function, and possibly a different approach in terms of learning rate adjustment and experimentation with regularization terms together with a more generous number of iterations per example.
But also the other problem formulations - especially the well-working 'Square' formulation - should be investigated and improved further, as they clearly hold much potential.

6.2 Conclusion

In this thesis, a novel decomposition algorithm was presented that is applicable to any given matrix which allows for reducing the size of matrices that have to be checked and offering a sufficient condition for copositivity of matrices. Moreover, if the special case discussed in Section 2.3 is applicable, a massive speedup while checking for copositivity can be achieved. The extensive discussion of gradient-based search algorithms to find violating vectors yielded very promising results, being able to flag matrices of very large

orders as not copositive within a limited number of iterations, rendering it a very useful tool for quickly finding violating vectors without any overhead in terms of preprocessing or complex implementation. Overall, the results of this thesis can help to speed up copositivity testing on multiple levels and offer great potential for future use.

7 Appendix

7.1 Gradient descent implementation

The following functions were used to carry out the experiment described in Chapter 5.

```
def initialize_start_vector(
    dim: int, vec: torch.Tensor, norm_order: int = 2
) -> torch.Tensor:
    """generates the initial starting vector

    :param dim: defines the size of the array
    :type dim: int
    :param vec: if a vector is already given, it will be used as the
    starting vector, defaults to None
    :type vec: torch.Tensor
    :param norm_order: what order to use for normalization of the
    generated or passed starting vector,
    defaults to 2
    :type norm_order: int, optional
    :return: normalized starting vector
    :rtype: torch.Tensor
    """
    if vec is None:
        start_vector = np.random.uniform(0, 1, (dim)).astype("float")
    else:
        start_vector = vec
    start_vector /= np.linalg.norm(start_vector, ord=norm_order)
    return start_vector
```

Listing 7.1: Initialization of starting vector

```
def postprocessing_vector(vector: torch.Tensor, method: str) -> torch.
    Tensor:
    """applies postprocessing to the vector

    :param vector: the vector that should be postprocessed
    :type vector: torch.Tensor
    :param method: what method should be used for postprocessing
    :type method: str
    :return: the postprocessed vector
    :rtype: torch.Tensor
    """
    if method == "standard":
        vector = torch.clamp(vector, min=0)
        vector /= torch.norm(vector, 2)
        return vector
```

```
    elif method == "square":
        vector /= torch.norm(vector, 2)
        return vector
    elif method == "softmax":
        return vector
```

Listing 7.2: Postprocessing vector after update step

```
def problem_formulation(
    matrix: torch.Tensor, vector: torch.Tensor, formulation: str
) -> torch.Tensor:
    """Defines what problem formulation should be used to calculate the
    result

    :param matrix: a real, symmetric matrix
    :type matrix: torch.Tensor
    :param vector: the current vector
    :type vector: torch.Tensor
    :param formulation: what formulation should be used to calculate the
    result
    :type formulation: str
    :return: Result of the respective problem formulation
    :rtype: torch.Tensor
    """
    if formulation == "standard":
        return torch.inner(torch.inner(vector, matrix), vector)
    elif formulation == "square":
        return torch.inner(torch.inner(vector**2, matrix), vector**2)
    elif formulation == "softmax":
        soft = torch.nn.Softmax(dim=0)
        return torch.inner(torch.inner(soft(vector), matrix), soft(vector
    ))
```

Listing 7.3: Calculating the result based on chosen problem formulation

```
def calc_gradient(
    matrix: torch.Tensor, vector: torch.Tensor, gradient_method: str
) -> torch.Tensor:
    """Calculates the gradient according to the gradient method (problem
    formulation)

    :param matrix: real, symmetric matrix
    :type matrix: torch.Tensor
    :param vector: the current vector
    :type vector: torch.Tensor
    :param gradient_method: gradient method/problem formulation that
    should be used
    :type gradient_method: str
    :return: _description_
    :rtype: torch.Tensor
    """
    if gradient_method == "standard":
        return torch.inner(matrix, vector)
    if gradient_method == "square":
```

```
18          return torch.inner(matrix, vector**2) * vector
19      if gradient_method == "softmax":
20          return vector.grad
```

Listing 7.4: Calculate the analytical gradient for ‘Standard‘ and ‘Square‘ or using stochastic gradient in formulation ‘Softmax‘

```
1  def adjust_learning_rate(
2      learning_rate: float,
3      decay_param: float,
4      result: float,
5      old_result: float,
6      method: str,
7  ) -> float:
8      """function to adjust the learning rate after each iteration
9
10     :param learning_rate: current learning rate
11     :type learning_rate: float
12     :param decay_param: decay parameter indicating how much the learning
       rate
13     should be decayed in the respective mode in one iteration
14     :type decay_param: float
15     :param result: the result from the current iteration
16     :type result: float
17     :param old_result: the previous result
18     :type old_result: float
19     :param method: the method that should be used to adjust the learning
       rate
20     :type method: str
21     :return: new learning rate
22     :rtype: float
23     """
24     if method == "simple":
25         pass
26     elif method == "halving":
27         if result > old_result:
28             learning_rate /= 2
29     elif method == "decay":
30         learning_rate = learning_rate * decay_param
31     return learning_rate
```

Listing 7.5: Method to adjust the learning rate after each iteration

```
1  def grad_descent(
2      matrix: np.ndarray,
3      learning_rate: float = 0.01,
4      max_iterations: int = 500,
5      vec: np.ndarray = None,
6      step_vector_method: str = "simple",
7      learning_rate_adjustment_method: str = "simple",
8      formulation: str = "standard",
9      decay_param: float = 0.99,
10 ) -> Tuple[np.ndarray, bool, float]:
```

```
    """Performs a gradient based search to find a violating vector for
    the given matrix

    :param matrix: real, symmetric matrix
    :type matrix: np.ndarray
    :param learning_rate: starting learning rate, defaults to 0.01
    :type learning_rate: float, optional
    :param max_iterations: maximum number of iterations allowed, after
    that the function will
    terminate, defaults to 500
    :type max_iterations: int, optional
    :param vec: a given (precomputed) starting vector can be passed in
    here, defaults to None
    :type vec: np.ndarray, optional
    :param step_vector_method: indicates the step vector method, defaults
     to "simple"
    :type step_vector_method: str, optional
    :param learning_rate_adjustment_method: indicates the learning rate
    adjustmend method, defaults to "simple"
    :type learning_rate_adjustment_method: str, optional
    :param formulation: what problem formulation to use, defaults to "
    standard"
    :type formulation: str, optional
    :param decay_param: the decay parameter for the learning rate
    adjustment method "decay", defaults to 0.99
    :type decay_param: float, optional
    :return:
    :rtype: _type_
    """
    dim = matrix.shape[0]
    start_vector = initialize_start_vector(dim, vec)

    tensor_matrix = torch.tensor(matrix).float().requires_grad_(False)
    tensor_vector = torch.tensor(start_vector).float().requires_grad_(
    True)

    result = 1
    counter = 0
    old_result = 99999999
    while result > 0 and counter < max_iterations:
        res = problem_formulation(tensor_matrix, tensor_vector,
    formulation)
        res.backward()
        result = res.item()
        if result < 0:
            break
        grad = calc_gradient(tensor_matrix, tensor_vector, formulation).
    detach()
        tensor_vector.requires_grad_(False)
        tensor_vector += calc_step_vector(grad, learning_rate,
    step_vector_method)
        tensor_vector = postprocessing_vector(tensor_vector, formulation)
        tensor_vector.requires_grad_(True)
```

```
53          counter += 1
54          learning_rate = adjust_learning_rate(
55              learning_rate,
56              decay_param,
57              result,
58              old_result,
59              learning_rate_adjustment_method,
60          )
61          oldresult = result
62      res_vector = tensor_vector.detach().numpy().astype("float32")
63      return res_vector, result < 0, counter, result
```

Listing 7.6: Main function that incorporates the other functions to find a violating vector for the given matrix

For a given matrix A, the function can be run for example with the following call:

```
1 grad_descent(
2           MATRIX,
3           learning_rate=0.01,
4           max_iterations=500,
5           formulation="standard",
6           learning_rate_adjustment_method="simple",
7           step_vector_method="simple",
8       )
```

Listing 7.7: Main function that incorporates the other functions to find

The stochastic gradient in the ‘Softmax‘ setting can be calculated easily within the ‘Pytorch‘ [Pas+19] framework, that’s why the entire gradient descent routine is written using this module rather than using ‘Numpy‘ - even though an implementation in ‘Numpy‘ for formulations ‘Standard‘ and ‘Square‘ are very easy to achieve and would look very similar to the here presented code.

7.2 Random matrix experiment

In the following, the numerical results for the random matrix experiments are listed. On the leftmost column, the matrix order is listed, followed by the number of matrices that have been identified as not copositive by at least one of the 18 algorithms. A column ‘$\#\ x$‘ contains the number of matrices for which method x was able to find a violating vector, ‘$\emptyset\ x$‘ indicates the average number of iterations necessary to find the violating vector (only cases where the algorithm was successful are considered and the result is rounded to integers).

Random matrix experiment results for methods 0, 1 and 2							
n	# not copositive	# 0	ϕ 0	# 1	ϕ 1	# 2	ϕ 2
8	642	561	100	473	370	588	107
9	745	650	106	555	389	673	121
10	794	709	86	618	367	728	115
11	811	744	87	662	365	746	119
12	858	779	74	703	370	812	127
13	899	830	81	743	373	840	130
14	922	860	72	784	376	863	139
15	940	878	66	798	366	885	138
16	940	889	60	827	380	896	138
17	959	923	67	838	383	905	141
18	969	934	65	848	366	920	148
19	975	939	61	864	373	923	155
20	976	949	55	871	379	927	155
50	1000	997	36	891	421	968	261
100	1000	1000	26	834	525	962	421
250	1000	1000	20	506	606	785	751
500	1000	998	16	246	602	129	736
1000	1000	925	6	142	570	32	727

Table 7.1: Random matrix experiment results for methods 1, 2 and 3

Random matrix experiment results for methods 3, 4 and 5							
n	# not copositive	# 3	ϕ 3	# 4	ϕ 4	# 5	ϕ 5
8	642	399	80	54	141	524	128
9	745	465	80	33	104	556	148
10	794	545	73	57	128	608	143
11	811	593	69	57	100	608	150
12	858	635	66	46	117	648	167
13	899	666	65	62	128	644	173
14	922	723	63	57	140	636	176
15	940	735	61	66	113	637	183
16	940	767	61	65	111	636	187
17	959	781	59	68	155	616	192
18	969	797	51	78	134	606	197
19	975	808	53	76	151	574	211
20	976	823	51	56	128	558	208
50	1000	931	36	48	132	126	249
100	1000	969	29	27	154	17	268
250	1000	995	26	9	161	1	0
500	1000	997	19	4	129	0	1000
1000	1000	925	7	5	130	0	1000

Table 7.2: Random matrix experiment results for methods 4, 5 and 6

Random matrix experiment results for methods 6, 7 and 8							
n	# not copositive	# 6	ϕ 6	# 7	ϕ 7	# 8	ϕ 8
8	642	561	100	473	370	588	107
9	745	650	106	555	389	673	121
10	794	709	86	618	367	728	115
11	811	744	87	662	365	746	119
12	858	779	74	703	370	812	127
13	899	830	81	743	373	840	130
14	922	860	72	784	376	863	139
15	940	878	66	798	366	885	138
16	940	889	60	827	380	896	138
17	959	923	67	838	383	905	141
18	969	934	65	848	366	920	148
19	975	939	61	864	373	923	155
20	976	949	55	871	379	927	155
50	1000	997	36	891	421	968	261
100	1000	1000	26	834	525	962	421
250	1000	1000	20	506	606	785	751
500	1000	998	16	246	602	129	736
1000	1000	925	6	142	570	32	727

Table 7.3: Random matrix experiment results for methods 7, 8 and 9

Random matrix experiment results for methods 9, 10 and 11							
n	# not copositive	# 9	$\emptyset$ 9	# 10	$\emptyset$ 10	# 11	$\emptyset$ 11
8	642	555	147	584	116	568	312
9	745	636	170	674	115	642	359
10	794	704	167	731	105	701	347
11	811	726	155	756	99	713	344
12	858	767	159	798	98	775	369
13	899	814	175	851	104	800	379
14	922	840	164	882	95	815	383
15	940	860	166	903	96	830	383
16	940	879	166	902	91	843	385
17	959	900	174	936	98	856	397
18	969	906	175	945	93	861	399
19	975	915	179	953	94	858	408
20	976	928	176	955	92	865	417
50	1000	963	212	999	97	762	515
100	1000	951	255	1000	117	525	582
250	1000	871	325	1000	173	221	587
500	1000	764	358	1000	240	106	573
1000	1000	816	384	1000	204	63	548

Table 7.4: Random matrix experiment results for methods 10, 11 and 12

Random matrix experiment results for methods 12, 13 and 14							
n	# not copositive	# 12	$\emptyset$ 12	# 13	$\emptyset$ 13	# 14	$\emptyset$ 14
8	642	283	109	349	106	103	95
9	745	294	118	388	104	72	109
10	794	322	113	457	102	87	94
11	811	365	114	492	101	83	100
12	858	352	111	535	102	67	122
13	899	362	114	528	94	67	121
14	922	386	110	582	98	58	121
15	940	409	107	595	92	81	134
16	940	403	103	601	94	71	111
17	959	397	106	612	96	63	92
18	969	417	108	639	93	66	132
19	975	423	106	620	81	67	123
20	976	424	110	648	92	51	114
50	1000	394	90	660	83	28	128
100	1000	322	95	576	77	12	109
250	1000	233	90	451	79	4	140
500	1000	181	89	356	80	5	219
1000	1000	168	83	354	85	3	153

Table 7.5: Random matrix experiment results for methods 13, 14 and 15

Random matrix experiment results for methods 15, 16 and 17							
n	# not copositive	# 15	$\emptyset$ 15	# 16	$\emptyset$ 16	# 17	$\emptyset$ 17
8	642	555	147	584	116	568	312
9	745	636	170	674	115	642	359
10	794	704	167	731	105	701	347
11	811	726	155	756	99	713	344
12	858	767	159	798	98	775	369
13	899	814	175	851	104	800	379
14	922	840	164	882	95	815	383
15	940	860	166	903	96	830	383
16	940	879	166	902	91	843	385
17	959	900	174	936	98	856	397
18	969	906	175	945	93	861	399
19	975	915	179	953	94	858	408
20	976	928	176	955	92	865	417
50	1000	963	212	999	97	762	515
100	1000	951	255	1000	117	525	582
250	1000	871	325	1000	173	221	587
500	1000	764	358	1000	240	106	573
1000	1000	816	384	1000	204	63	548

Table 7.6: Random matrix experiment results for methods 16, 17 and 18

7.3 DIMACS experiment

Lastly, the the results for the DIMACS challenge can be found in the following two tables. The first column holds the instance name and the second the respective best known result. In the columns after that the results for the different methods are listed.

DIMACS experiment results for methods 1-9										
instance name	best known estimate	1	2	3	4	5	6	7	8	9
C1000.9	68	57	10	10	44	10	10	57	10	10
C125.9	34	34	15	17	28	10	9	34	15	17
C2000.5	16	2	2	2	2	2	1	2	2	2
C2000.9	80	62	10	10	47	10	9	62	10	10
C250.9	44	38	12	13	32	10	9	38	12	13
C4000.5	18	2	2	2	2	2	2	2	2	2
C500.9	57	49	11	10	38	10	9	49	11	10
DSJC1000_5	15	3	2	2	3	2	1	3	2	2
DSJC500_5	13	12	3	2	8	2	2	12	3	2
MANN_a27	126	118	88	93	117	78	80	118	88	93
MANN_a45	345	330	221	226	329	204	215	330	221	226
MANN_a81	1100	1080	673	685	1079	637	678	1080	673	685
MANN_a9	16	16	12	11	14	11	10	16	12	11
brock200_1	21	19	7	8	15	4	3	19	7	8
brock200_2	12	9	4	4	8	2	1	9	4	4
brock200_3	15	12	5	5	10	2	2	12	5	5
brock200_4	17	14	6	6	12	3	2	14	6	6
brock400_1	27	21	5	4	16	4	3	21	5	4
brock400_2	29	21	5	4	16	4	3	21	5	4
brock400_3	31	21	5	4	17	4	3	21	5	4
brock400_4	33	21	5	4	17	4	3	21	5	4
brock800_1	23	17	3	2	12	2	2	17	3	2
brock800_2	24	17	3	2	11	2	2	17	3	2
brock800_3	25	17	3	2	11	2	2	17	3	2
brock800_4	26	17	3	2	11	2	2	17	3	2
c-fat200-1	12	12	9	2	10	1	1	12	9	2
c-fat200-2	24	24	17	7	18	1	1	24	17	7
c-fat200-5	58	58	42	19	48	2	1	58	42	19
c-fat500-1	14	1	2	1	1	1	1	1	2	1
c-fat500-10	126	3	75	2	3	1	1	3	75	2
c-fat500-2	26	1	8	1	1	1	1	1	8	1
c-fat500-5	64	1	35	1	1	1	1	1	35	1
gen200_p0.9_44	44	36	14	16	30	10	9	36	14	16
gen200_p0.9_55	55	46	14	17	33	10	9	46	14	17
gen400_p0.9_55	55	45	13	12	37	10	9	45	13	12
gen400_p0.9_65	65	45	12	12	37	10	9	45	12	12
gen400_p0.9_75	75	46	14	12	40	10	9	46	14	12
hamming10-2	512	384	91	93	279	89	93	384	91	93
hamming10-4	40	32	6	5	25	5	5	32	6	5
hamming6-2	32	32	16	19	29	9	9	32	16	19
hamming6-4	4	4	3	3	3	1	1	4	3	3
hamming8-2	128	127	30	28	98	27	28	127	30	28
hamming8-4	16	16	4	2	10	2	2	16	4	2
johnson16-2-4	8	8	4	4	6	4	4	8	4	4
johnson32-2-4	16	16	8	8	9	8	8	16	8	8
johnson8-2-4	4	4	3	3	3	2	2	4	3	3
johnson8-4-4	14	14	6	6	12	4	4	14	6	6
keller4	11	9	4	5	8	3	2	9	4	5
keller5	27	19	5	4	16	4	4	19	5	4
keller6	59	17	6	5	17	5	5	17	6	5
p_hat1000-1	10	2	3	2	2	1	1	2	3	2
p_hat1000-2	46	22	11	8	22	2	2	22	11	8
p_hat1000-3	68	60	11	7	50	4	3	60	11	7
p_hat1500-1	12	1	2	1	1	1	1	1	2	1
p_hat1500-2	65	9	12	5	9	2	2	9	12	5
p_hat1500-3	94	90	12	6	64	4	4	90	12	6
p_hat300-1	8	8	4	4	6	1	1	8	4	4
p_hat300-2	25	25	11	12	21	2	2	25	11	12
p_hat300-3	36	33	11	15	30	4	4	33	11	15
p_hat500-1	9	3	3	4	3	1	1	3	3	4
p_hat500-2	36	34	11	15	28	2	2	34	11	15
p_hat500-3	50	48	12	13	41	4	4	48	12	13
p_hat700-1	11	2	3	2	2	1	1	2	3	2
p_hat700-2	44	44	12	13	31	2	2	44	12	13
p_hat700-3	62	57	13	10	49	4	4	57	13	10
san1000	15	3	7	3	3	2	2	3	7	3
san200_0.7_1	30	16	14	11	15	3	3	16	14	11
san200_0.7_2	18	12	11	11	12	5	3	12	11	11
san200_0.9_1	70	48	25	29	45	10	9	48	25	29
san200_0.9_2	60	47	15	16	36	10	9	47	15	16
san200_0.9_3	44	34	13	13	28	10	9	34	13	13
san400_0.5_1	13	8	6	6	7	2	2	8	6	6
san400_0.7_1	40	22	19	3	20	3	3	22	19	3
san400_0.7_2	30	18	14	4	16	3	3	18	14	4
san400_0.7_3	22	14	11	7	13	3	3	14	11	7
san400_0.9_1	100	55	14	10	50	10	9	55	14	10
sanr200_0.7	16	16	6	7	13	3	3	16	6	7
sanr200_0.9	42	40	13	14	34	10	9	40	13	14
sanr400_0.5	13	12	3	3	9	2	2	12	3	3
sanr400_0.7	21	18	4	3	14	3	3	18	4	3

Table 7.7: DIMACS experiment results for methods 1-9

DIMACS experiment results for methods 10-18										
instance name	best known estimate	10	11	12	13	14	15	16	17	18
C1000.9	68	32	58	10	12	15	10	32	58	10
C125.9	34	31	33	14	16	24	9	31	33	14
C2000.5	16	2	6	2	2	2	2	2	6	2
C2000.9	80	24	63	10	11	13	10	24	63	10
C250.9	44	33	39	12	14	20	9	33	39	12
C4000.5	18	2	4	2	2	2	2	2	4	2
C500.9	57	36	49	11	13	18	9	36	49	11
DSJC1000_5	15	3	10	2	2	2	2	3	10	2
DSJC500_5	13	4	11	2	2	2	2	4	11	2
MANN_a27	126	117	117	109	116	118	84	117	117	109
MANN_a45	345	330	329	293	324	330	225	330	329	293
MANN_a81	1100	1079	1079	901	1047	1080	704	1079	1079	901
MANN_a9	16	15	15	12	14	16	11	15	15	12
brock200_1	21	14	18	4	5	7	3	14	18	4
brock200_2	12	6	9	2	2	2	1	6	9	2
brock200_3	15	8	11	2	3	3	2	8	11	2
brock200_4	17	12	15	3	3	4	2	12	15	3
brock400_1	27	12	22	4	4	6	3	12	22	4
brock400_2	29	12	22	4	4	5	3	12	22	4
brock400_3	31	13	22	4	4	5	3	13	22	4
brock400_4	33	12	22	4	4	5	3	12	22	4
brock800_1	23	6	17	2	3	3	2	6	17	2
brock800_2	24	6	18	2	3	3	2	6	18	2
brock800_3	25	6	17	2	3	3	2	6	17	2
brock800_4	26	6	17	2	3	3	2	6	17	2
c-fat200-1	12	6	11	1	1	1	1	6	11	1
c-fat200-2	24	13	23	1	1	1	1	13	23	1
c-fat200-5	58	31	57	2	2	4	1	31	57	2
c-fat500-1	14	2	3	1	1	1	1	2	3	1
c-fat500-10	126	42	125	1	2	2	1	42	125	1
c-fat500-2	26	3	25	1	1	1	1	3	25	1
c-fat500-5	64	11	63	1	1	1	1	11	63	1
gen200_p0.9_44	44	33	37	13	16	22	9	33	37	13
gen200_p0.9_55	55	34	54	14	16	23	9	34	54	14
gen400_p0.9_55	55	36	46	12	14	21	10	36	46	12
gen400_p0.9_65	65	36	47	11	14	21	9	36	47	11
gen400_p0.9_75	75	38	50	12	16	25	10	38	50	12
hamming10-2	512	325	511	99	137	204	93	325	511	99
hamming10-4	40	14	33	6	7	8	5	14	33	6
hamming6-2	32	31	31	18	17	28	9	31	31	18
hamming6-4	4	3	4	2	1	2	1	3	4	2
hamming8-2	128	113	127	36	46	72	28	113	127	36
hamming8-4	16	7	15	3	3	4	2	7	15	3
johnson16-2-4	8	7	7	4	4	5	4	7	7	4
johnson32-2-4	16	12	15	8	8	9	8	12	15	8
johnson8-2-4	4	3	3	3	2	3	2	3	3	3
johnson8-4-4	14	13	13	6	5	7	4	13	13	6
keller4	11	7	10	3	3	4	2	7	10	3
keller5	27	10	19	4	5	6	4	10	19	4
keller6	59	15	37	5	6	8	5	15	37	5
p_hat1000-1	10	2	9	1	1	1	1	2	9	1
p_hat1000-2	46	15	44	2	3	6	1	15	44	2
p_hat1000-3	68	29	62	4	7	11	3	29	62	4
p_hat1500-1	12	2	8	1	1	1	1	2	8	1
p_hat1500-2	65	16	62	2	3	6	2	16	62	2
p_hat1500-3	94	33	89	4	7	12	4	33	89	4
p_hat300-1	8	4	7	1	1	1	1	4	7	1
p_hat300-2	25	15	24	2	3	6	2	15	24	2
p_hat300-3	36	26	33	5	7	12	4	26	33	5
p_hat500-1	9	3	8	1	1	1	1	3	8	1
p_hat500-2	36	16	35	2	3	6	2	16	35	2
p_hat500-3	50	30	48	5	7	12	4	30	48	5
p_hat700-1	11	2	8	1	1	1	1	2	8	1
p_hat700-2	44	16	42	2	3	6	2	16	42	2
p_hat700-3	62	32	60	5	7	12	4	32	60	5
san1000	15	7	7	2	3	5	2	7	7	2
san200_0.7_1	30	14	17	4	6	12	3	14	17	4
san200_0.7_2	18	11	11	5	10	12	3	11	11	5
san200_0.9_1	70	44	45	17	25	44	10	44	45	17
san200_0.9_2	60	36	58	13	17	28	9	36	58	13
san200_0.9_3	44	30	35	12	14	21	9	30	35	12
san400_0.5_1	13	6	6	2	3	5	2	6	6	2
san400_0.7_1	40	19	19	3	5	10	3	19	19	3
san400_0.7_2	30	14	14	3	5	9	3	14	14	3
san400_0.7_3	22	11	11	4	6	10	3	11	11	4
san400_0.9_1	100	49	67	11	16	30	9	49	67	11
sanr200_0.7	16	12	16	3	4	5	3	12	16	3
sanr200_0.9	42	35	39	12	14	21	9	35	39	12
sanr400_0.5	13	5	11	2	2	2	2	5	11	2
sanr400_0.7	21	10	18	3	3	4	3	10	18	3

Table 7.8: DIMACS experiment results for methods 10-18

Bibliography

[BE10] Immanuel Bomze and Gabriele Eichfelder. "Copositivity detection by difference-of-convex decomposition and ω-subdivision". In: *Mathematical Programming* 138 (Feb. 2010).

[Bom00] Immanuel Bomze. "Linear-time copositivity detection for tridiagonal matrices and extension to block-tridiagonality". In: *Siam Journal on Matrix Analysis and Applications* 21 (Mar. 2000).

[Bom08] Immanuel Bomze. "Perron–Frobenius property of copositive matrices, and a block copositivity criterion". In: *Linear Algebra and its Applications* 429 (July 2008), pp. 68–71.

[Bom96] Immanuel M. Bomze. "Block pivoting and shortcut strategies for detecting copositivity". In: *Linear Algebra and its Applications* 248 (1996), pp. 161–184.

[BP05] Immanuel Bomze and Laura Palagi. "Quartic Formulation of Standard Quadratic Optimization Problems". In: *Journal of Global Optimization* 32 (June 2005), pp. 181–205.

[Bre22] Sjoerd van Bree. "Testing copositivity in pentadiagonal matrices". Bachelor thesis. University of Twente, 2022.

[BSU12] Immanuel Bomze, Werner Schachinger and Gabriele Uchida. "Think co(mpletely)positive! Matrix properties, examples and a clustered bibliography on copositive optimization". In: *Journal of Global Optimization* 52 (Mar. 2012), pp. 423–445.

[CG80] W. Chan and Alan George. "A linear time implementation of the reverse Cuthill-McKee algorithm." In: *BIT* 20 (Mar. 1980), pp. 8–14.

[CHL67] Richard Cottle, G. Habetler and C. Lemke. "Quadratic forms semi-definite over convex cones". In: *Proceedings of the Princeton Symposium on Mathematical Programming* (Aug. 1967), p. 28.

[CHL70] R.W. Cottle, G.J. Habetler and C.E. Lemke. "On classes of copositive matrices". In: *Linear Algebra and its Applications* 3 (July 1970), pp. 295–310.

[CM69] E.H. Cuthill and J. McKee. "Reducing the bandwith of sparse symmetric matrices". In: *ACM Proceedings of the 1969 24th national conference* (Jan. 1969), pp. 157–172.

[CMS95] Jean-Pierre Crouzeix, Juan Martínez-Legaz and Alberto Seeger. "An alternative theorem for quadratic forms and extensions". In: *Linear Algebra and its Applications* 215 (Jan. 1995), pp. 121–134.

[Dia62] P.H. Diananda. "On nonnegative forms in real variables some or all of which are non-negative". In: *Proc. Cambridge Philos. Soc.* 58 (1962), pp. 17–25.

[Dic19] Peter Dickinson. "A new certificate for copositivity". In: *Linear Algebra and its Applications* 569 (May 2019).

[Dim] Dimacs. *DIMACS benchmark set*. URL: `https://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark` (visited on 26/05/2023).

[Dos] Ronak Doshi. *Cuthill-McKee Algorithm*. URL: `https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm` (visited on 12/05/2023).

[Had83] K.P. Hadeler. "On copositive matrices". In: *Linear Algebra and its Applications* 49 (1983), pp. 79–89.

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[HJ13] Roger A. Horn and Charles R. Johnson. *Matrix Analysis, 2nd edition*. Cambridge University Press, New York, 2013.

[HN63] M. Hall and M. Newman. "Copositive and completely positive quadratic forms". In: *Proc. Cambridge Philos. Soc.* 59 (1963), pp. 329–339.

[HS10] Jean-Baptiste Hiriart-Urruty and Alberto Seeger. "A variational approach to copositive matrices". In: *SIAM Review* 52 (Jan. 2010), pp. 593–629.

[Ikr02] K.D. Ikramov. "Linear-time algorithm for verifying the copositivity of an acyclic matrix". In: *Computational Mathematics and Mathematical Physics* 42 (Dec. 2002), pp. 1701–1703.

[Jac76] D. Jacobson. "A generalization of Finsler's theorem for quadratic inequalities and equalities". In: *Quaestiones Mathematicae* 1 (Jan. 1976), pp. 19–28.

[JR09] Charles Johnson and Robert Reams. "Scaling of symmetric matrices by positive diagonal congruence". In: *Linear & Multilinear Algebra* 57 (Mar. 2009), pp. 123–140.

[JT96] David J. Johnson and Michael A. Trick. "Cliques, coloring, and satisfiability: Second DIMACS implementation challenge, workshop, October 11-13, 1993". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1996).

[Kap00] Wilfred Kaplan. "A test for copositive matrices". In: *Linear Algebra and Its Applications* 313 (July 2000), pp. 203–206.

[KP02] Etienne Klerk and Dmitrii Pasechnik. "Approximation of the stability number of a graph via copositive programming". In: *SIAM Journal on Optimization* 12 (Apr. 2002), pp. 875–892.

[Mot52] Theodore S. Motzkin. "Copositive quadratic forms". In: *National Bureau of Standards Report* 1818 (1952), pp. 11–22.

[Mur88] K.G. Murty. *Linear Complementarity, Linear and Nonlinear Programming*. Heldermann Verlag, Berlin, Jan. 1988.

[Pas+19] Adam Paszke et al. “PyTorch: An imperative style, high-performance deep learning library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[PH11] Wayne Pullan and Holger Hoos. “Dynamic local search for the maximum clique problem”. In: *Journal of Artificial Intelligence Research* 25 (Sept. 2011).

[Väl86] Hannu Väliaho. “Criteria for copositive matrices”. In: *Linear Algebra and its Applications* 81 (Sept. 1986), pp. 19–34.

[Vir+20] Pauli Virtanen et al. “SciPy 1.0: Fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272.

[Vli11] Willemieke van Vliet. “Copositive plus matrices”. Master thesis. University of Groningen, 2011.

[YL09] Shang-jun Yang and Xiao-xin Li. “Algorithms for determining the copositivity of a given symmetric matrix”. In: *Linear Algebra and Its Applications* 430 (Jan. 2009), pp. 609–618.

[Yua90] Ya-xiang Yuan. “On a subproblem of trust region algorithms for constrained optimization”. In: *Math. Program.* 47 (May 1990), pp. 53–63.

[ZD11] Julius Zilinskas and Mirjam Dür. “Depth-first simplicial partition for copositivity detection, with an application to MaxClique”. In: *Optimization Methods & Software* 26 (June 2011), pp. 499–510.