A cutting plane approach to upper bounds on the kissing number

Bovengrenzen aan het kusgetal via lineaire snijvlakken

N. Riemens

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

TU Delft

A cutting plane approach to upper bounds on the kissing number

Bovengrenzen aan het kusgetal via lineaire snijvlakken

by

N. Riemens

to obtain the degree Bachelor of Science in Applied Mathematics
at the Delft University of Technology,
to be defended publicly on Tuesday August 22, 2023 at 13:30 PM.

Student number: 5163218
Project duration: April 24, 2023 – August 22, 2023
Thesis committee: Dr. F. M. de Oliveira Filho, TU Delft, supervisor
Dr. ir. W. G. M. Groenevelt, TU Delft

TU Delft

Preface

This thesis completes my Bachelor Applied Mathematics. A cutting plane approach to upper bounds on the kissing number was the topic of my explicit preference, and it has been extremely interesting to study the kissing number, to code in Julia, to gain practical experience with optimization problems. I expect to come across many of the topics applied during this work in the future again.

Fernando Oliveira has been of great help during the entire project, both with his knowledge on the kissing number and his ideas on solving optimization problems, for which I am thankful. He has provided a lot of the material on which this work is built. I extend my gratitude to Wolter Groenevelt for being a part of the graduation committee. I thank my brother and my parents. Their support has meant a lot.

Noud Riemens
Delft, August 15, 2023

Abstract

Upper bounds for the kissing number can be written as a semidefinite program (SDP) through the Delsarte-Goethals-Seidel method for spherical codes. This thesis solves the resulting SDP with a cutting plane approach, in which a sequence of linear programs (LPs) is solved with the addition of linear constraints every round. We study the computational efficiency of dense and sparser cuts. Sparse cuts are obtained through a relation to the k-Sparse Principal Component Analysis problem. For the modest polynomial degrees considered, the dense and sparse methods show similar performance. Upper bounds are obtained through calculations in standard and where necessary quadruple precision. Lastly, it is shown that under a linear cutting plane approach the SDP is solved quicker if not every subsequent LP is solved till optimality.

Layman Abstract

The kissing number in dimension n is the maximum number of non-overlapping n-dimensional unit spheres that can be aligned such that they all touch a central n-dimensional unit sphere. For most dimensions only upper and lower bounds are known. Upper bounds can be calculated through an optimization problem with an infinite amount of linear constraints. In this thesis its optimal solution is approached through solving a sequence of linear programs, adding new linear constraints each round. These constraints can involve a lot of variables (in which case they are called dense), or a lesser amount (sparse). A strategy is presented to find effective sparse cuts. For the modest problem sizes considered, the dense and sparse strategies lead to similar computational effort required to calculate the kissing number upper bounds. The results show that in general the linear programs do not have to be solved till optimality, which leads to shorter computation times.

Contents

1

Introduction

Visualize - or grab - a tennis ball. How many other tennis balls would you be able to place around this first one, such that every outer tennis ball touches (”kisses”) it? The maximum amount possible is known as the kissing number in dimension three. With the help of some extra hands you could probably let 12 tennis balls kiss a central one, but are you convinced that there is no room for number 13? The mathematical formulation of the kissing number naturally extends to higher dimensions. A unit sphere centered at the origin is defined as the set of points at distance 1 of the origin, i.e. $S^{n-1} = \{x \in \mathbb{R}^n : ||x|| = 1\}$. What is the maximum number of unit spheres that can simultaneously touch a central one, while none of them overlap? Trying to fit 7-dimensional spheres around another 7-dimensional sphere is likely best left to a computer.

Even for a computer this is no easy task. Generally, the exact kissing numbers for higher dimensions are unknown: we only have lower and upper bounds. The upper bounds for kissing numbers can be obtained through solving an optimization problem. To understand this optimization problem, and attempt to solve it, some preliminaries are necessary. The *trace inner product* of two matrices $A, B \in \mathbb{R}^{nxn}$ is given by $\langle A, B\rangle = Tr(A^T B) = \sum_{i,j=1}^{n} A_{ij} B_{ij}$. A symmetric matrix X is *positive semidefinite* (denoted $X \succeq 0$) if $v^T X v \geq 0$ for all $v \in \mathbb{R}^n$, or equivalently if all eigenvalues of X are nonnegative. A semidefinite program is the problem of finding a positive semidefinite matrix such that a given linear combination of the matrix elements is minimized, while given linear constraints in the matrix elements hold (Definition 1.0.1).

Definition 1.0.1. Let $X, C, A_i \in \mathbb{R}^{nxn}$ be symmetric matrices and $b_i \in \mathbb{R}$, $i \in \{1, \ldots, m\}$. A semidefinite program (SDP) is an optimization problem of the following form:

$$\begin{aligned} \text{minimize} \quad & \langle C, X\rangle \\ \text{subject to} \quad & \langle A_i, X\rangle = b_i, \quad i = 1, \ldots, m \\ & X \succeq 0. \end{aligned} \tag{1.1}$$

in which X is the variable matrix.

While semidefinite optimization is a relatively modern field of optimization mathematics, it is well-studied and has practical and theoretical importance. In this thesis a non-standard approach, referred to as the cutting plane method, to solving (1.1) will be studied with the goal of calculating upper bounds on the kissing number. The cutting plane approach consists of solving a sequence of easier optimization problems, in which each iteration cuts away a part of its feasible region till the feasible region and an optimal solution therein of (1.1) is reached. The cutting planes considered in this thesis will be linear, and of specific interest will be the construction of these cuts and the consequences for computational efficiency. Particularly the performance of dense and sparser cuts for the kissing number problem will be compared.

1.1. Content overview

A more complete review on the kissing number is given in Chapter 2. Bounds on the kissing number and obtaining upper bounds via a SDP formulation are treated in Section 2.1. Having shown that the kissing number bounds can be written in the form (1.1), Chapter 3 discusses the cutting plane approach for (1.1). Section 3.1 explores potential advantages of using sparse cuts and lays out a method for obtaining a single sparse cut. With this, a strategy to obtain a round of sparse cuts is presented in Section 3.2. Finally, Chapter 4 applies the cutting plane method with strategies using dense and sparse cuts to the kissing number problem. Analysis on computational efficiency is given in Section 4.1, whereas the obtained kissing number upper bounds can be found in Section 4.2. The results are explicitly stated and discussed through Chapter 5.

2

The Kissing Number

The kissing number is the maximum number of non-overlapping unit spheres that can simultaneously touch a central unit sphere. With n the dimension of the Euclidean space in which this problem is considered, the kissing number is denoted by τ_n. For the first two dimensions the kissing number problem has trivial solutions. In dimension one the sphere configuration in Figure 2.1 leaves no contact points on the central sphere left, thus $\tau_1 = 2$. In dimension two the configuration in Figure 2.2 has all neighbouring spheres touching and hence $\tau_2 = 6$.

Figure 2.1: When a line is placed in a one-dimensional space, there is room for two other lines to touch the original line. Hence the kissing number in the first dimension is two [28].

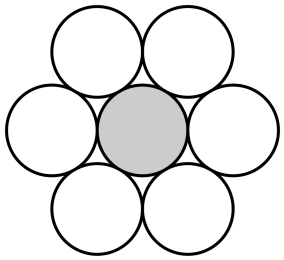


Figure 2.2: In dimension two the spheres are circles. Exactly six non-overlapping circles can kiss a central circle, hence $\tau_2 = 6$ [29].

The formulation of the kissing number problem is not incredibly recent, but progress is. It is simple to show that in the third dimension twelve spheres are possible (for instance by the configuration in Figure 2.3), but the first correct proof showing that a configuration with thirteen spheres is impossible and thus that $\tau_3 = 12$ was published only in 1953 by Schütte and van der Waerden [37]. Sphere configurations in higher dimensions can not be as easily visualized as in the first three dimensions, but the kissing number problem is still well-defined for $n \geq 4$. In 2003 Musin proved $\tau_4 = 24$ [30].

2.1. Bounds on the kissing number

Up to and including dimension four the kissing number is exactly known. Which techniques allowed Musin to prove $\tau_4 = 24$, and what is known about the kissing number in higher dimensions? For $n \geq 5$, with the exception of $n = 8$ and $n = 24$, the exact kissing number is unknown: only lower and upper bounds are available. An excellent and up-to-date overview of the current best bounds is kept by Henry Cohn on [10].

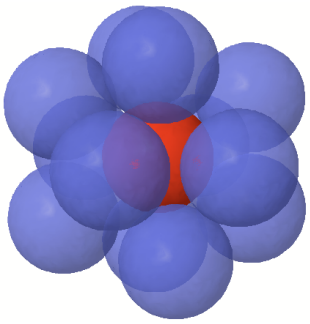


Figure 2.3: Unlike in the first two dimensions, the solution for the kissing number problem in the third dimension is not trivial. Here one of the possible configurations with 12 spheres is shown - there is still space left, but not enough for a thirteenth sphere [8].

Generally, lower bounds for a kissing number come from finding valid configurations in which the centers of the outer spheres lie on some lattice. The configuration in Figure 2.3 proves $\tau_3 \geq 12$, and particularly the outer sphere centers are placed on vertices of a regular icosahedron [8]. Likewise, lattice configurations show for instance $\tau_5 \geq 40$, $\tau_6 \geq 72$ ([18]) or $\tau_{17} \geq 5346$, $\tau_{18} \geq 7398$ ([21]). A recent preprint by Ganzhinov [14] finds lower bound configurations from representation theory ($\tau_{10} \geq 510$, $\tau_{11} \geq 592$). Finding upper bounds requires different techniques: by definition no valid sphere configuration is possible for a strict upper bound. Upper bounds for the kissing number are a topic of ongoing research, both in theoretical as well as computational aspects. The ground for current research was laid in 1973 by Delsarte, Goethals and Seidel with the development of their two-point upper bound method [11], also known as the linear programming bound.

2.1.1. Delsarte-Goethals-Seidel method

Consider the points of contact between any two outer unit spheres and the central sphere in a valid kissing configuration; denote the associated vectors with x, y. The minimum angular distance between these points of contact is then $\frac{\pi}{3}$. Equivalently, $x \cdot y \leq \cos\frac{\pi}{3}$ where $\cdot$ denotes the standard inner product. For instance, the configuration in Figure 2.2 has exactly an angular distance of $\frac{\pi}{3}$ for any two neighbouring outer spheres, and angular distances $> \frac{\pi}{3}$ for non-neighbouring outer spheres. Solving the kissing number problem is equivalent to searching for the maximum amount of points we can place on a unit sphere S^{n-1} such that they have minimum angular distance $\frac{\pi}{3}$, which is an instance of what is known as the spherical code problem. Formally the maximum size of a spherical code is denoted by $A(n, \theta)$:

$$A(n, \theta) = \max\{|C| : C \subset S^{n-1},\ x \cdot y \leq \cos\theta\ \forall x, y \in C\ s.t.\ x \neq y\},$$

and thus $\tau_n = A(n, \frac{\pi}{3})$. An upper bound for $A(n, \theta)$ is given by [11]:

Theorem 2.1.1 (Delsarte-Goethals-Seidel)**.** Let $F(t) = \sum_{k=0}^{d} f_k P_k^n(t)$. If:

1. $f_k \geq 0$ for all $k \geq 1$ and $f_0 > 0$ and
2. $F(t) \leq 0$ for all $t \in [-1, \cos\theta]$,

then

$$A(n, \theta) \leq \frac{F(1)}{f_0},$$

in which the Gegenbauer polynomials $P_k^n(t)$ are given by the recursive relationship

$$P_k^n(t) = \frac{2k + n - 4}{k + n - 3} t P_{k-1}^n(t) - \frac{k - 1}{k + n - 3} P_{k-2}^n(t)$$

for $k \geq 2$, and $P_0^n(t) = 1$, $P_1^n(t) = t$. A full contemporary proof of Theorem 2.1.1 can be found in [4]. Less technical but recommended is the review in [32].

Theorem 2.1.1 provides upper bounds for instances of the spherical code problem: particularly the lowest upper bounds are of interest. First note that the polynomials $P_k^n(t)$ are normalized in such a way that $P_k^n(1) = 1$, hence $F(1) = \sum_{k=0}^{d} f_k$. Using $\theta = \frac{\pi}{3}$, finding the lowest Delsarte-Goethals-Seidel upper bound for the kissing number problem is the following optimization problem[1]:

$$\begin{aligned} \text{minimize} \quad & 1 + \sum_{k=1}^{d} f_k \\ \text{subject to} \quad & f_k \geq 0 \quad \forall k = 1, \ldots d, \\ & 1 + \sum_{k=1}^{d} f_k P_k^n(t) \leq 0 \quad \forall t \in [-1, 0.5]. \end{aligned} \tag{2.1}$$

The linear programming problem (2.1) contains an infinite amount of linear constraints, namely one for every $t \in [-1, 0.5]$. Hence, the problem still needs some massaging to be of practical use. In particular, the polynomial $1 + \sum_{k=1}^{d} f_k P_k^n(t)$ of degree d should be nonpositive on the interval $[-1, 0.5]$. A theorem of Lukács ([39]) specifies when this is the case.

Theorem 2.1.2 (Lukács)**.** Let $f(t)$ be a polynomial of even degree d. If and only if f is nonpositive on $[a, b]$, f can be written as:

$$f(t) = -h_1^2(t) - (t - a)(b - t)h_2^2(t),$$

where h_1 and h_2 are polynomials of at most degree $\frac{d}{2}$ and $\frac{d}{2} - 1$ respectively.

This thesis only concerns itself with the case in which d is even; similar results hold for d uneven. By Theorem 2.1.2 the infinite amount of linear constraints on the polynomial $1 + \sum_{k=1}^{d} f_k P_k^n(t)$ becomes:

$$1 + \sum_{k=1}^{d} f_k P_k^n(t) = -h_1^2(t) - (t + 1)(0.5 - t)h_2^2(t). \tag{2.2}$$

Sum-of-squares polynomials are polynomials of the form $g(t) = p_1^2 + \ldots + p_m^2$. Writing $q_1 = h_1^2$ and $q_2 = h_2^2$, it is clear that q_1 and q_2 are sum-of-squares polynomials, specifically the sum of one square. Thus the constraint (2.2) can be written as

$$1 + \sum_{k=1}^{d} f_k P_k^n(t) = -q_1(t) - (t + 1)(0.5 - t)q_2(t), \tag{2.3}$$

in which q_1 and q_2 are sums-of-squares. Theorem 2.1.3 connects sum-of-squares polynomials to positive semidefinite matrices.

Theorem 2.1.3. Let $q \in \mathbb{R}[t]$ be a polynomial of even degree and let B be a basis of $\mathbb{R}[t]_{\leq \frac{d}{2}}$ with v_B the associated basis vectors. If and only if q is sum-of-squares there is a positive semidefinite matrix $Q : B \times B \to \mathbb{R}$ such that $q = v_B^T Q v_B$.

Proof. Included is a proof for "⇒". q is sum-of-squares, i.e. $q = p_1^2 + \ldots + p_m^2$ for some $p_1, \ldots, p_m$. Since q is of degree d, each p_i has degree at most $\frac{d}{2}$ and can be written as $p_i = u_i^T v_B$ for some $u_i : B \to \mathbb{R}$. Then $q = v_B^T Q v_B$ with $Q = u_1 u_1^T + \ldots + u_m u_m^T$ positive semidefinite. □

[1] The code in Appendix A, B allows decision variables $f_0, \ldots, f_d$ instead of $f_1, \ldots, f_d$. This does not influence the optimization problem, but is slightly less efficient.

Denote the spaces of polynomials up to degree $\frac{d}{2}$ and $\frac{d}{2}-1$ by $\mathbb{R}[t]_{\leq \frac{d}{2}}$ and $\mathbb{R}[t]_{\leq \frac{d}{2}-1}$ respectively and let B_1 and B_2 be bases of $\mathbb{R}[t]_{\leq \frac{d}{2}}$ and $\mathbb{R}[t]_{\leq \frac{d}{2}-1}$. By Theorem 2.1.3 the polynomials q_1 and q_2 can be represented with the use of positive semidefinite matrices. In this thesis only the standard basis is used - de Laat et al. [20] show selecting a different basis can be beneficial. Thus, write $v_{B_1}^T = (1, t, ..., t^{\frac{d}{2}})$ and $v_{B_2}^T = (1, t, ..., t^{\frac{d}{2}-1})$. Then there exist matrices $X_1 \succeq 0$ and $X_2 \succeq 0$ such that $q_1 = v_{B_1}^T X_1 v_{B_1}$ and $q_2 = v_{B_2}^T X_2 v_{B_2}$. Alternatively q_1 and q_2 can be represented as a trace inner product: $q_1 = \langle v_{B_1} v_{B_1}^T, X_1 \rangle$ and $q_2 = \langle v_{B_2} v_{B_2}^T, X_2 \rangle$.

With the constraint (2.3) rewritten using trace inner products, the lowest Delsarte-Goethals-Seidel upper bound for the kissing number is found by solving:

$$\begin{aligned} \text{minimize} \quad & 1 + \sum_{k=1}^{d} f_k \\ \text{subject to} \quad & f_k \geq 0 \quad \forall k = 1, ...d, \\ & 1 + \sum_{k=1}^{d} f_k P_k^n(t) + \langle v_{B_1} v_{B_1}^T, X_1 \rangle \\ & + \langle (t+1)(0.5-t) v_{B_2} v_{B_2}^T, X_2 \rangle = 0, \\ & X_1, X_2 \succeq 0. \end{aligned} \tag{2.4}$$

Finally, let $X = \begin{pmatrix} X_1 & & & & \\ & X_2 & & & \\ & & f_1 & & \\ & & & \ddots & \\ & & & & f_d \end{pmatrix}$ and $v_B v_B^T = \begin{pmatrix} v_1 & & & & \\ & v_2 & & & \\ & & 0 & & \\ & & & \ddots & \\ & & & & 0 \end{pmatrix}$ in which v_1 and v_2 are the matrices $v_{B_1} v_{B_1}^T$ and $(t+1)(0.5-t) v_{B_2} v_{B_2}^T$ respectively, to simplify (2.4). In particular the nonnegativity constraints for the decision variables f_i are incorporated into the decision matrix, of which their is now only one (2.5).

$$\begin{aligned} \text{minimize} \quad & 1 + \sum_{k=1}^{d} f_k \\ \text{subject to} \quad & 1 + \sum_{k=1}^{d} f_k P_k^n(t) + \langle v_B v_B^T, X \rangle = 0, \\ & X \succeq 0. \end{aligned} \tag{2.5}$$

The semidefinite optimization problem (SDP) (2.5) outputs an upper bound for τ_n, depending on the maximum degree d of the Gegenbauer polynomials $P_k^n(t)$ used. The size of the optimization problem is not dependent on the dimension n. With an increase of d a larger set of polynomials is considered, thus a higher d leads to a lower upper bound found. This highlights the need for a computationally efficient way to solve (2.5).

2.1.2. Recent improvements

While this thesis only considers the Delsarte-Goethals-Seidel upper bound in the form of (2.5), it is worth mentioning its limitations and recent improvements. The formulation (2.1) was used by Odlyzko and Sloane [31] to calculate a considerable amount of then best upper bounds for kissing numbers (although via a constraint sampling method and subsequent interval arithmetic to arrive at upper bounds instead of the semidefinite approach (2.5)). Still their best result for the fourth dimension was $\tau_4 \leq 25.5585$, i.e. $\tau_4 \leq 25$. In fact, in 2000 Arestov and Babenko proved that the Delsarte-Goethals-Seidel upper bound for τ_4 will never be lower than 25 ([3]). Musin [30] finally showed that there exists conditions under

which the constraint $1 + \sum_{k=1}^{d} f_k P_k^n(t) \leq 0$ does not have to hold towards the left side of the interval $[-1, 0.5]$, which through a more involved optimization problem lead to $\tau_4 = 24$. An interesting review can once again be found in [32].

Further progress has been made by Bachoc and Vallentin [4] with the consideration of a three points distance distribution on the unit sphere. The resulting SDP gives tighter bounds than the two points distance distribution SDP (2.5). However, both optimization problems suffer from numerical instability. While this instability is worse for higher dimensions, Mittelmann and Vallentin [26] showed that already for $n = 5$ quadruple precision leads to a better bound ($\tau_5 \leq 44$, previously ≤ 45 ([4])). In 2018 Machado and Oliveira [9] calculated new best upper bounds for $\tau_9, \ldots, \tau_{23}$ by exploiting polynomial symmetry in the formulation of the three point bound. While the numerical solver outcomes have not been rigorously verified as correct polynomial solutions, computational speedup has been achieved by Leijenhorst and de Laat with a semidefinite solver ([23]) for problems with low-rank constraint matrices, leading to improved upper bounds for $\tau_{11}, \ldots, \tau_{23}$.

3

A Cutting Plane Approach

Semidefinite optimization is a relatively modern field within optimization mathematics, applicable to a wide range (Figure 3.1) of real-life as well as more theoretical problems. In control and systems theory, semidefinite programming is used in the search for Lyapunov functions for a range of different systems, and linear matrix inequalities are found in a variety of problems [7]. Other semidefinite optimization applications - a non-exhaustive list can be found in [41] - range from eigenvalue optimization, to robust optimization, as well as relaxations for combinatorial problems. A notable result is a 0.87856-approximation algorithm for the maximum cut problem by Goemans and Williamson [15]. Sphere packings and spherical codes fall under this last category as well, and indeed upper bounds for the kissing number have been written as a semidefinite programming problem (2.5). A formal expression of a semidefinite program is given by Definition 1.0.1.

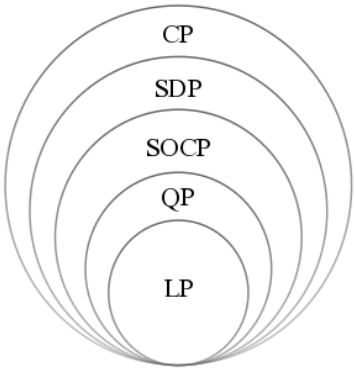


Figure 3.1: Semidefinite programming encapsulates a wide range of problems in the class of conic optimization problems. In particular any linear program, quadratic program, or second-order cone program can be written as a SDP [2].

There are multiple ways to view the matrix X in Definition 1.0.1. In the context of this thesis, particularly helpful is to realize $X \in \mathbb{R}^{nxn}$ is essentially an array x of decision variables, i.e. $x = (x_1, \ldots, x_{0.5(n^2+n)})$. A SDP minimizes a linear combination $c^T x$ of these decision variables, while some linear constraints $a_i^T x = b_i$ must hold. However, the constraint $v^T X v \geq 0 \ \forall v \in \mathbb{R}^n$ adds to the complexity of the optimization problem. For instance, a classical example by Kyachiyan shows a SDP for which even the bit-size of an optimal solution is exponential in the amount of variables [34]. In most practical cases more favorable results hold and optimal solutions up to an additive ϵ-precision can be found by polynomial time algorithms. Algorithms to solve SDPs have mainly been generalizations of interior-point methods for linear programs (LPs). An analysis of several interior-point methods for SDPs, their complexity and practical performance can be found in [7]. The general theory of interior-point methods for convex optimization is treated in [35].

An alternative approach is to relax the constraint $v^T X v \geq 0 \ \forall v \in \mathbb{R}^n$ to include only a finite set of vectors $\{v_j\}_{j=1}^k, k \in \mathbb{N}_{\geq 1}$ and to solve the resulting LP, or to solve a sequence of LPs with the addition of constraints $v^T X v \geq 0$ in every iteration. These constraints $v^T X v \geq 0$ are known as (linear) cutting planes. The sampling method used by Odlyzko and Sloane [31] to calculate upper bounds for the kissing number is in fact a rudimentary implementation of the single LP relaxation. A more involved example is the work of Krishnan and Mitchell [19]. Sequential linear cutting planes have previously mostly been considered in the context of quadratically constrainted quadratic programs [33, 38].

Algorithm 1: Cutting Plane Method(SDP)

Input : An initial LP relaxation of a SDP
Parameters: `TerminatingConditions`: check if the LP solution is accepted as SDP solution
Output : Solution matrix X

`Initialize`: $\text{LP}_1 \leftarrow$LP, $t \leftarrow 1$
repeat
  Solve LP_t to obtain a LP solution $\hat{X}_t$
  Obtain a round of cuts $\{v_i\}_{i=1}^p$ using `GenerateCuts` (LP)
  $\text{LP}_{t+1} \leftarrow \text{LP}_t$ with the addition of the cuts $\{v_i\}_{i=1}^p$
  $t \leftarrow t + 1$
until `TerminatingConditions`
$X \leftarrow \hat{X}_t$
return X

Algorithm 1 more formally describes this method of sequential LP solving. How to obtain the set of cuts each round, i.e. which algorithm to use for *GenerateCuts*(LP)? A first option is to use the eigenvectors $\{v_j\}_{j=1}^m$ corresponding to negative eigenvalues of a solution $\hat{X}_t$. These eigenvectors are valid cuts: $v_j \hat{X}_t v_j < 0$, and certainly the constraints $v_j X v_j \geq 0$ are valid for the SDP (1.1).

3.1. The use of k-sparse eigencuts

A known downside to this approach lies in the constraints added from the eigenvectors v_i. Typically these cuts will be dense, and when a lot of them are used the LPs become slow to solve, as well as potentially suffer from numerical issues. Furthermore, the amount of cuts added per iteration is quite limited (one per negative eigenvalue), so a lot of LPs will need to be solved before the positive semidefinite cone is succesfully approached. Results from Baltean-Lugojan et al. and Qualizza et al. [5, 33] show that the use of sparse cuts can improve computational effort required. In particular Qualizza et al. [33] "sparsify" the dense cuts found from eigenvectors corresponding to negative eigenvalues, among the application of other methods. Recently Dey et al. [13] have performed an extensive computational study on a cutting plane method in which sparsity is directly enforced, and it is their method applied in this thesis.

Consider the support of a vector v, defined as the nonzero entries of this vector v and denoted by $supp(v)$. Enforcing a sparsity k on v means $|supp(v)| \leq k$. Thus Definition 3.1.1 characterizes k-sparse eigencuts. A principle submatrix of $\hat{X}$ for a certain index set I is the matrix formed by the rows and columns with indices in I of $\hat{X}$, and denoted by $\hat{X}_I$.

Definition 3.1.1. A k-sparse eigencut of $\hat{X} \in \mathbb{R}^n$ is a vector $v \in \mathbb{R}^n$ such that:

- $v^T \hat{X} v < 0$,
- $\|v\|_0 := |supp(v)| \leq k$, and
- The k-length vector consisting of the nonzero entries of v is a unit eigenvector of the principal submatrix of $\hat{X}$ obtained through the indices in $supp(v)$.

Ideally, the k-sparse eigencuts which close the most of the gap in objective value between the current LP relaxation and the SDP would be added each round. Exactly selecting these most efficient vectors would require solving the subsequent LPs, which defeats the purpose of adding lightweight cuts. Con-

sider instead the violation of a cut v, the value $v^T \hat{X} v$ (< 0), as a measure for its efficiency. In particular finding the most promising (i.e. violated) k-sparse eigencut then equals the optimization problem (3.1), which is similar to the k-Sparse Principal Component Analysis (k-SCPA) problem (Definition 3.1.2).

$$\begin{aligned} \text{minimize} \quad & v^T \hat{X} v \\ \text{subject to} \quad & ||v||_2 = 1 \\ & ||v||_0 \leq k \end{aligned} \tag{3.1}$$

Definition 3.1.2. For a positive semidefinite matrix $A \in \mathbb{R}^{nxn}$ and a sparsity level $k \in \mathbb{N}$, the k-Sparse Principal Component Analysis (k-SPCA) problem consists of finding the following $v \in \mathbb{R}^n$:

$$\begin{aligned} \text{maximize} \quad & v^T A v \\ \text{subject to} \quad & ||v||_2 = 1 \\ & ||v||_0 \leq k. \end{aligned} \tag{3.2}$$

When solving the most violated k-sparse eigencut problem, the matrix $\hat{X}$ will not be positive semidefinite (by definition). The k-SPCA problem however is defined for matrices $X \succeq 0$. Dey et al. [13] show that in any case an instance of (3.1) can be translated into an instance of (3.2); see Lemma 3.1.1. Their proof is included and contains an explicit construction of this translation.

Lemma 3.1.1. For any matrix $\hat{X} \not\succeq 0, \hat{X} \in \mathbb{R}^{nxn}$ and $k \in \mathbb{N}$, there exists a matrix $A \succeq 0, A \in \mathbb{R}^{nxn}$ such that finding the most violated k-sparse eigencut of $\hat{X}$ (3.1) is equivalent to solving the k-SPCA problem (3.2), i.e. both optimization problems have the same solution vector $v \in \mathbb{R}^n$.

Proof. Denote the largest eigenvalue of $\hat{X}$ by λ^{max}, and let $A = \lambda^{max} I - \hat{X}$ in which I is the nxn identity matrix. Then all eigenvalues of A are positive: A is positive semidefinite. Furthermore

$$v^T \hat{X} v = v^T (\hat{X} - \lambda^{max} I) v + \lambda^{max} v^T v,$$

and using $v^T v = 1$ and the fact that the solution of a minimization problem is equal to that of the problem maximizing the negative of the same objective function, (3.1) can be written as

$$\begin{aligned} \lambda^{max} - \text{maximize} \quad & v^T (\lambda^{max} I - \hat{X}) v \\ \text{subject to} \quad & ||v||_2 = 1 \\ & ||v||_0 \leq k \end{aligned}$$

which is an instance of k-SPCA (3.2) using the positive semidefinite matrix A. □

Thus, the most violated k-sparse eigencut of an intermediate LP solution $\hat{X}$ can be found by solving the k-SPCA problem for $A = \lambda^{max} I - \hat{X}$. The k-SCPA problem is NP-hard [27], but practically efficient methods with good results on convergence and empirical worst-case performance exist [12, 42]. In particular, consider the Truncated Power Method (Algorithm 2) proposed by Yuan and Zhang [42], which is a variation on the power method; every iteration includes a truncation (Definition 3.1.3) step via which k-sparsity is enforced.

Definition 3.1.3. The truncation operation $Truncate(x, F)$ for a given vector x and index set F sets elements of x not in F to zero:

$$[Truncate(x, F)]_i = \begin{cases} [x]_i & i \in F \\ 0 & \text{else.} \end{cases}$$

Algorithm 2: Truncated Power Method(A, x_0, k)

Input : A symmetric matrix $A \in \mathbb{R}^{nxn}$ and an initial vector $x_0 \in \mathbb{R}^n$
Parameters: Cardinality $k \in \{1, ..., n\}$
Output : k-sparse vector x

Initialize: $t \leftarrow 1$
repeat
  $x'_t = Ax_{t-1}/||Ax_{t-1}||$
  Let $F_t =$ supp (x'_t, k) be the indices of x'_t with the largest k absolute values
  $\hat{x}_t =$ Truncate(x'_t, F_t)
  Normalize $x_t = \hat{x}_t/||\hat{x}_t||$
  $t \leftarrow t + 1$
until Convergence
$x \leftarrow x_t$
return x

Algorithm 2 allows for the calculation of a single k-sparse eigencut approximately solving (3.1). This is not yet a complete strategy for generating cuts in between subsequent LPs: a multitude of k-sparse eigencuts should be added, capable of reducing the relaxation gap in a manner similar to dense eigencuts.

3.2. Generating a round of k-sparse eigencuts

The computational study by Dey et al. [13] shows that a round of k-sparse eigencuts should include cuts from multiple supports to be effective. Furthermore, the violation $v^T \hat{X} v$ is a suitable measure for the strength of a cut, which justifies the approach (3.1) and thus allows a round of cuts to be generated without having to solve additional LPs. Dey et al. propose the use of Algorithm 3 (*SparseRound*$(\hat{X}, k)$), which returns a round of k-sparse eigencuts given a matrix $\hat{X} \not\succeq 0$ and a sparsity k. Note the similarity between Algorithm 3 and the standard matrix deflation method - particularly, for $k = n$ all eigenvectors corresponding to negative eigenvalues are returned [13].

Algorithm 3: SparseRound$(\hat{X}, k)$: one round of k-sparse eigencuts

Input : A matrix $\hat{X} \in \mathbb{R}^{nxn}$ with $\hat{X} \not\succeq 0$, and a sparsity level k
Parameters: MaxNumSupports: maximum number of considered supports
TruncatedPowerMethod: Algorithm 2
Output : A sequence of k-sparse eigencuts $\{\hat{w}_j\}_{j=1}^p$

Initialize: $p \leftarrow 0$, $i \leftarrow 1$, $X^1 \leftarrow \hat{X}$, and $w \leftarrow$ TruncatedPowerMethod(X^1)
while $w^T X^i w < 0$ *and* $i <$ MaxNumSupports **do**
  $I \leftarrow$ supp (w)
  Let λ_i^{min} and q_i denote the most negative eigenvalue and its corresponding unit eigenvector of principal submatrix X_I^i
  Let $\hat{w}_i$ denote q_i lifted to $\mathbb{R}^{nxn}$ by setting all components not in I to 0
  $X^{i+1} \leftarrow X^i - \lambda_i^{min} \hat{w}_i \hat{w}_i^T$
  $i \leftarrow i + 1$ and $p \leftarrow p + 1$
  $w \leftarrow$ TruncatedPowerMethod(X^i)
end
return $\{\hat{w}_j\}_{j=1}^p$

Matrix deflation is prone to numerical instability, whereas Algorithm 2 returns a cut w upon reaching some predefined measure of convergence. The solution of Algorithm 3 is to take the support I of w and explicitly calculate the eigenvector corresponding to the most negative eigenvalue of X_I. This does not negatively affect the computational effort required to solve (3.2): the NP-hardness lies in finding an optimal support, not in finding w for some given support. Specifically this allows an iteration limit to be set for Algorithm 2. Lastly, note that Algorithm 2 is called after the translation from (3.1) to (3.2) as given by Lemma 3.1.1, i.e. *TruncatedPowerMethod*$(X^i) =$ *TruncatedPowerMethod*$(\lambda^{max} I - X^i, v_0, k)$ in which

λ^{max} denotes the largest eigenvalue of X^i. Dey et al. [13] prove that every cut provided by Algorithm 3 is a valid cut for the input matrix $\hat{X}$ (Theorem 3.2.1). Thus a strategy to generate a complete round of k-sparse cuts for a matrix $\hat{X} \not\succeq 0$ has been given as an alternative to the use of denser eigenvectors.

Theorem 3.2.1. All vectors $\hat{w}_i$ from $\{\hat{w}_j\}_{j=1}^{p}$ generated by Algorithm 3 are valid cuts for their respective matrices X^i, and for the LP solution $\hat{X}$ given as input, i.e.:

1. $\hat{w}_i^T X^i \hat{w}_i < 0$ for every $i \in \{1, \ldots, p\}$, and
2. $\hat{w}_i^T \hat{X} \hat{w}_i < 0$ for every $i \in \{1, \ldots, p\}$.

Proof. The first part follows from the definition of $\hat{w}_i$. Secondly, write

$$\hat{X} = X^i + \sum_{j=1}^{i-1} \lambda_j^{min} \hat{w}_j \hat{w}_j^T.$$

Then using the first part and $\lambda_j^{min} < 0$,

$$\hat{w}_i^T \hat{X} \hat{w}_i = \hat{w}_i^T X^i \hat{w}_i + \hat{w}_i^T \left(\sum_{j=1}^{i-1} \lambda_j^{min} \hat{w}_j \hat{w}_j^T\right) \hat{w}_i$$

$$= \hat{w}_i^T X^i \hat{w}_i + \sum_{j=1}^{i-1} \lambda_j^{min} (\hat{w}_i^T \hat{w}_j)^2 < 0.$$

□

4

Application to the Kissing Number Problem

Upper bounds for the kissing number have been written as a SDP (2.5), to which the cutting plane approach given by Algorithm 1 can now be applied. In particular the performance of dense eigenvector cuts will be compared to the performance of the k-sparse eigencuts generated by Algorithm 3. The SDPs will generally be of smaller sizes than those considered by Dey et al. in [13], and furthermore the matrix X in (2.5) is block diagonal. For these reasons finding a sufficient amount of k-sparse eigencuts is challenging, which potentially leads to solving a sequence of quite similar LPs, and it might often happen that there are in fact no k-sparse eigencuts. However, adding a multitude of light-weight cuts is still attractive whenever possible. This leads to the following two strategies for *GenerateCuts* (Algorithm 1):

- **Dense:** Add all eigenvectors corresponding to negative eigenvalues of $\hat{X}$ as cuts for the next LP.
- **Sparse:** Generate k-sparse eigencuts for $\hat{X}$ via Algorithm 3. If this generates at least as much cuts as there are negative eigenvalues, and the amount of cuts is greater than two, the k-sparse eigencuts are added as cuts for the next LP. Else, use the eigenvectors corresponding to negative eigenvalues.

The requirement for at least three k-sparse eigencuts has been found helpful in the last stages of Algorithm 1, i.e. results in quicker convergence. The kissing number SDP (2.5) comes with two parameters, dimension n and polynomial degree d. The sparsity level k will be fixed depending on degree d at $k = \frac{d}{2} - 1$ such that the cuts for both relevant matrix blocks are sparse, for the second block minimally. Thus, this is not a computational study on an optimal sparsity level, and for matrices of larger size a lower sparsity level would be more appropriate. The instances are referred to as Dense(n, d) or Sparse(n, d). The SDPs considered can generally be solved within seconds by standard interior point methods, for instance via SDPA [1]. All initial LPs include nonnegativity constraints for diagonal entries. Lastly, in all cases the symmetry of matrix X is exploited in the LP formulation.

4.1. Computational results

The cutting plane approach is implemented in Julia [6], a high-level, high-performance programming language. The standard precision implementation (Appendix A) relies on JuMP [24], whereas the high-precision version (Appendix B) is implemented directly through the underlying MOI [22]; JuMP at this time does not support arbitrary precision [17]. All calculations are done single-threaded using an Intel i7-8750H CPU @2.20GHz.

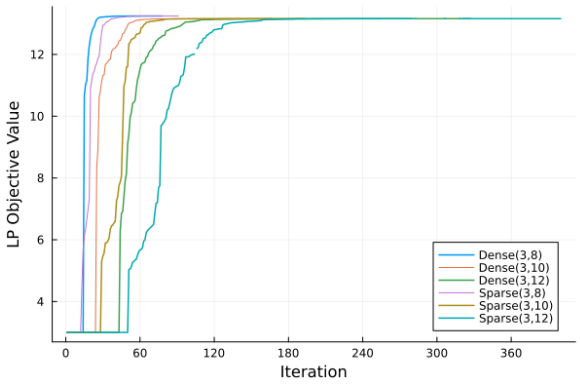
Solver. Interior point LP methods are more suitable for Algorithm 1, which was confirmed during preliminary analysis. Specifically the open-source solver Tulip [40] is used. Tulip has arbitrary precision capabilities. Unless mentioned otherwise, solver parameters are set to their standard values. Note that all references to ”standard precision” in fact mean double precision - this is Tulip’s standard precision.

Cut management. As more LPs are solved, constraints from old iterations might no longer be tight, slowing down each LP while not contributing to the approach of the PSD cone. For this reason cuts are deleted if they are inactive past a tolerance ($v^T normalized(\hat{X})v > 10^{-3}$) for two iterations in a row.

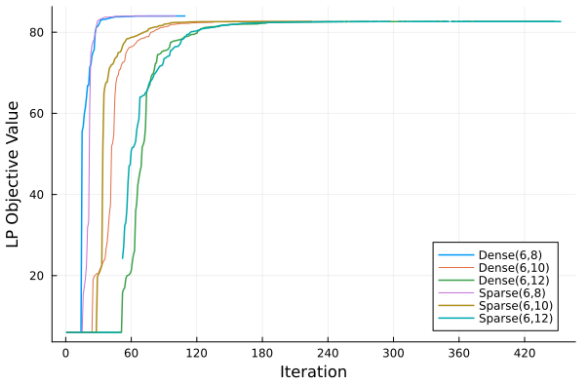
Parameters and tolerances. An eigenvalue λ is considered negative if $\lambda < 10^{-6}$; thus *Terminating-Conditions* in Algorithm 1 is true if all eigenvalues of $\hat{X}_t$ are larger than -10^{-6}. Algorithm 2 is initiated with the eigenvector corresponding to the smallest eigenvalue of X^i, and *Convergence* in Algorithm 2 is defined as $||x_{t+1} - x_t|| < 10^{-12}$, or an iteration limit of 10^4 is reached. Algorithm 3 adds cut while $w^T X^i w < -10^{-7}$. Elements of q_i (Algorithm 3) are considered to be nonzero if their absolute values are larger than 10^{-9}.

4.1.1. Standard precision computations

Before analyzing the performance of Sparse compared to Dense: does Algorithm 1 sufficiently approach a PSD solution at all? Figure 4.1 shows that this is the case for at least relatively small values of n and d. In general, not every LP will be solved till optimality, but rather the solver will terminate on hitting its IPM iteration limit, more often so for larger optimization problems. This is not a problem as long as a PSD solution is eventually still reached, but does point towards numerical instability for iteration limits greater than a hundred [40].

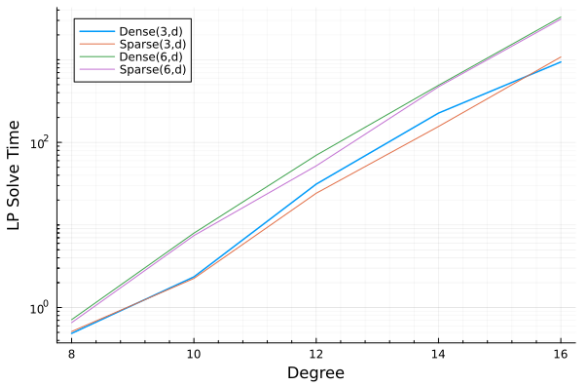


(a) The LP values per iteration for Sparse and Dense in dimension 3, polynomial degrees 8, 10 and 12.

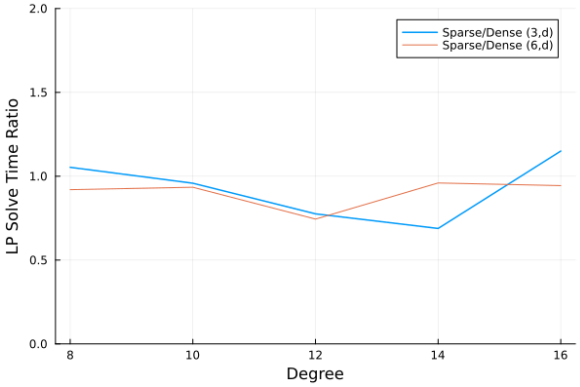


(b) The LP values per iteration for Sparse and Dense in dimension 6, polynomial degrees 8, 10 and 12.

Figure 4.1: The objective values of the iteratively solved LPs for two dimensions of the kissing number and different polynomial degrees for the Dense and Sparse methods. Algorithm 1 converges to a suitable solution for the SDPs. A higher polynomial degree implies a lower upper bound, but more LPs need to be solved. The IPM limit is set to 300; still, not every LP is solved till optimality.



(a) The time spent solving LPs (in seconds) using Sparse and Dense for dimension 3 and 6, with polynomial degrees 8 - 16.



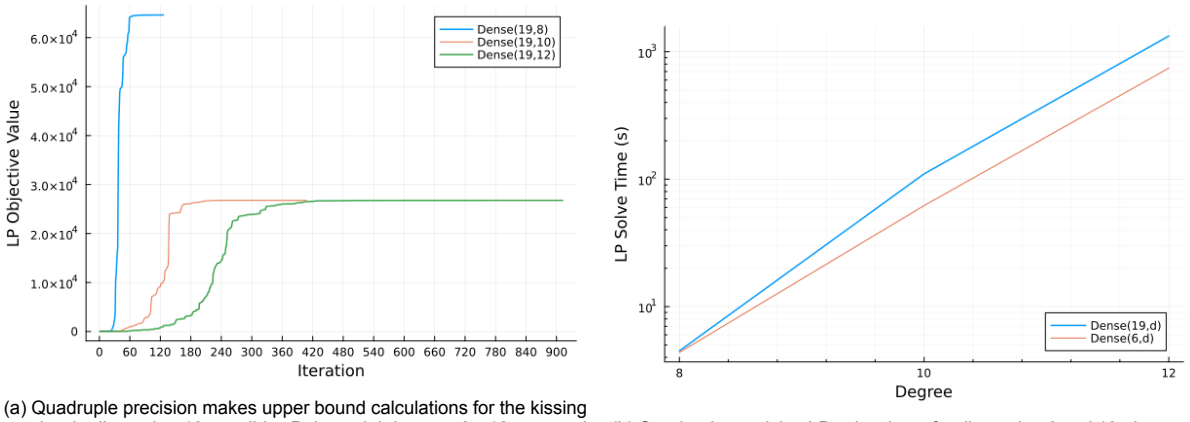
(b) The ratio of time spent solving LPs between Sparse and Dense for dimension 3 and 6, with polynomial degrees 8 - 16.

Figure 4.2: The computation times required to solve different cases of Sparse and Dense are compared. Figure 4.2a shows the respective times for polynomial degrees 8 - 16 in dimension 3 and 6; notice the log scale. Figure 4.2b shows the ratio between the time spent solving LPs for Sparse and Dense. Mostly Sparse is slightly quicker.

Due to the iterative nature of Algorithm 2, the Sparse strategy requires more computational effort to generate cuts than Dense. However for larger matrices generating cuts is not the computational bottleneck - solving the LPs is. To compare Dense with Sparse, only the time spent solving LPs is taken into account. Figure 4.2 shows these times for the cases in Figure 4.1 and for some higher polynomial degrees. From Figure 4.2a it is clear that solve time increases greatly with a higher polynomial degree. Sparse and Dense do not differ significantly, as can be seen in Figure 4.2b. The most positive result of these results is $(3,14)$, for which the Sparse/Dense time ratio is 0.688.

4.1.2. High precision computations

The cutting plane approach can not be blamed for all numerical difficulties encountered. The optimization problem tends to be less stable for larger dimensions. For instance, using standard precision computations SDPA can only solve $(19,d)$ for $d \geq 28$. To still be able to compute upper bounds, Algorithm 1 including cut generation is implemented in quadruple precision using DoubleFloats [36]. Of course, performing calculations in higher precision comes at a computational cost.



(a) Quadruple precision makes upper bound calculations for the kissing number in dimension 19 possible. Polynomial degrees 8 - 12 are used and the iterative LP values plotted.

(b) Quadruple precision LP solve times for dimension 6 and 19, degrees 8 - 12. Compare dimension 6 with Figure 4.2a.

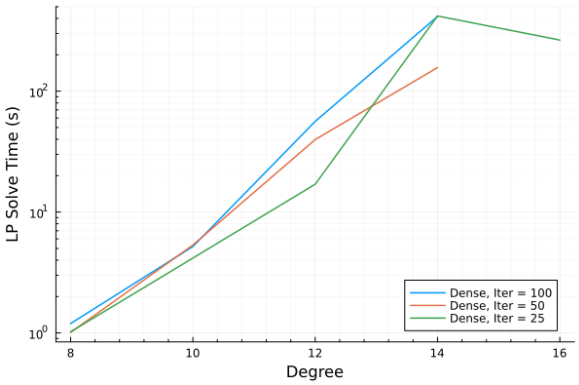
Figure 4.3: Using quadruple precision, the cutting plane method converges for higher dimensions as well, including $n = 19$. The operations are computationally more expensive than using standard precision.

Figure 4.3 shows that less stable problems can be solved as well. The Sparse algorithm did not lead to significant improvements for computational effort required; results are comparable to those in standard precision. This does show there is no numerical problem with the cuts being generated from Algorithm 3 using standard precision, i.e. the matrix deflation is not too unstable.

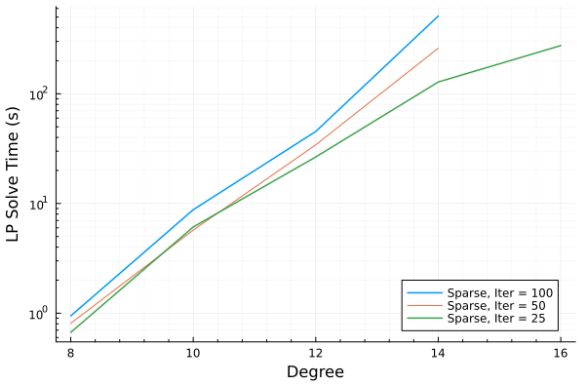
4.1.3. IPM iteration limit

There are multiple possible approaches to solving LPs. Commonly used is the Simplex method, which guarantees a vertex optimal solution and has superior warm-start capabilities. Still, interior point methods are more suited for cutting plane approaches, the problems studied here included. The advantage of IPMs lies exactly in not returning a vertex solution, generally resulting in stronger cuts being generated [25]. Do the LPs even have to be solved till optimality at all? There are two possible advantages. First, if an LP is terminated early less time has been spent solving it. Secondly, a more interior point could result in stronger cuts. To study this, a couple of problems have been solved with varying IPM iteration limits. When a PSD solution is found (which might not be optimal due to early termination) the solver continues with iteration limit 200 till an optimal PSD solution is found. The results are shown in Figure 4.4.

A lower IPM iteration limit leads to an improvement of time spent solving LPs. In particular, the result seems more pronounced for larger SDPs, although one of the Dense cases ran into numerical issues. By halving the IPM iteration limit, reasonably no more than a halving of the time spent solving LPs might be expected, unless the cuts generated are also of higher quality. For some runs the latter seems to be the case - for instance, compare Dense$(5,14)$ with iteration limits 100 and 50. In further support of this is the fact that lowering the IPM iteration limit frequently leads to less iterations of LPs being necessary.



(a) The time spent solving LPs (in seconds) using Dense$(5, 8-14/16)$ for varying maximum IPM iterations.



(b) The time spent solving LPs (in seconds) using Sparse$(5, 8-14/16)$ for varying maximum IPM iterations.

Figure 4.4: Both for Sparse and Dense the required solving time is positively affected by a lower IPM iteration limit. For limits 50 and 100 the algorithms are run for degrees 8 - 14, for limit 25 degree 16 is added. For higher degrees, generally the improvement seems to increase. Dense$(5, 14)$ with IPM limit 25 ran into numerical difficulties. All calculations are performed using standard precision.

4.2. Upper bounds on the kissing number

Finally, an overview of upper bounds obtained using the cutting plane approach can now be presented. Upper bounds are calculated for τ_3 up to τ_{24}. All calculations are performed such that they terminate in a reasonable time frame: the upper bounds took between 300 and 7200 seconds of LP solving. All computations are performed with an IPM iteration limit of 50 or 25. The results are shown in Table 4.1.

n	Upper Bound	Strategy	n	Upper Bound	Strategy
3	13.1583117	Sparse(3,18)	14	3492.20320	*Dense(14,14)*
4	25.5564353	*Dense(4,16)*	15	5431.02394	*Dense(15,12)*
5	46.3375628	Sparse(5,18)	16	8326.94691	*Dense(16,12)*
6	82.6311980	Sparse(6,16)	17	12290.0331	*Dense(17,12)*
7	140.162430	Sparse(7,16)	18	18199.2794	*Dense(18,12)*
8	239.999906	*Dense(8,12)*	19	26770.9892	*Dense(19,12)*
9	380.099018	Sparse(9,14)	20	39654.9827	*Dense(20,12)*
10	594.481567	*Dense(10,16)*	21	59693.0942	*Dense(21,12)*
11	914.388883	*Dense(11,14)*	22	88391.8427	*Dense(22,12)*
12	1416.08958	*Dense(12,14)*	23	130338.994	*Dense(23,12)*
13	2232.63206	*Dense(13,14)*	24	196559.963	*Dense(24,12)*

Table 4.1: Computed upper bounds on the kissing number τ_n for dimensions $n = 3$ up to $n = 24$ using the cutting plane approach. The strategies denoted in italics are performed in quadruple precision.

In dimension 4 the standard precision implementation encountered numerical difficulties; hence high precision is used. For the matrices considered in Table 4.1, the Dense strategy is preferable for high precision calculations due to the computational effort necessary for Algorithm 2. For larger sized matrices this would not be a problem: both strategies would spend most time solving LPs. Note the outcomes for $n = 8$ and $n = 24$. For these dimensions the bound provided by (2.5) is actually tight (240 and 196560 respectively). The cutting plane approach converges to these bounds from below for smaller allowed tolerances on the eigenvalues of X.

5

Conclusion

In this thesis upper bounds on the kissing number in dimensions $3 - 24$ have been calculated through a semidefinite program obtained via the Delsarte-Goethals-Seidel method. The SDP has been solved using a cutting plane approach, in which iteratively linear programs are solved with the addition of linear cuts each round. A central topic has been the effect of k-sparse eigencuts on computational efficiency, and to that end a Sparse and a Dense strategy have been presented. Due to the size and block diagonal structure of the variable matrices considered, k-sparse eigencuts exist in low number. For this reason the Sparse and Dense strategies performed fairly similar. With both strategies the PSD cone is succesfully approached, where necessary with high precision computations to overcome numerical instabilities. In any case the time spent solving LPs grows quickly with the size of the variable matrix. Allowing less iterations for the interior point method reduced computational effort required. Most importantly, the cuts generated through non-optimal intermediate LP solutions seem to be of higher quality.

A numerical tolerance was allowed on the eigenvalues of the variable matrix, which means the polynomials returned are not strictly valid. An interesting question is if a priori a kissing number upper bound could be bounded for a known eigenvalue tolerance. This would allow a more informed choice of tolerance. The impact of sparsity on cuts could be better studied with a larger computational capacity. For larger matrices it should be possible to generate more as well as sparser cuts. Lastly, not solving each LP till optimality is a promising strategy to reduce the time spent solving the LP as well as generate a higher quality round of cuts. When to stop solving inside an LP as well as the impact on problems of larger size are possible topics for further research.

References

[1] URL: https://sdpa.sourceforge.net/index.html.

[2] Akshayka. *A hierarchy of convex optimization problems*. Nov. 2019. URL: https://commons.wikimedia.org/wiki/File:Hierarchy_compact_convex.png.

[3] V. V. Arestov and A. G. Babenko. “Estimates of the maximal value of angular code distance for 24 and 25 points on the unit sphere in □4”. In: *Mathematical Notes* 68.4 (2000), pp. 419–435. DOI: 10.1007/bf02676721.

[4] Christine Bachoc and Frank Vallentin. “New upper bounds for kissing numbers from semidefinite programming”. In: *Journal of the American Mathematical Society* 21.3 (Nov. 2007), pp. 909–924. DOI: 10.1090/s0894-0347-07-00589-9. URL: https://doi.org/10.1090%2Fs0894-0347-07-00589-9.

[5] Radu Baltean-Lugojan et al. “Scoring positive semidefinite cutting planes for quadratic optimization via trained neural networks”. In: (2019). URL: https://api.semanticscholar.org/CorpusID:208175312.

[6] Jeff Bezanson et al. “Julia: A fresh approach to numerical computing”. In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671.

[7] Stephen P. Boyd. *Linear Matrix Inequalities in system and control theory*. SIAM, 1994.

[8] Robert Bradshaw. *Kissing number in dimension three*. Feb. 2008. URL: https://commons.wikimedia.org/wiki/File:Kissing-3d.png.

[9] Fabrício Caluza Machado and Fernando Mário de Oliveira Filho. “Improving the semidefinite programming bound for the kissing number by exploiting polynomial symmetry”. In: *Experimental Mathematics* 27.3 (2017), pp. 362–369. DOI: 10.1080/10586458.2017.1286273.

[10] Henry Cohn. *Kissing numbers*. URL: https://cohn.mit.edu/kissing-numbers.

[11] P. Delsarte, J. M. Goethals, and J. J. Seidel. “Spherical codes and designs”. In: *Geometriae Dedicata* 6.3 (1977), pp. 363–388. DOI: 10.1007/bf03187604.

[12] Santanu S. Dey, Rahul Mazumder, and Guanyi Wang. “Using ℓ1-relaxation and integer programming to obtain dual bounds for sparse PCA”. In: *Operations Research* 70.3 (2022), pp. 1914–1932. DOI: 10.1287/opre.2021.2153.

[13] Santanu S. Dey et al. “Cutting plane generation through sparse principal component analysis”. In: *SIAM Journal on Optimization* 32.2 (2022), pp. 1319–1343. DOI: 10.1137/21m1399956.

[14] Mikhail Ganzhinov. *Highly symmetric lines*. 2022. arXiv: 2207.08266 [math.FA].

[15] Michel X. Goemans and David P. Williamson. “.879-approximation algorithms for Max Cut and Max 2sat”. In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing; - STOC ’94* (1994). DOI: 10.1145/195058.195216.

[16] Tamás Görbe. *The difficulty of kissing (Cover Picture)*. Nov. 2015. URL: https://tamasgorbe.wordpress.com/2015/11/18/the-difficulty-of-kissing/.

[17] Jump-Dev. *Generic numeric type in jump · issue2025 · Jump-dev/jump.jl*. URL: https://github.com/jump-dev/JuMP.jl/issues/2025.

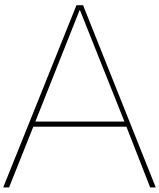
[18] A. Korkine and G. Zolotareff. “Sur les Formes quadratiques”. In: *Mathematische Annalen* 6.3 (1873), pp. 366–389. DOI: 10.1007/bf01442795.

[19] Kartik Krishnan and John Mitchell. “Semi-infinite linear programming approaches to semidefinite programming problems”. In: *Novel Approaches to Hard Discrete Optimization* (2003), pp. 123–142. DOI: 10.1090/fic/037/08.

[20] David de Laat, Fernando Mário de Oliveira Filho, and Frank Vallentin. “Upper bounds for packings of spheres of several radii”. In: *Forum of Mathematics, Sigma* 2 (Sept. 2014). DOI: 10.1017/fms.2014.24. URL: https://doi.org/10.1017%2Ffms.2014.24.

[21] John Leech. “Notes on sphere packings”. In: *Canadian Journal of Mathematics* 19 (1967), pp. 251–267. DOI: 10.4153/cjm-1967-017-0.

[22] Benoît Legat et al. “MathOptInterface: a data structure for mathematical optimization problems”. In: *INFORMS Journal on Computing* 34.2 (2021), pp. 672–689. DOI: 10.1287/ijoc.2021.1067.
[23] Nando Leijenhorst and David de Laat. *Solving clustered low-rank semidefinite programs arising from polynomial optimization*. 2023. arXiv: 2202.12077 [math.OC].
[24] Miles Lubin et al. “JuMP 1.0: Recent improvements to a modeling language for mathematical optimization”. In: *Mathematical Programming Computation* (2023). DOI: 10.1007/s12532-023-00239-3.
[25] John E. Mitchell. “Cutting plane methods and subgradient methods”. In: *Decision Technologies and Applications* (2009), pp. 34–61. DOI: 10.1287/educ.1090.0064.
[26] Hans Mittelmann and Frank Vallentin. “High-Accuracy Semidefinite Programming Bounds for Kissing Numbers”. In: *Experimental Mathematics* 19.2 (Jan. 2010), pp. 175–179. DOI: 10.1080/10586458.2010.10129070. URL: https://doi.org/10.1080%2F10586458.2010.10129070.
[27] Baback Moghaddam, Yair Weiss, and Shai Avidan. “Generalized spectral bounds for sparse LDA”. In: *Proceedings of the 23rd international conference on Machine learning; - ICML ’06* (2006). DOI: 10.1145/1143844.1143925.
[28] N. Mori. *Kissing number in dimension one*. Feb. 2007. URL: https://commons.wikimedia.org/wiki/File:Kissing-1d.svg.
[29] N. Mori. *Kissing number in dimension two*. Feb. 2007. URL: https://commons.wikimedia.org/wiki/File:Kissing-2d.svg.
[30] Oleg R. Musin. *The kissing number in four dimensions*. 2006. arXiv: math/0309430 [math.MG].
[31] A.M Odlyzko and N.J.A Sloane. “New bounds on the number of unit spheres that can touch a unit sphere in n dimensions”. In: *Journal of Combinatorial Theory, Series A* 26.2 (1979), pp. 210–214. ISSN: 0097-3165. DOI: https://doi.org/10.1016/0097-3165(79)90074-8. URL: https://www.sciencedirect.com/science/article/pii/0097316579900748.
[32] Florian Pfender and Günter Ziegler. “Kissing Numbers, Sphere Packings, and Some Unexpected Proofs”. In: *Notices of the American Mathematical Society* (June 2004).
[33] Andrea Qualizza, Pietro Belotti, and François Margot. “Linear programming relaxations of quadratically constrained quadratic programs”. In: *Mixed Integer Nonlinear Programming* (2011), pp. 407–426. DOI: 10.1007/978-1-4614-1927-3_14.
[34] Motakuri V. Ramana. “An exact duality theory for semidefinite programming and its complexity implications”. In: *Mathematical Programming* 77.1 (1997), pp. 129–162. DOI: 10.1007/bf02614433.
[35] James Renegar. *A mathematical view of interior-point methods in convex optimization*. Society for industrial and applied mathematics, 2001.
[36] Jeffrey Sarnoff and JuliaMath. *DoubleFloats*. Version 1.2.2. June 2022. URL: https://github.com/JuliaMath/DoubleFloats.jl.
[37] K. Schütte and B. L. van der Waerden. “Das problem der Dreizehn Kugeln”. In: *Mathematische Annalen* 125.1 (1952), pp. 325–334. DOI: 10.1007/bf01343127.
[38] Hanif D. Sherali, Evrim Dalkiran, and Jitamitra Desai. “Enhancing RLT-based relaxations for Polynomial Programming problems via a new class of V-semidefinite cuts”. In: *Computational Optimization and Applications* 52.2 (2011), pp. 483–506. DOI: 10.1007/s10589-011-9425-z.
[39] Gabor Szegö. *Orthogonal polynomials*. American Mathematical Soc., 1974.
[40] Mathieu Tanneau, Miguel F. Anjos, and Andrea Lodi. “Design and implementation of a modular interior-point solver for linear optimization”. en. In: *Mathematical Programming Computation* (Feb. 2021). ISSN: 1867-2957. DOI: 10.1007/s12532-020-00200-8. URL: https://doi.org/10.1007/s12532-020-00200-8 (visited on 03/07/2021).
[41] Lieven Vandenberghe and Stephen Boyd. “Semidefinite programming”. In: *SIAM Review* 38.1 (1996), pp. 49–95. DOI: 10.1137/1038003.
[42] Xiao-Tong Yuan and Tong Zhang. *Truncated Power Method for Sparse Eigenvalue Problems*. 2011. arXiv: 1112.2679 [stat.ML].



Julia Code - Standard Precision

```
using AbstractAlgebra
using Symbolics
using DelimitedFiles
using ToeplitzMatrices
using BlockDiagonals
using LinearAlgebra
using JuMP
using DataStructures
import MathOptInterface as MOI
import Tulip
setprecision(BigFloat, 1024)

function gegenbauer_polynomials(n::Int, dmax::Int, t)
    n >= 2 || error("n must be >= 2")
    dmax >= 1 || error("dmax must be >= 1")

    a = (n - 3) // 2
    ret = [parent(t)(1), t]
    for k = 2:dmax
        push!(ret, ((2k + 2a - 1) // (k + 2a) * t * ret[end]
                    - (k - 1) // (k + 2a) * ret[end - 1]))
    end

    return ret
end

R, t = PolynomialRing(RealField, "t")

function coeff_b(n::Int, dmax::Int, t)
    b = zeros((dmax+1, dmax+1))

    for j = 1:dmax+1
        R, t = PolynomialRing(RealField, "t")
        geg = gegenbauer_polynomials(n, dmax, t)[j]
        for k = 0:dmax
            b[j, k+1] = BigFloat(coeff(geg, k))
        end
    end

    return b
end

function h(n::Int, i::Int)
    first_column = zeros(BigFloat, 1, n + 1)
    last_row = zeros(BigFloat, 1, n + 1)
    for j = 1:n+1
        if 1 + j - 2 == i
            first_column[j] = BigFloat("1")
```

```
        end
        if n + j - 1 == i
            last_row[j] = BigFloat("1")
        end
    end
    return Hankel(vec(first_column), vec(last_row));
end

function lp_sos(n::Int, dmax::Int)
    a = Matrix{BigFloat}[]
    a_fin = Matrix{BigFloat}[]
    for i = 0:dmax
        push!(a, BlockDiagonal([h(Int(dmax / 2), i), -1/2 * h(Int(dmax / 2 - 1), i - 1) + 1/2
     * h(Int(dmax / 2 - 1), i) - h(Int(dmax / 2 - 1), i - 2)]));
    end
    for i = reverse(1:dmax+1)
        coeff = coeff_b(n, dmax, t)[:, i]
        for j = reverse(1:length(coeff))
            if j > i
                a[i] = a[i] - coeff[j]*a[j]
            end
            if j == i
                a[j] = a[j] / coeff[j]
            end
        end
        k = zeros(BigFloat, dmax + 1, dmax + 1)
        k[i, i] = BigFloat("1")
        push!(a_fin, BlockDiagonal([a[i], k]));
    end
    a_fin = reverse(a_fin)

    open("lp_sos.txt", "w") do file
        ar = zeros(dmax + 1)
        ar[1] = dmax + 1
        writedlm(file, [ar])
        ar[1] = 3
        writedlm(file, [ar])
        if isodd(dmax)
            dim1 = (dmax + 1) / 2
            dim2 = (dmax + 1) / 2
            dim3 = dmax + 1
            writedlm(file, [Int(dim1) Int(dim2) -Int(dim3)])
        else
            dim1 = (dmax + 1 + 1) / 2
            dim2 = (dmax + 1 - 1) / 2
            dim3 = dmax + 1
            ar = zeros(dim3)
            ar[1] = Int(dim1)
            ar[2] = Int(dim2)
            ar[3] = -Int(dim3)
            writedlm(file, [ar])
        end
        writedlm(file, [-1 zeros(1, dmax)])

        for m = 0:length(a_fin)
            for i = 1:2*dmax+2
                for j = i:2*dmax+2
                    line2 = []
                    if m == 0
                        if i == j
                            if i >= dim1 + dim2 + 1
                                val = -1
                            else
                                val = 0
                            end
                        else
                            val = 0
                        end
                    else
                        val = a_fin[m][i,j]
                    end
```

```
                    if val != 0
                        if i >= dim1 + dim2 + 1
                            blocknum = 3
                            pos1 = i - dim1 - dim2
                            pos2 = j - dim1 - dim2
                            line = [m blocknum pos1 pos2 val]
                            for i = 1:length(line)-1
                                push!(line2, Int.(line[i]));
                            end
                            push!(line2, line[length(line)])
                        elseif i >= dim1 + 1
                            blocknum = 2
                            pos1 = i - dim1
                            pos2 = j - dim1
                            line = [m blocknum pos1 pos2 val]
                            for i = 1:length(line)-1
                                push!(line2, Int.(line[i]));
                            end
                            push!(line2, line[length(line)])
                        else
                            blocknum = 1
                            pos1 = i
                            pos2 = j
                            line = [m blocknum pos1 pos2 val]
                            for i = 1:length(line)-1
                                push!(line2, Int.(line[i]));
                            end
                            push!(line2, line[length(line)])
                        end
                    for i = 1:(dim3-5)
                        push!(line2, 0)
                    end
                    writedlm(file, [line2])
                    end
                end
            end
        end
    end
    return a_fin
end

function truncate(v::Vector{Float64}, k::Int)
    ab = abs.(v)
    b = partialsortperm(ab, 1:k, rev=true)
    truncated = zeros(length(v));
    for i = 1:k
        truncated[b[i]] = v[b[i]]
    end
    return truncated
end

function tpmethod(A::Matrix{Float64}, v::Vector{Float64}, k::Int)
    if checkpsd(A) == false
        print("NON PSD")
        return [1]
    end
    v_new = v
    v_new = normalize(v_new)
    v_old = zeros(length(v_new));
    counter = 0
    while abs(norm(v_new - v_old)) >= 1e-12
        if abs(norm(v_new + v_old)) <= 1e-12
            break
        end
        counter = counter + 1
        if counter == 10000
            return v_new
        end
        v_old = v_new
        if norm(A*v_old) == 0
```

```
            print("WARNINGZERO")
            break
        end
        v_new = A*v_old/norm(A*v_old)
        v_new = truncate(v_new, k)
        v_new = normalize(v_new)
    end
    v_new = truncate(v_new, k)
    v_new = normalize(v_new)
    return v_new
end

function sparseround(M::Matrix{Float64}, maxnumsupports::Int, k::Int, mode::Int)
    eigencuts = []
    M_c = copy(M)
    if mode == 2 || mode == 0
        eigenval = real.(eigen(M).values)
        eigenvec = real.(eigen(M).vectors)
        count = 0
        for eig in eigenval
            count = count + 1
            if eig < 0 - 1e-06
                push!(eigencuts, eigenvec[:, count]);
            end
        end
        return eigencuts
    end

    eigenval = real.(eigen(M).values)
    count = 0
    for eig in eigenval
        if eig < 0 - 1e-06
            count = count + 1
        end
    end

    lmin::Float64 = eigen(M).values[1]
    lmax::Float64 = last(eigen(M).values);
    A_psd::Matrix{Float64} = lmax*I - M
    vecmin::Vector{Float64} = eigen(M).vectors[:, 1]
    w::Vector{Float64} = tpmethod(A_psd, vecmin, k)
    if w == [1]
        return sparseround(M, maxnumsupports, k, 0)
    end

    i = 1
    while transpose(w)*M*w < -1e-7 && i < maxnumsupports
        supp = findall(!iszero, w)
        princ = M[supp, supp]
        cut = zeros(length(w));
        lmin = real.(eigen(princ).values[1])
        vmin::Vector{Float64} = real.(eigen(princ).vectors[:, 1])
        for i = 1:length(vmin)
            if abs(vmin[i]) > 1e-09
                cut[supp[i]] = vmin[i]
            end
        end
        push!(eigencuts, cut)
        M::Matrix{Float64} = M - lmin*cut*transpose(cut)
        lmax = last(real.(eigen(M).values));
        A_psd = lmax*I - M
        vecmin = real.(eigen(M).vectors[:, 1])
        w = tpmethod(A_psd, vecmin, k)
        if w == [1]
            return sparseround(M, maxnumsupports, k, 0)
        end
        i = i + 1
    end
    if length(eigencuts) < count || (length(eigencuts) < 3 && length(eigencuts) == count)
        eigencuts = sparseround(M_c, maxnumsupports, k, 0)
    end
```

```
    return eigencuts
end

function matrixreconstruct(dat)
    M_l = []
    for i = 1:(Int(dat[1])+1)
        M = zeros(2*Int(dat[1]), 2*Int(dat[1]))
        push!(M_l, M)
    end
    for i = 5:Int(size(dat)[1])
        m = Int(dat[i, 1])
        block = Int(dat[i, 2])
        if block == 1
            j = Int(dat[i, 3])
            k = Int(dat[i, 4])
        elseif block == 2
            j = Int(dat[i, 3] + dat[3, 1])
            k = Int(dat[i, 4] + dat[3, 1])
        else
            j = Int(dat[i, 3] + dat[3, 1] + dat[3, 2])
            k = Int(dat[i, 4] + dat[3, 1] + dat[3, 2])
        end
        M_l[m+1][j, k] = dat[i, 5]
        M_l[m+1][k, j] = dat[i, 5]
    end
    return M_l
end

function soltomatrix(dim::Int, v::Vector)
    matrix = zeros(2*dim, 2*dim)
    count = 1
    for i = 1:2*dim
        for j = i:2*dim
            if (i <= dim && j <= dim) || i == j
                matrix[i, j] = v[count]
                matrix[j, i] = v[count]
                count = count + 1
            end
        end
    end
    return matrix
end

function checkpsd(M::Matrix)
    M2 = copy(M)
    l_min = eigmin(M2)
    if l_min >= -1e-06
        return true
    else
        return false
    end
end

function cuttingplane(dimension, degree, maxnumsupports, sparsity, mode)
    val_list = []
    total_in_solve = 0
    if mode == 0
        #DENSE
        sparsity_used = (degree + 1) * 2
    else
        #SPARSE
        sparsity_used = sparsity
    end
    model = Model(Tulip.Optimizer)
    MOI.set(model, MOI.RawOptimizerAttribute("IPM_IterationsLimit"), 50)
    set_silent(model)
    lp_sos(dimension, degree)
    dat = readdlm("lp_sos.txt")
    dim = Int(dat[1])
    M_l = matrixreconstruct(dat)
    b = zeros(BigFloat, dim)
```

```
    b[1] = BigFloat(-1)

    @variable(model, x[i = 1:2*dim, j = i:2*dim; (i <= dim && j <= dim) || i == j])
    @objective(model, Max, sum(-1*x[j, j] for j = (dim+1):2*dim))

    variablelist = []
    for j = 1:2*dim
        for k = j:2*dim
            if (j <= dim && k <= dim) || j == k
                push!(variablelist, [j, k])
            end
        end
    end

    for i = 1:length(M_l)
        for j = 1:2dim
            for k = (j+1):2*dim
                M_l[i][j,k] = 2*M_l[i][j,k]
            end
        end
    end

    starting_cuts = []
    @constraint(model, c[i = 1:dim], sum(M_l[i+1][j, k]*x[j, k] for (j,k) in variablelist) ==
     b[i])
    for j = 1:2*dim
        f = @constraint(model, x[j,j] >= 0)
        c = zeros(2*dim)
        c[j] = 1
        push!(starting_cuts, c)
    end
    starting_constraints = ConstraintRef[]
    for (F, S) in list_of_constraint_types(model)
        for con in all_constraints(model, F, S)
            push!(starting_constraints, con)
        end
    end

    optimize!(model)
    val = -MOI.get(model, MOI.ObjectiveValue()) + 1
    push!(val_list, val)
    solution_summary(model)
    sol_v = []
    for i = 1:2*dim
        for j = i:2*dim
            if (i <= dim && j <= dim) || i == j
                push!(sol_v, value(x[i,j]));
            end
        end
    end
    sol_m = soltomatrix(dim, sol_v)
    cutpool = []
    cutpool_tot = []
    it = 1
    cc = 0
    cuts_counter = Dict()
    threshold = 1e-3
    c_it = 0
    while (checkpsd(sol_m) == false && it <= 50000)
        cuts = sparseround(sol_m, maxnumsupports, sparsity_used, mode)
        cuts = unique(cuts)
        println(length(cuts));
        for c in cuts
            push!(cutpool, c)
            push!(cutpool_tot, c)
            cuts_counter[c] = 0;
        end
        cutpool = unique(cutpool);
        cut_m = []
        for i = 1:length(cuts)
            push!(cut_m, cuts[i]*transpose(cuts[i]));
```

```
        end

        mult = ones(2*dim, 2*dim)
        for i = 1:2*dim
            for j = (i+1):2*dim
                mult[i,j] = 2;
            end
        end
        for cut in cuts
            cut_m = cut*transpose(cut)
            d = @constraint(model, sum(mult[j,k]*cut_m[j, k]*x[j, k] for (j,k) in
    variablelist) >= 0)
            set_name(d, string(cut));
        end
        println(length(cutpool));
        for cut in cutpool
            if transpose(cut)*normalize(sol_m)*cut >= threshold && it > 200
                cuts_counter[cut] = cuts_counter[cut] + 1
                if cut in starting_cuts
                    #
                else
                    if cuts_counter[cut] == 2
                        push!(starting_cuts, cut)
                        c_name = string(cut)
                        delete(model, constraint_by_name(model, c_name));
                        deleteat!(cutpool, findfirst(x->x==cut,cutpool));
                        unregister(model, :c_name)
                        delete!(cuts_counter, cut)
                    end
                end
            else
                cuts_counter[cut] = 0
            end
        end
        optimize!(model)
        total_in_solve = total_in_solve + MOI.get(model, MOI.SolveTimeSec())
        println(round(solve_time(model), digits = 5));
        sol_v2 = []
        for i = 1:2*dim
            for j = i:2*dim
                if (i <= dim && j <= dim) || i == j
                    push!(sol_v2, value(x[i,j]));
                end
            end
        end
        sol_m = soltomatrix(dim, sol_v2)
        violated_count = 0
        val = -MOI.get(model, MOI.ObjectiveValue()) + 1
        push!(val_list, val)
        println((MOI.get(model, MOI.ObjectiveValue())));
        if checkpsd(sol_m) == true && c_it == 1
            print(eigmin(sol_m))
            print("PSD reached")
            break
        end
        if checkpsd(sol_m) == true && c_it == 0
            c_it = c_it + 1
            println("switch iteration limit")
            MOI.set(model, MOI.RawOptimizerAttribute("IPM_IterationsLimit"), 300)
            set_attribute(model,  "IPM_CorrectionLimit", 5)
            optimize!(model)
            sol_v2 = []
            for i = 1:2*dim
                for j = i:2*dim
                    if (i <= dim && j <= dim) || i == j
                        push!(sol_v2, value(x[i,j]));
                    end
                end
            end
            sol_m = soltomatrix(dim, sol_v2)
        end
```

```
473         it = it + 1
474     end
475     print(it)
476     println(MOI.get(model, MOI.TerminationStatus()))
477     println(MOI.get(model, MOI.ObjectiveValue()))
478     println("total in solve:")
479     println(total_in_solve)
480     return val_list
481 end
```

B

Julia Code - High Precision

```
using LinearAlgebra
using JuMP
using DataStructures
import MathOptInterface as MOI
using GenericLinearAlgebra
import Tulip
using IterativeRefinement
using DoubleFloats
setprecision(BigFloat, 512)

function truncate(v::Vector{Double64}, k::Int)
    ab = abs.(v)
    b = partialsortperm(ab, 1:k, rev=true)
    truncated = zeros(Double64, length(v));
    for i = 1:k
        truncated[b[i]] = v[b[i]]
    end
    return truncated
end

function tpmethod(A::Matrix{Double64}, v::Vector{Double64}, k::Int)
    v_new::Vector = v
    v_new = normalize(v_new)
    v_old = zeros(Double64, length(v_new));
    counter = 0
    while abs(norm(v_new - v_old)) >= 1e-12
        if abs(norm(v_new + v_old)) <= 1e-12
            break
        end
        counter = counter + 1
        if counter == 1000
            return v_new
        end
        v_old = v_new
        if norm(A*v_old) == 0
            print("WARNINGZERO")
            break
        end
        v_new = A*v_old/norm(A*v_old)
        v_new = truncate(v_new, k)
        v_new = normalize(v_new)
    end
    v_new = truncate(v_new, k)
    v_new = normalize(v_new)
    return v_new
end

cutpool = []

```

```
function sparseround(M::Matrix{Double64}, maxnumsupports::Int, k::Int, mode::Int)
    eigencuts = []
    eigencuts_or = []
    M_c = copy(M)
    M_est = convert(Matrix{Float64}, M)
    ef = eigen(M_est)
    lambda_est_list = real.(ef.values)
    eigenval = []
    eigenvec_or = real.(ef.vectors)
    eigenvec = []
    if mode == 2 || mode == 0
        for i = 1:length(lambda_est_list)
            try
                push!(eigenval, rfeigen(M, convert(Vector{Double64}, real.(ef.vectors[:, i])), Double64(lambda_est_list[i]))[1])
                push!(eigenvec, rfeigen(M, convert(Vector{Double64}, real.(ef.vectors[:, i])), Double64(lambda_est_list[i]))[2])
            catch
            else
                e = 1e-500
                M_adj = M + e*I
                push!(eigenval, rfeigen(M_adj, convert(Vector{Double64}, real.(ef.vectors[:, i])), Double64(lambda_est_list[i]))[1])
                push!(eigenvec, rfeigen(M_adj, convert(Vector{Double64}, real.(ef.vectors[:, i])), Double64(lambda_est_list[i]))[2])
            end
        end
        count = 0
        for eig in eigenval
            count = count + 1
            if eig < 0 - 1e-06
                push!(eigencuts_or, eigenvec_or[:, count]);
                push!(eigencuts, eigenvec[count]);
            end
        end
        return eigencuts
    end

    count = 0
    for eig in lambda_est_list
        if eig < 0 - 1e-06
            count = count + 1
        end
    end

    M2::Matrix{Float64} = convert(Matrix{Float64}, M)
    M3::Matrix{Double64} = copy(M)
    lmin::Double64 = minimum(real(GenericLinearAlgebra._eigvals!(M3)));
    lmax::Double64 = maximum(real(GenericLinearAlgebra._eigvals!(M3)));
    A_psd::Matrix{Double64} = lmax*I - M
    vecmin::Vector{Float64} = eigen(M2).vectors[:, 1]
    w::Vector{Double64} = tpmethod(A_psd, convert(Vector{Double64}, vecmin), k)

    i = 1
    while (transpose(w)*M*w < -1e-7 && i < maxnumsupports)
        supp = findall(!iszero, w)
        princ = M[supp, supp]
        princ2::Matrix{Float64} = convert(Matrix{Float64}, princ)
        princ3::Matrix{Double64} = copy(princ)
        cut::Vector{Double64} = zeros(Double64, length(w));
        lmin = minimum(real(GenericLinearAlgebra._eigvals!(princ3)));
        vmin::Vector{Float64} = real.(eigen(princ2).vectors[:, 1])

        ef = eigen(princ2)
        lambda_est = real(ef.values[1])
        eigenv_est = real.(ef.vectors[:, 1])
        try
            lambda = rfeigen(princ, convert(Vector{Double64}, eigenv_est), Double64(lambda_est))[1]
            eigenv = rfeigen(princ, convert(Vector{Double64}, eigenv_est), Double64(lambda_est))[2]
```

```
            eigenv = normalize(eigenv);
        catch
        else
            e = 1e-500
            princ_adj = princ + e*I
            lambda = rfeigen(princ_adj, convert(Vector{Double64}, eigenv_est), Double64(
    lambda_est))[1]
            eigenv = rfeigen(princ_adj, convert(Vector{Double64}, eigenv_est), Double64(
    lambda_est))[2]
            eigenv = normalize(eigenv);
        end

        cut = zeros(Double64, length(w));
        for i = 1:length(eigenv)
            if abs(eigenv[i]) > 1e-9
                cut[supp[i]] = eigenv[i]
            end
        end
        push!(eigencuts, cut)

        M = M - lmin*cut*transpose(cut)
        M3 = copy(M)
        lmax = maximum(real(GenericLinearAlgebra._eigvals!(M3)));
        A_psd = lmax*I - M
        M2 = convert(Matrix{Float64}, M)
        vecmin = real.(eigen(M2).vectors[:, 1])
        w = tpmethod(A_psd, convert(Vector{Double64}, vecmin), k)
        i = i + 1
    end
    if length(eigencuts) < count|| (length(eigencuts) < 3 && length(eigencuts) == count)
        eigencuts = sparseround(M_c, maxnumsupports, k, 0)
    end
    return eigencuts
end

function matrixreconstruct(dat)
    M_l = Matrix{Double64}[]
    for i = 1:(Int(dat[1])+1)
        M = zeros(2*Int(dat[1]), 2*Int(dat[1]))
        push!(M_l, M)
    end
    for i = 5:Int(size(dat)[1])
        m = Int(dat[i, 1])
        block = Int(dat[i, 2])
        if block == 1
            j = Int(dat[i, 3])
            k = Int(dat[i, 4])
        elseif block == 2
            j = Int(dat[i, 3] + dat[3, 1])
            k = Int(dat[i, 4] + dat[3, 1])
        else
            j = Int(dat[i, 3] + dat[3, 1] + dat[3, 2])
            k = Int(dat[i, 4] + dat[3, 1] + dat[3, 2])
        end
        M_l[m+1][j, k] = dat[i, 5]
        M_l[m+1][k, j] = dat[i, 5]
    end
    return M_l
end

function soltomatrix(dim::Int, v::Vector)
    matrix = zeros(Double64, 2*dim, 2*dim)
    count = 1
    for i = 1:2*dim
        for j = i:2*dim
            if (i <= dim && j <= dim) || i == j
                matrix[i, j] = Double64(v[count])
                matrix[j, i] = Double64(v[count])
                count = count + 1
            end
        end
```

```
    end
    return matrix
end

function checkpsd(M::Matrix)
    M2 = copy(M)
    l = real(GenericLinearAlgebra._eigvals!(M2));
    l_min = minimum(l)
    if l_min >= -1e-06
        return true
    else
        return false
    end
end

function cuttingplane(dimension::Int, degree::Int, maxnumsupports::Int, sparsity::Int, mode::Int)
    val_list = []
    total_in_solve = 0
    if mode == 0
        #DENSE
        sparsity_used = (degree + 1) * 2
    else mode == 1
        #SPARSE
        sparsity_used = sparsity
    end
    model = Tulip.Optimizer{Double64}()
    MOI.set(model, MOI.RawOptimizerAttribute("IPM_IterationsLimit"), 50)
    lp_sos(dimension, degree)
    dat = readdlm("lp_sos.txt", Double64)
    dim = Int(dat[1])
    println(dim)
    M_l = matrixreconstruct(dat)
    b = zeros(Double64, dim)
    b[1] = Double64("-1")

    c = zeros(Double64, Int((dim * dim + dim) / 2 + dim))
    for i = Int(((dim * dim + dim) / 2 + 1)):Int(((dim * dim + dim) / 2 + dim))
        c[i] = Double64("-1")
    end
    x = MOI.add_variables(model, (dim * dim + dim) / 2 + dim)

    MOI.set(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Double64}}(),
           MOI.ScalarAffineFunction(
                [MOI.ScalarAffineTerm(c[i], x[i]) for i = (Int((dim * dim + dim) / 2 + 1)):Int(((dim * dim + dim) / 2 + dim))], Double64("0")),
       );
    MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)

    variablelist = []
    for j = 1:2*dim
        for k = j:2*dim
            if (j <= dim && k <= dim) || j == k
                push!(variablelist, [j, k])
            end
        end
    end

    for i = 1:length(M_l)
        for j = 1:2dim
            for k = (j+1):2*dim
                M_l[i][j,k] = Double64("2")*M_l[i][j,k]
            end
        end
    end

    starting_cuts = []

    for i = 1:dim
```

```
        matrix = M_l[i+1]
        c_vector = zeros(Double64, Int((dim * dim + dim) / 2 + dim))
        j = 0
        k = 1
        z_c = 0
        for z = 1:(dim+1)*(dim)
            if z <= dim*dim
                j = j + 1
                if j >= k
                    z_c = z_c + 1
                    c_vector[z_c] = Double64(matrix[k,j])
                end
                if j % dim == 0 && z < dim*dim
                    j = 0
                    k = k + 1
                end
            else
                z_c = z_c + 1
                j = j + 1
                k = k + 1
                c_vector[z_c] = Double64(matrix[k,j])
            end
        end
        con = MOI.add_constraint(
            model,
            MOI.ScalarAffineFunction{Double64}(MOI.ScalarAffineTerm.(c_vector, x), Double64("
    0.0")),
            MOI.EqualTo(b[i]),
            );
    end

    x_count = 0
    for j = 1:2*dim
        for k = j:2*dim
            if (j <= dim && k <= dim) || j == k
                x_count = x_count + 1
                if j == k
                    f = MOI.add_constraint(
                        model,
                        x[x_count],
                        MOI.GreaterThan(Double64("0"))
                        );
                    c = zeros(2*dim)
                    c[j] = 1
                    push!(starting_cuts, c)
                end
            end
        end
    end

    MOI.optimize!(model)
    print(MOI.get(model, MOI.TerminationStatus()))
    print(MOI.get(model, MOI.ObjectiveValue()))
    val = -MOI.get(model, MOI.ObjectiveValue()) + 1
    push!(val_list, val)
    x_sol = MOI.get(model, MOI.VariablePrimal(), x)
    sol_m = soltomatrix(dim, x_sol)
    cutpool = []
    cutpool_tot = []
    it = 1
    cuts_counter = Dict()
    cuts_index = Dict()
    threshold = 1e-3
    c_it = 0
    while (checkpsd(sol_m) == false && it <= 50000)
        if it > 50
            cuts = sparseround(sol_m, maxnumsupports, sparsity_used, mode)
        else
            cuts = sparseround(sol_m, maxnumsupports, sparsity_used, 0)
        end
        cuts = unique(cuts)
```

```
        println(length(cuts));
        for c in cuts
            push!(cutpool, c)
            push!(cutpool_tot, c)
            cuts_counter[c] = 0;
        end
        cutpool = unique(cutpool);
        cut_m = []
        for i = 1:length(cuts)
            push!(cut_m, cuts[i]*transpose(cuts[i]));
        end

        mult = ones(Double64, 2*dim, 2*dim)
        for i = 1:2*dim
            for j = (i+1):2*dim
                mult[i,j] = Double64("2");
            end
        end

        for cut in cuts
            cut_m = cut*transpose(cut)
            c_vector = zeros(Double64, Int((dim * dim + dim) / 2 + dim))
            j = 0
            k = 1
            z_c = 0
            for z = 1:(dim+1)*(dim)
                if z <= dim*dim
                    j = j + 1
                    if j >= k
                        z_c = z_c + 1
                        c_vector[z_c] = Double64(mult[k,j]*cut_m[k,j]);
                    end
                    if j % dim == 0 && z < dim*dim
                        j = 0
                        k = k + 1
                    end
                else
                    j = j + 1
                    k = k + 1
                    z_c = z_c + 1
                    c_vector[z_c] = Double64(mult[k,j]*cut_m[k,j]);
                end
            end
            d = MOI.add_constraint(
                model,
                MOI.ScalarAffineFunction{Double64}(MOI.ScalarAffineTerm.(c_vector, x),
    Double64("0")),
                MOI.GreaterThan(Double64("0")),
                );
            MOI.set(model, MOI.ConstraintName(), d, string(cut));
            cuts_index[cut] = d
        end

        println(length(cutpool));
        for cut in cutpool
            if transpose(cut)*normalize(sol_m)*cut >= threshold && it > 200
                cuts_counter[cut] = cuts_counter[cut] + 1
                if cut in starting_cuts
                    #
                else
                    if cuts_counter[cut] == 2
                        push!(starting_cuts, cut)
                        c_name = string(cut)
                        MOI.delete(model, cuts_index[cut]);
                        deleteat!(cutpool, findfirst(x->x==cut,cutpool));
                        delete!(cuts_counter, cut)
                    end
                end
            else
                cuts_counter[cut] = 0
            end
```

```
        end

        MOI.optimize!(model)
        println(MOI.get(model, MOI.TerminationStatus()))
        println(MOI.get(model, MOI.ObjectiveValue()))
        x_sol = MOI.get(model, MOI.VariablePrimal(), x)
        val = -MOI.get(model, MOI.ObjectiveValue()) + 1
        push!(val_list, val)
        total_in_solve = total_in_solve + MOI.get(model, MOI.SolveTimeSec());
        println("total")
        println(total_in_solve)
        if total_in_solve > 360000
            break
        end
        println(round((MOI.get(model, MOI.SolveTimeSec())), digits = 5));
        sol_m = soltomatrix(dim, x_sol)
        if checkpsd(sol_m) == true && c_it == 1
            print(eigmin(sol_m))
            print("PSD reached")
            break
        end
        if checkpsd(sol_m) == true && c_it == 0
            c_it = c_it + 1
            println("switch iteration limit")
            MOI.set(model, MOI.RawOptimizerAttribute("IPM_IterationsLimit"), 300)
            MOI.optimize!(model)
            x_sol = MOI.get(model, MOI.VariablePrimal(), x)
            sol_m = soltomatrix(dim, x_sol)
        end
        it = it + 1
    end
    print(it)
    println(MOI.get(model, MOI.TerminationStatus()))
    println(MOI.get(model, MOI.ObjectiveValue()))
    println("total in solve:")
    println(total_in_solve)
    return val_list
end
```