Projekt
Computational Convexity
Berechnung von Dicke (R_1) und Zylinderradius (R_{d-1})

Manuel Mayr Tobias Braunschober

15. Juli 2006

Inhaltsverzeichnis

1 Allgemeine Betrachtungen **1**

1.1 Begriffe 1

1.2 Algorithmische Komplexität 1

2 Zylinderradius **4**

2.1 Ansatz mit SOCP 4

2.1.1 Kugelapproximation 4

2.1.2 Löser 5

2.2 Ansatz ohne SOCP 5

2.2.1 NO SOCP 5

2.2.2 ϵ–Abschätzung 6

2.2.3 NOSOCP adaptiv 7

2.2.4 Verbesserungen 8

2.2.5 Nutze **meb** zu Beginn 10

2.2.6 Megiddo 10

2.2.7 Frühzeitiges Anwenden von checkrv 10

2.2.8 Nutze Schranken lokal 10

2.2.9 Verbesserung von *quotient* durch Fixpunktidee 11

2.2.10 Vergleich zwischen SOCP und noSOCP 12

2.3 Untere Schranken mit SDP für den verankerten Zylinder 13

2.3.1 Relaxierung durch SDPs 14

3 Dicke **17**

3.1 Adaptives Verfahren **minslab_mg** 17

3.2 **minslab_mg** und **checkrv** 17

4 Zylinderradius in l_∞ **19**

4.1 Vorbereitung 19

4.2 Line of Sight 19

4.2.1 Problemstellung 20

4.2.2 Lösung durch lineare Programmierung 20

4.3 Algorithmische Lösung . . . 21
4.3.1 Approximationsvariante . . . 23

5 Codeoptimierung **25**
5.1 Matrixmultiplikation versus Iterationen . . . 25
5.2 A priori Reservierung von Speicher . . . 27
5.2.1 Statische Datenstrukturen . . . 27
5.2.2 Dynamische Datenstrukturen . . . 27
5.3 Optimierung über die `Mex`-Schnittstelle . . . 30
5.4 Prozessorcache . . . 31

6 Testläufe **34**
6.1 Schranken . . . 36
6.2 Zylinder . . . 39
6.2.1 C2 . . . 39
6.2.2 S2 . . . 40
6.2.3 C3 . . . 40
6.2.4 S3 . . . 41
6.2.5 T10 . . . 41
6.2.6 T100 . . . 42
6.2.7 T1000 . . . 42
6.2.8 UF . . . 42
6.2.9 CF . . . 43
6.3 Dicke . . . 43
6.3.1 C2 . . . 43
6.3.2 S2 . . . 44
6.3.3 C3 . . . 44
6.3.4 S3 . . . 44
6.3.5 T10 . . . 45
6.3.6 T100 . . . 45
6.3.7 CF . . . 45
6.3.8 UF . . . 45
6.3.9 S4 . . . 45

7 Analyse **46**
7.1 Schranken . . . 46
7.1.1 2–dimensional . . . 46
7.1.2 3–dimensional . . . 47
7.1.3 >3–dimensional . . . 47
7.2 Dicke . . . 47
7.3 Zylinderradius . . . 47

7.3.1 2–dimensional 48
7.3.2 3–dimensional 48
7.3.3 >3–dimensional 48

8 Weiteres Verbesserungspotential 49
8.1 Heuristiken 49
8.2 Verwendete Programme 49
8.3 Programmiersprache 50
8.4 Programmiertechnik 50

Zusammenfassung

Das Projekt entstand im Rahmen der Vorlesung *Computational Convexity* [1], die im Wintersemester 2005/06 von Rene Brandenberg gehalten wurde. Hierbei wurden Optimale Containment Probleme betrachtet, wie zum Beispiel der Zylinderradius oder die Dicke einer gegebenen Punktmenge. Zu den gegebenen Problemstellungen wurden unter anderem bereits in der Diplomarbeit von Simon Rittsteiger [11] Algorithmen entwickelt. Hierbei sei besonders das *adaptive Verfahren* zu nennen, auf welches im folgenden noch genauer eingegangen wird. Hauptaugenmerk dieses Projekts war in niedriger Dimension die Algorithmen durch Heuristiken, welche die Schranken für die theoretische Laufzeit nicht verbessern, aber auch nicht verschlechtern, zu beschleunigen. Die Einschränkung auf niedrige Dimensionen beruht allein darauf, dass die Berechnungszeit in höheren Dimensionen, zumindest zur Zeit, noch zu groß ist. Die Heuristiken funktionieren in höheren Dimension ebenfalls. Im Speziellen wurden andere Lösungsansätze ausprobiert.

Kapitel 1

Allgemeine Betrachtungen

In diesem Kapitel werden einige Konventionen festgelegt, die es zu beachten gilt, wenn von der Dicke oder dem Zylinderradius einer Punktmenge gesprochen wird. Sie stammen größtenteils aus der Vorlesung [1] und der Diplomarbeit von Rittsteiger [11].

1.1 Begriffe

Definition 1.1 (Der äußere k–Radius R_k)**.** Sei F ein $(d-k)$–dimensionaler affiner Unterraum des $\mathbb{R}^d$ und $\rho \geq 0$, so heißt $F + \rho\mathbb{B}$ k–Zylinder mit Radius ρ. Somit ist der äußere k–Radius[1] $R_k(P)$ einer endlichen Punktmenge P das Minimum der Radien aller k–Zylinder, die P enthalten.

Definition 1.2 (Die Dicke R_1)**.** Die Dicke[2] einer endlichen Punktmenge P sei $2R_1(P)$.

Definition 1.3 (Der Zylinderradius R_{d-1})**.** Der Zylinderradius einer endlichen Punktmenge P sei R_{d-1}.

1.2 Algorithmische Komplexität

Bei der Bestimmung der Dicke und des Zylinderradius handelt es sich um schwere Probleme im Sinne der algorithmischen Komplexität. Unter der üblichen Annahme, dass $\mathbb{P} \neq \mathbb{NP}$, existiert kein Algorithmus, der diese Probleme

[1] Hierbei gilt natürlich $R_k(P) = R_k(conv(P))$, da Zylinder konvexe Mengen sind.

[2] Dies ist Analog zu der Problemstellung den minimalen Abstand zweier stützender Hyperebenen zu finden.

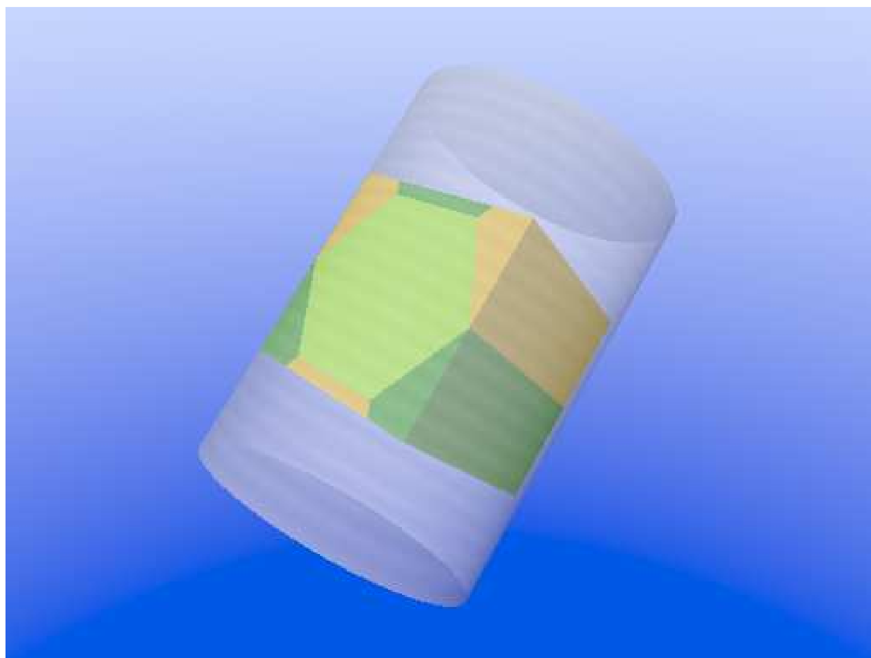

Abbildung 1.1: Körper mit minimalem umspannenden Zylinder.

in polynomieller Laufzeit in der Größe des Inputs löst. Deshalb können wir, für $P \subseteq \mathbb{R}^d$, R_{d-1} und R_1 meist nicht exakt bestimmen, sondern müssen uns mit Approximationen abfinden, die innerhalb einer vorzugebenden Fehlerschranke $\epsilon > 0$ genau arbeiten.

Definition 1.4 (Polynomial Time Approximation Scheme)**.** Eine Familie von Algorithmen ($\mathcal{A}(\epsilon)$, $\epsilon > 0$) heißt Polynomial Time Approximation Scheme (PTAS) zu einem gegebenen Minimierungsproblem mit optimalen Zielfunktionswert $\omega^\star$, wenn für jedes ϵ der Algorithmus $\mathcal{A}(\epsilon)$ in polynomialer Zeit in der Inputgröße eine zulässige Lösung x mit Zielfunktionswert $\omega \leq (1+\epsilon)\omega^\star$ bestimmt.

Für R_{d-1} existieren PTAS, allerdings ist deren Laufzeit in Abhängigkeit von $\frac{1}{\epsilon}$ für R_{d-1} so schlecht, dass keine sinnvollen Berechnungszeiten zu erwarten sind ($dn^{O(\frac{1}{\epsilon^5}\log(\frac{1}{\epsilon})}$[1]).

Bemerkung 1.1. Allerdings kann man versuchen durch effiziente Berechnung von guten oberen und unteren Schranken und entsprechenden Heuristiken für ein gegebenes Problem, den Algorithmus schnell zum Abbruch zu bringen und so bessere Laufzeiten zu erzielen. Dadurch ändert sich zwar nichts an der algorithmischen Komplexität, aber in der Praxis können sich solche Verfahren sehr gut eignen[11].

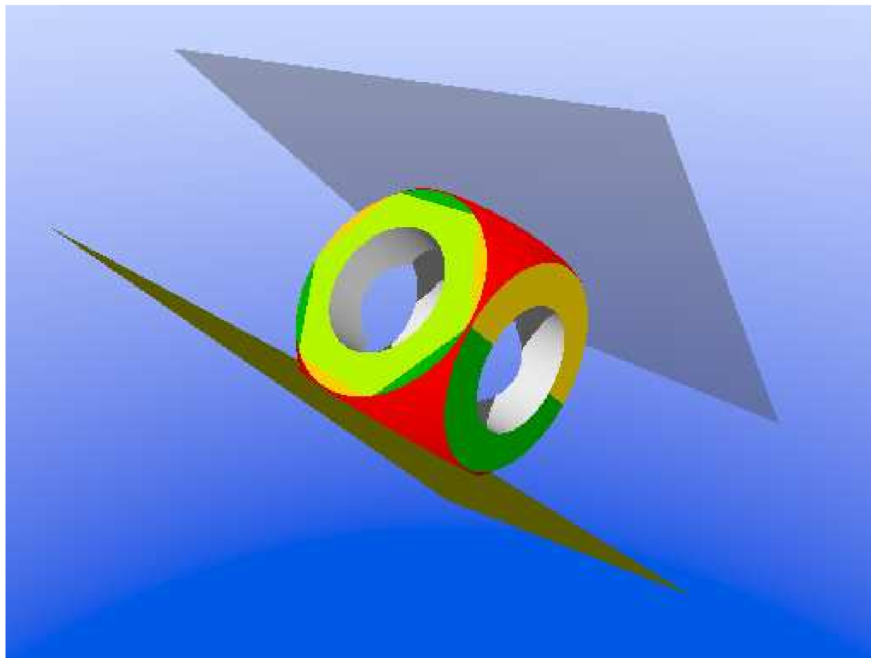

Abbildung 1.2: Dicke eines Körpers.

Kapitel 2

Zylinderradius

Um das Problem der Bestimmung des Zylinderradius $R_{d-1}(P)$ zu lösen wurden zwei unterschiedliche Ansätze verfolgt, die mit verschiedenen Verfahren um Schranken zu berechnen kombiniert wurden.

2.1 Ansatz mit SOCP

Ein Ansatz der in [11] verfolgt wurde ist ein Approximationsalgorithmus für den kleinsten umschließenden Zylinder der in [4] publiziert wurde. Ausgehend vom reinen Optimierungsproblem wurden nach und nach neue Heuristiken zur Verbesserung der Laufzeit hinzugefügt. Der Grundgedanke liegt darin für jeden Richtungsvektor aus der Kugelapproximation ein SOCP zu lösen. Dafür stehen in Matlab Solver zur Verfügung.

2.1.1 Kugelapproximation

Damit man nur endlich viele Richtungen prüfen muss, wird eine Kugelapproximation erzeugt, die aus endlichen vielen normieren Vektoren besteht, welche die Kugeloberfläche approximieren. Diese heißt auch Diskretisierung des Richtungsraums $\mathbb{S}^{d-1} := \{x \in \mathbb{R}; ||x|| = 1\}$. Gesucht ist eine, von der Approximationsgüte ϵ abhängige, endliche Teilmenge $V_d^\epsilon \subset \mathbb{S}^{d-1}$ mit folgenden Eigenschaften: Sei $\theta_\epsilon := \arccos\left(\frac{1}{1+\epsilon}\right)$. Die Menge $V_d^\epsilon \subset \mathbb{S}^{d-1}$ erfülle $\forall x \in \mathbb{R}^d \exists a \in V_d^\epsilon : 0 \leq \angle(a, x) \leq \theta_\epsilon$. Diese Kugelapproximation kann mit **kugelap** erzeugt werden.

2.1.2 Löser

Der Zylinderradius ist der maximale orthogonale Abstand zwischen den Punkten aus P und der Zylinderachse. Der minimale Zylinderradius ist entsprechend der minimale Radius über alle Richtungen der Zylinderachse. Die Idee ist nun, bei der Suche der, nicht notwendig eindeutigen, optimalen Zylinderachse, Punkte aus P nicht mehr orthogonal auf eine Zylinderachse zu projizieren, sondern orthogonal zu einem Richtungsvektor $a \in V_d^\epsilon$. Demnach ist der schiefe Abstand eine Approximation des minimalen Zylinderradius von oben.

2.2 Ansatz ohne SOCP

Bei der Berechnung von **mincyl** und **mincyl_mg**, den Programmen von [11] um den Zylinderradius zu berechnen, ist das größte Problem, dass das Lösen der SOCPs nahezu die gesamte Laufzeit des Programms beansprucht. Dies führt zu der Idee, das Lösen von SOCPs durch ein anderes Verfahren zu ersetzen.

2.2.1 NO SOCP

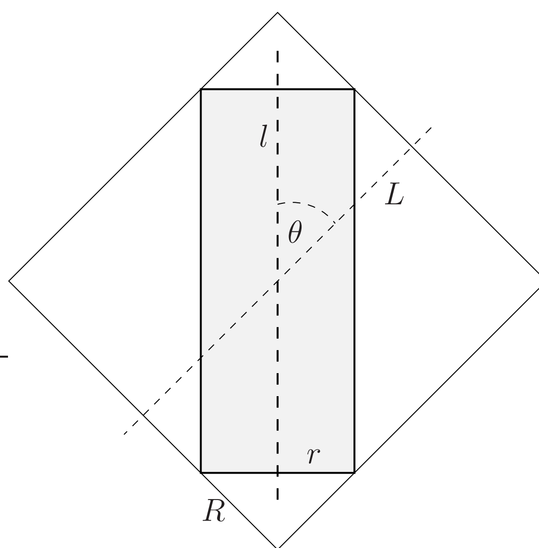

Abbildung 2.1: Vergleich optimaler Zylinder – schiefer Zylinder.

Im Gegensatz zum Ansatz mit SOCPs werden nun nicht mehr schiefe Projektionen zugelassen, sondern feste Projektionsrichtungen vorgegeben. Diese

Projektionsrichtungen sind jeweils senkrecht zu den Richtungsvektoren aus der Kugelapproximation. Entlang dieser Projektionsrichtungen wird das Problem um eine Dimension reduziert. In der $(d-1)$–dimensionalen Hyperebene berechnet man nun die kleinste Umkugel mit einem in Matlab-Code bereits umgesetzten Algorithmus **meb** (Minimum Enclosing Ball[8]). Dieser Algorithmus findet eine Umkugel in $O(\frac{nd}{\epsilon}+\frac{1}{\epsilon^{4.5}}\log\frac{1}{\epsilon})$. Der Radius dieser Umkugel entspricht nun dem Zylinderradius R_{d-1}, wenn die vorgegebene Projektionsrichtung parallel zur Zylinderachse des minimalen umschließenden Zylinders ist. Nachdem man alle vorgegebenen Projektionsrichtungen geprüft hat ist das Minimum dieser Lösungen eine Approximation für den Zylinderradius R_{d-1} von oben.

2.2.2 ϵ–Abschätzung

Gehen wir zunächst davon aus, dass **meb** ohne eigenen Fehler arbeitet.
Aus dem vorgegebenen ϵ muss ein Winkel θ für die Kugelapproximation errechnet werden, für die dann, nach Abarbeitung aller Richtungsvektoren, eine $(1+\epsilon)$–Approximation garantiert ist. Anders als bei dem Ansatz mit SOCPs, wo sich der Winkel aus $\theta = \arccos\left(\frac{1}{1+\epsilon}\right)$ berechnet, ist der Zusammenhang nun der folgende: Um ϵ zu garantieren muss für den Winkel θ gelten: $\theta = \epsilon \cdot \frac{2r}{l}$. Wobei, wie man Abbildung 2.1 entnehmen kann, r dem optimalen Zylinderradius und l der Länge dieses Zylindersegments entspricht. Da diese beiden Parameter unbekannt sind, müssen sie approximiert werden. Im Folgenden wird nur noch von $quotient := \frac{2r}{l}$ gesprochen, da dieser ein wesentlicher Bestandteil für die Geschwindigkeit des Algorithmus ist. Je größer *quotient* ist, desto weniger Richtungsvektoren müssen geprüft werden, da dann die Kugelapproximation für ein gefordertes ϵ gröber gewählt werden kann.

Der Quotient wird von unten abgeschätzt: $quotient = \frac{2r}{l} \geq \frac{2r}{diam(P)} \geq \frac{2r'}{diam'(P)}$. Wobei $r' \leq r$ eine untere Schranke für den optimalen Zylinderradius und $diam'(P) \geq diam(P)$ eine obere Schranke für den Durchmesser von P ist. Der *quotient* bestimmt nur wie viele Richtungsvektoren überprüft werden müssen, also wie schnell der Algorithmus terminiert. Für die Approximationsgüte ist ausschließlich ϵ entscheidend.

Bestimmung der Schranken:

- Obere Schranke für $diam(P)$: (Faktor 2) $O(n)$ [3]

  1. Wähle beliebigen Punkt $p \in P$
  2. Finde den am weitest entfernten Punkt $q \in P$ zu p.

  Dann gilt: $diam(P) \leq 2 \cdot d(p,q)$

- Untere Schranke für $R_{d-1} = r$ (Faktor 4) $O(n)$ [3]

  1. Wähle beliebigen Punkt $p \in P$
  2. Finde den weitest entfernten Punkt $q \in P$ zu p.
  3. Berechne für den Zylinder mit Achse $\overline{p, q}$ über das bereits bestehende Programm **segment**$(P,p,(q-p))$ den Punkt $h \in P$ der zur Achse $\overline{p, q}$ den größten Abstand hat. Diese drei Punkte bilden ein Dreieck. Die kleinste halbe Höhe $\frac{h_{min}}{2}$ dieses Dreiecks ist die gesuchte untere Schranke.

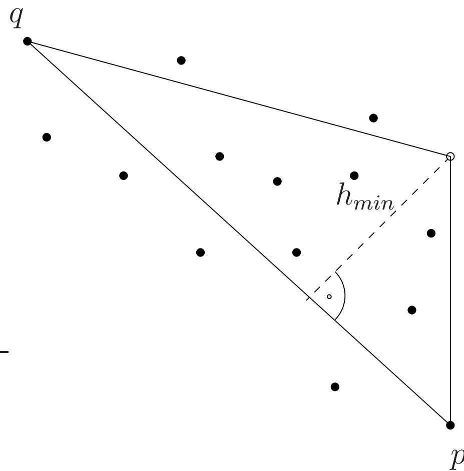

Abbildung 2.2: Untere Schranke für den Zylinderradius mittels Dreiecken.

Als obere Schranke erhalten wir zusätzlich: $r \geq 4 \cdot \frac{h_{min}}{2}$

Da **meb** selbst auch einen Fehler verursacht muss ϵ angepasst werden. Sei dabei ϵ weiterhin die geforderte Approximationsgüte, ϵ_{meb} sei der Fehler des Lösers und ϵ_{kugel} derjenige Anteil, der den Winkel für die Kugelapproximation bestimmt. Nun wird ϵ_{kugel} wie folgt aus den beiden anderen bekannten Größen ϵ_{meb} und ϵ gewonnen. Es gelte $1+\epsilon = (1+\epsilon_{meb})(1+\epsilon_{kugel})$. Wir geben uns nun ein Verhältnis λ zwischen ϵ_{meb} und ϵ_{kugel} vor. Sei nun $\epsilon_{meb} = \lambda\epsilon_{kugel}$ gefordert, so ergibt sich $\epsilon_{kugel} = \lambda\epsilon_{meb} = \frac{-(1+\lambda)+\sqrt{(1+\lambda)^2+4\lambda\epsilon}}{2\lambda}$
Es wurde $\lambda = 0.01$ gewählt.

2.2.3 NOSOCP adaptiv

Wie bereits in den Funktionen **mincyl_mg** und **minslab_mg**[11] wurde auch für **mincyl_mg_noSOCP** der adaptive Gedanke umgesetzt. Es

wird wieder zuerst eine Kugelapproximation zu einem Startepsilon erzeugt und dann sukzessive verfeinert bis ein Zielepsilon erreicht wird. Für die erste Kugelapproximation wurde der *quotient* wie bereits erklärt berechnet. Nachdem alle Richtungsvektoren abgearbeitet wurden ist ein Radius zu diesem Epsilon bekannt: $radius_\epsilon \leq (1+\epsilon)radius_{optimal}$ und offensichtlich gilt $radius_{optimal} \geq \frac{radius_\epsilon}{1+\epsilon}$ Damit ist, wenn $\frac{radius_\epsilon}{1+\epsilon} \geq r'$, eine neue, bessere untere Schranke für den Zylinderradius gefunden. Mit dieser oberen Schranke kann geprüft werden, welche Richtungsvektoren nicht verfeinert werden müssen (**checkrv**).

Im Ansatz ohne SOCP wird ϵ in jedem Schritt um einen Faktor $\frac{dimension}{dimension+2}$ verfeinert. Dieser Faktor wurde in [11] empirisch ermittelt. In diesem Ansatz hat sich jedoch eine kleinere Verfeinerung als besser erwiesen. Wir haben $\frac{dimension+1}{dimension+3}$ als besser ermittelt. Allerdings ist der Winkel der Kugelapproximation hier nicht nur von ϵ, sondern auch noch von *quotient* abhängig. In jedem Schritt kann *quotient* größer werden, also muss sichergestellt werden, dass θ auch wirklich ausreichend kleiner wird. Entsprechend wird gefordert, dass $\theta_{neu} \leq \theta \cdot \frac{1}{2} \cdot \frac{dimension+1}{dimension+3}$ gilt. Diese Mindestverbesserung ist, ebenso wie der andere Faktor, ein weiterer Schalter an dem die Laufzeit, jedoch nicht die algorithmische Komplexität, durch Variation der Werte verbessert werden kann.

2.2.4 Verbesserungen

Entscheidend für die Laufzeit ist die Approximationsgüte von *quotient* und damit die Anzahl der Richtungsvektoren für die auch wirklich ein **meb** gelöst werden muss.

Faktor 4 Approximation für R_{d-1}

Um eine möglichst gute untere Schranke zu bekommen, muss es unser Ziel sein, ein Dreieck zu finden mit einer möglichst großen halben minimalen Höhe. Dieses Dreieck sollte zwei Bedingungen erfüllen:

1. Das Verhältnis der kürzesten zur längsten Seite soll groß sein.

2. Die Seiten sollen alle lang sein.

Als erster Versuch werden Dreiecke erzeugt, die eine möglichst lange Seite haben, in der Hoffnung, dass dann die kürzeste Seite auch lang ist. Dazu wählt man zuerst einen zufälligen Punkt p und berechnet den dazu am weitest entfernten Punkt q. Zu q soll wieder der am weitest entfernte Punkt p' gewählt werden. Dann definieren q und p' die erste Dreiecksseite. Der dritte

Punkt h wird so berechnet, dass der Abstand von h zur Achse durch q, p' maximal ist. Die neue Idee besteht nun darin den zuvor berechneten Punkt h als Startpunkt zu nehmen. Dazu berechnet man wieder den am weitest entfernten Punkt h' von h und bestimmt mittels **segment** h'' zur Achse $\overline{h, h'}$. Da das vorherige Dreieck berechnet wurde gibt es jetzt zwei Kandidaten für die untere Schranke.

Faktor $\sqrt{3}$ Approximation für den Durchmesser

Sei $p' \in P$ ein zufällig gewählter Punkt, und q der zu p' am weitest entfernte Punkt in P und p der zu q am weitest entfernte Punkt in P, und sei $\overline{pq}$ der Abstand zwischen p und q, so ist nach [2] bereits $\sqrt{3} \cdot \overline{pq}$ eine obere Schranke für den Durchmesser.

Faktor 2 Approximation für R_{d-1}

1. Wähle beliebigen Punkt $p' \in P$
2. Finde den weitest entfernten Punkt $q \in P$ zu p'.
3. Finde den weitest entfernten Punkt $p \in P$ zu q.
4. Berechne den Punkt h der zur Achse pq maximalen Abstand hat.
5. Gilt $\overline{pq} \geq \overline{hq}$?
   (a) Ja: So ist $\frac{h_{min}}{2}$ eine untere Schranke für R_{d-1} und h_{min} eine obere Schranke.
   (b) Nein: $h \leftarrow p$ GOTO 4

Mehrere Versuche gute Dreiecke zu finden

Die erste Kugelapproximation hängt neben dem geforderten Startepsilon entscheidend von den gefundenen Schranken ab. Diese Schranken können allerdings bei unglücklicher Wahl des ersten Punktes p sehr schlecht sein. Um dies zu verhindern und da die Berechnung der Dreiecke relativ schnell ist wurde ein Parameter n eingefügt der angibt wie oft diese Dreiecke berechnet werden sollen. Dabei wurde darauf geachtet, dass nicht mehrmals zufällig der gleiche Punkt gewählt wird. Für die Wahl der Punkte werden die Spalten der Punktmatrix P in eine zufällige Reihenfolge gebracht und p nimmt in jedem Schritt den i–ten $i \in [1, n]$ Wert an.

Entsprechend liegt die Chance höher, dass der Punkt für die Faktor 2 Approximation mittiger liegt und damit eine bessere Approximation liefert

wie zuvor. Entsprechend liegt die Chance für die Faktor 4 Approximation höher einen Punkt näher an einer Ecke von $conv(P)$ zu finden, sodass das Dreieck sehr langgezogen ist.

2.2.5 Nutze meb zu Beginn

Eine weitere Idee ist den Löser **meb** zu Beginn dazu zu nutzen die Umkugel für P zu bestimmen. Dies ist dann eine ϵ_{meb}–Approximation für den Umkugelradius, der selbst eine obere Schranke für den halben Durchmesser ist und damit möglicherweise bessere Approximationen liefert als die Dreiecksmethode. Darüberhinaus ist die Laufzeit eines einzelnen **meb** Aufrufs sehr schnell. [10]

2.2.6 Megiddo

In einem späteren Kapitel wird mit **megiddo**[9] ein anderer Ansatz beschrieben den Zylinderradius zu bestimmen. Es existiert auch eine Approximations–Variante des Algorithmus, der zumindest für kleine Dimensionen d und wenige Punkte relativ schnell obere Schranken der Güte $(1+\epsilon)\sqrt{d}$ liefert. Allerdings ist es ein exponentieller Algorithmus. Nur solange $(1+\epsilon)\sqrt{d} \leq 2$ lohnt es sich diesen Ansatz zu verfolgen.
Also: Nutze **megiddo** für die oberen und unteren Schranken wenn $d \leq 4$.

2.2.7 Frühzeitiges Anwenden von checkrv

In **mincyl_mg** wurde **checkrv** erst nach der zweiten Verfeinerung der Kugelapproximation benutzt, da erst nach dem ersten Durchlauf untere und obere Schranken für den Zylinderradius bekannt waren.

In der NOSOCP Variante haben wir allerdings durch die Faktor 2 und Faktor 4 Approximationen bereits - möglicherweise sehr pessimistische - Schranken. Aufgrund dessen ist es möglich, gleich nachdem die erste Kugelapproximation berechnet wurde und vor den Löseraufrufen, zu prüfen welche dieser Richtungsvektoren bereits verworfen werden können. Dadurch können potentiell **meb**–Aufrufe gespart werden.

2.2.8 Nutze Schranken lokal

Ein weiterer Verbesserungsschritt besteht darin, anstelle der globalen Schranken, die Schranken lokal zu nutzen. Die lokalen unteren Schranken für den Zylinderradius sind stets größer oder gleich der globalen unteren Schranke. Aus den lokalen unteren Schranken lässt sich analog ein lokaler *quotient*

errechnen, der stets größer oder gleich dem globalen *quotient* ist. Entsprechend ergeben sich für die einzelnen Kugelkappen größere oder zumindest gleich große Winkel θ bei gleichbleibender ϵ–Garantie.

2.2.9 Verbesserung von *quotient* durch Fixpunktidee

Nach dem Aufruf des Lösers **meb** verändern sich die lokale untere Schranke *loc_lower* und das lokale ϵ.

$$loc_lower = \frac{1}{1+\epsilon} \cdot radius_{meb}$$

Wird nun durch eine bessere untere Schranke der lokale *quotient* vergrößert, so verkleinert sich in gleichem Maße das lokale ϵ:

$$\epsilon = \frac{\theta}{quotient}$$

Ein kleineres ϵ führt jedoch wieder auf eine größere untere Schranke:

$$loc_lower = \frac{1}{1+\epsilon} \cdot radius_{meb}$$

Zusammen ergibt dies:

$$F = \frac{loc_lower_{neu}}{loc_lower_{alt}} \quad , F \geq 1$$

$$quotient_{neu} = F \cdot quotient_{alt}$$

$$\epsilon_{neu} = \frac{\epsilon_{alt}}{F}$$

Mit $G = \frac{1+\epsilon_{alt}}{1+\frac{\epsilon_{alt}}{F}} \quad , G \geq 1$ ergibt sich:

$$loc_lower_{neu} = G \cdot loc_lower_{alt}$$

damit lässt sich dann wiederum die globale untere Schranke und der globale *quotient* bestimmen, die beide größer oder gleich wie zuvor sind. Es verbessern sich also potenziell pro Iteration die untere Schranke, *quotient* und auch ϵ. Da nicht garantiert ob und wie schnell die Fixpunktiteration terminiert werden nur einige Schritte ausgeführt. Hierfür wurde ein Parameter gesetzt, der festlegt wie oft diese - sehr schnelle - Berechnung gemacht werden soll. Zusätzlich wurde ein weiterer Parameter gesetzt, der festlegt, wie groß die Mindestverbesserung von *quotient* in jedem Schritt sein soll. Wird diese Mindestverbesserung unterschritten so wird der Durchlauf der Schleife vorzeitig abgebrochen. Hierbei hat sich gezeigt, dass nahezu nie mehr als fünf Schritte notwendig waren, da sich dann die drei Werte kaum noch verbessert haben.

2.2.10 Vergleich zwischen SOCP und noSOCP

Der größte Nachteil des Ansatzes ohne SOCPS im Vergleich zu jenem mit SOCP ist die größere Anzahl der zu lösenden Problemen, die jedoch unterschiedlich schnelle Löser besitzen. Die Zahl der Löseraufrufe bei SOCP ist abhängig von ϵ, bei der Version ohne SOCP ist sie zusätzlich abhängig von *quotient*.

$$\theta_{SOCP}(\epsilon) = \arccos\left(\frac{1}{1+\epsilon}\right)$$
$$\theta_{noSOCP}(\epsilon, quotient) = \epsilon \cdot quotient$$

Demnach ergibt sich bei gleichem Winkel:

$$\epsilon_{noSOCP} = \frac{1}{quotient} \cdot \arccos\left(\frac{1}{1+\epsilon_{SOCP}}\right)$$

Beispiel

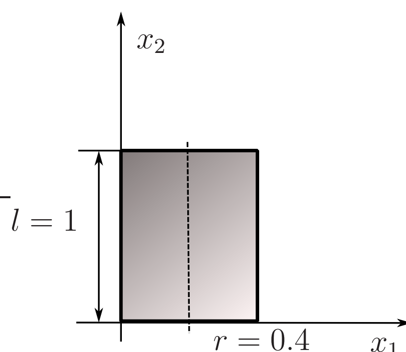

Abbildung 2.3: Vergleich SOCP – noSOC.

Wurde bei dem Ansatz mit SOCP bei einem geforderten $\epsilon_{SOCP} = 0.01$ ein Winkel $\theta_{SOCP} = 0.1408$ benötigt, so ergibt sich bei gleichem Winkel, dass für $quotient = 0.8$ dieser Winkel bereits bei $\epsilon_{noSOCP} = 0.1761$ erforderlich ist. Die SOCP–Version erreicht also bei gleicher Anzahl der Aufrufe, in diesem Fall, eine 17.6044 mal bessere Approximationsgüte, wenn von Beginn an mit dem optimalen $quotient = 0.8$ gerechnet wurde. Dieser Faktor um den die Approximationsgüte mit SOCP besser ist wird je kleiner ϵ wird um so größer.

Folgende Tabelle zeigt einen Überblick für dieses Beispiel:

ϵ_{SOCP}	$Faktor$
0.1	5.3712
0.01	17.6044
0.001	55.8784
0.0001	176.7693

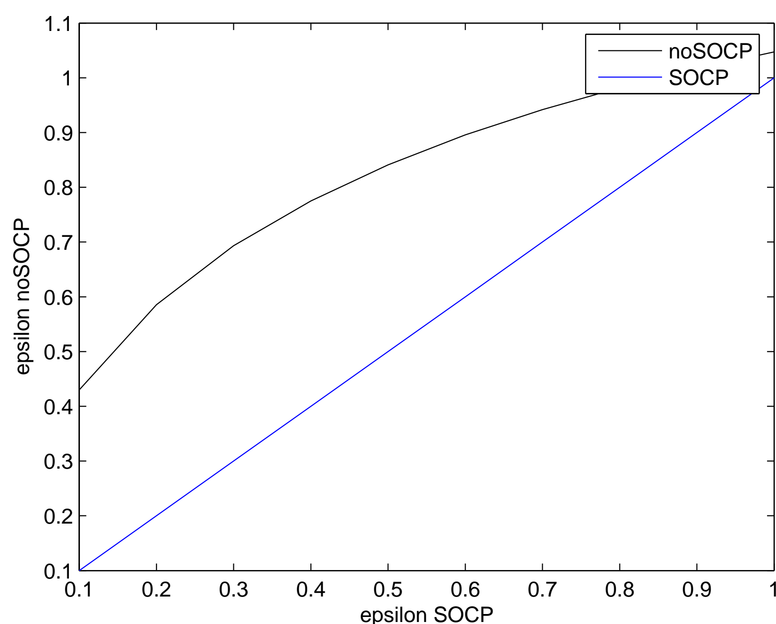

Abbildung 2.4: SOCP vs noSOCP.

Zur Illustration ein paar Bilder für unterschiedliche ϵ–Intervalle (2.4. 2.5, 2.6, 2.7 und 2.8), in denen der Unterschied zwischen $0.8 \cdot x$ und $\arccos\left(\frac{1}{1+x}\right)$ dargestellt wird.

2.3 Untere Schranken mit SDP für den verankerten Zylinder

Eine interessante Verallgemeinerung der linearen Programmierung und SOCPs stellt die die semidefinite Programmierung (SDP) dar. Für unsere Zwecke eignen sich SDPs, um bessere untere Schranken für den optimalen Zylinderradius zu finden.

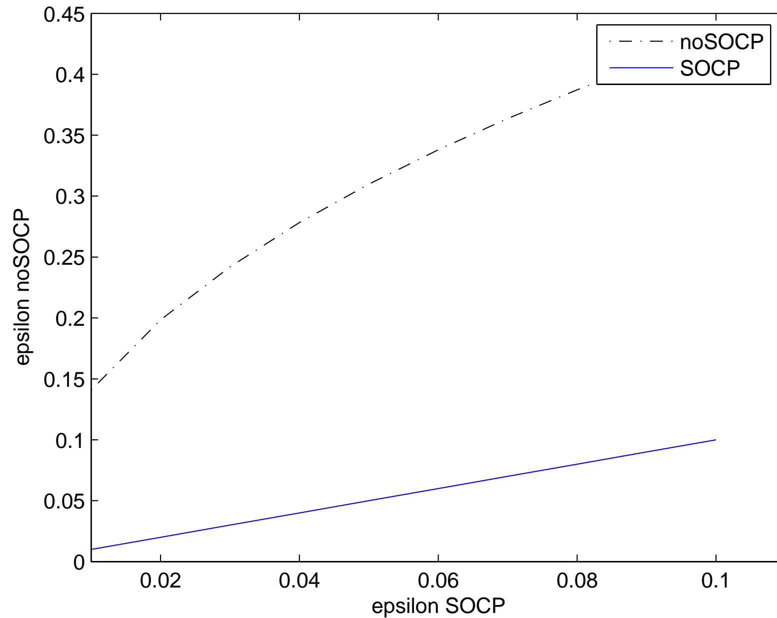

Abbildung 2.5: SOCP vs noSOCP.

2.3.1 Relaxierung durch SDPs

Um solche unteren Schranken mit SDPs berechnen zu können müssen wir das Problem relaxieren, woraus sich folgendes SDP ergibt.

Sei $P \subseteq \mathbb{R}^d$, $X = xx^T$, mit $x \in \mathbb{R}^d$ und $a \in V_\epsilon^d$. Des weiteren sei $\beta = \cos\theta$ aus der Kugelapproximation. Mit Tr bezeichnen wir die Spur einer Matrix und $\succeq$ bedeutet in diesem Kontext positiv semidefinit.

$$\begin{aligned} r^* &= \min\ r \\ Tr(pp^T X) &\leq r && \forall p \in P \\ Tr(X) &= d-1 \\ Tr(aa^T X) &\leq \beta^2 \\ E - X &\succeq 0 \\ X &\succeq 0 \end{aligned} \tag{SDP1}$$

Mit SDP1 erhalten wir eine untere Schranke $r \leq R(P)^2$ für den Zylinderradius in Richtung a für die entsprechende Kugelkappe mit Winkel β.

Allerdings gibt es hierbei ein Problem: Diese Berechnung ist nur gültig, wenn der Ankerpunkt auch wirklich auf der Zylinderachse liegt.

Die lokale Berechnung der unteren Schranken mittels SDP innerhalb der

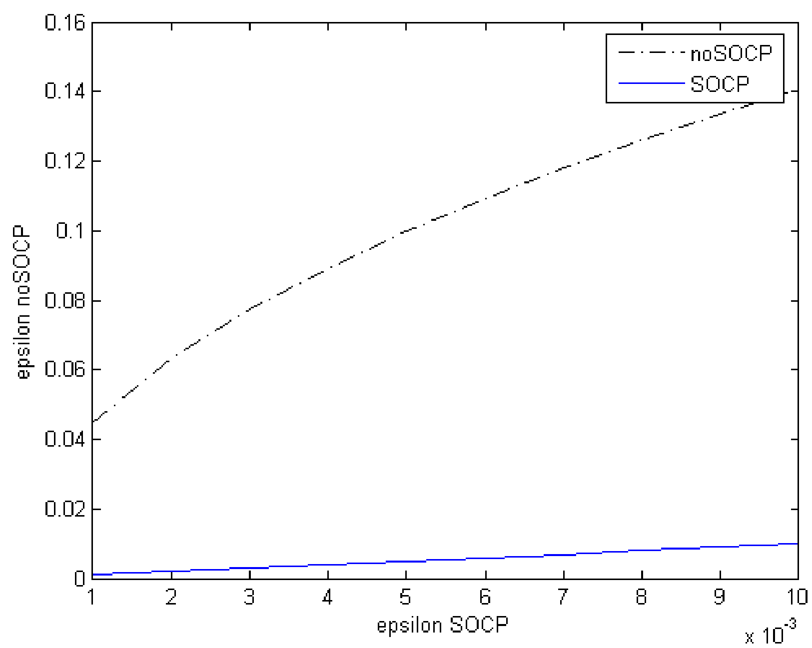

Abbildung 2.6: SOCP vs noSOCP.

einzelnen Kugelkappen und entsprechenden Richtungsvektoren wurde sowohl in der Version mit als auch ohne **SOCP** in einer neuen Version eingearbeitet. Diese Version funktioniert selbstverständlich nur, wenn auch wirklich ein verankerter Zylinder vorliegt.

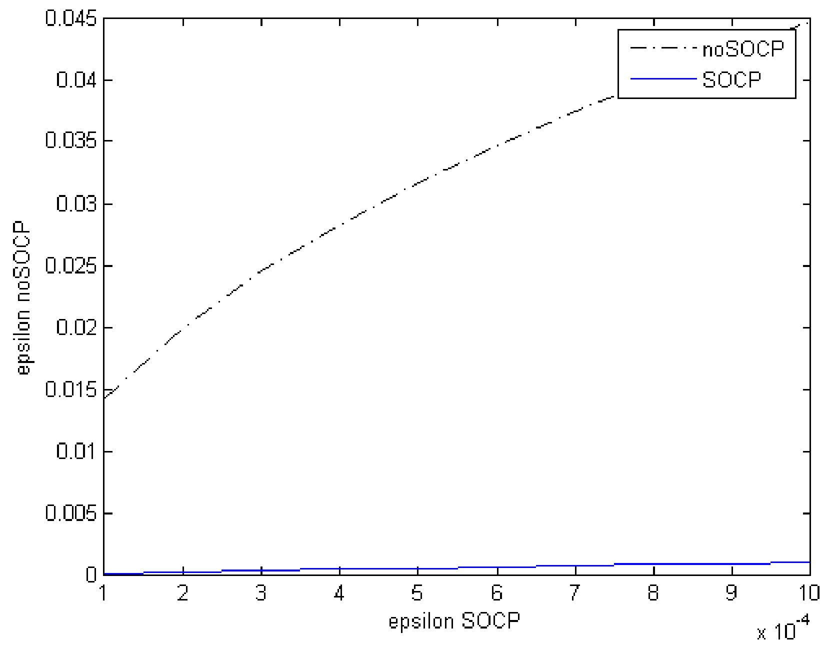

Abbildung 2.7: SOCP vs noSOCP.

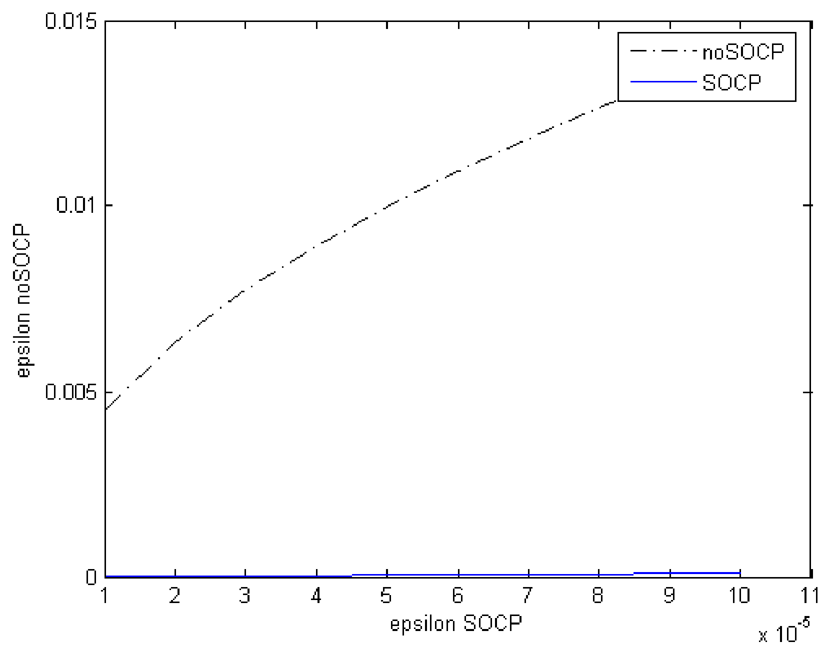

Abbildung 2.8: SOCP vs noSOCP.

Kapitel 3

Dicke

Ähnliche Ideen, wie bei der Bestimmung von R_{d-1}, die im vorherigen Kapitel näher erläutert wurden, finden auch bei der Berechnung der Dicke Anwendung. Die betrachteten Ansätze dazu findet man in [5] und [11].

3.1 Adaptives Verfahren minslab_mg

Als ersten Einstieg in die Thematik und als erste Schritte haben wir die Ideen des adaptiven Verfahren von **mincyl_mg** auf das Programm **minslab** zur Berechnung der Dicke umgesetzt. Der Algorithmus verfeinert jetzt die Kugelapproximation ebenso wie in **mincyl_mg** nur noch lokal dort wo es nötig ist.

3.2 minslab_mg und checkrv

Eine weitere Verbesserung die in **minslab_mg** eingearbeitet wurde ist die Anwendung von **checkrv** um Richtungsvektoren zu verwerfen oder zumindest zu berechnen ab wann sie verworfen werden können. Dazu muss jedoch beachtet werden, dass der Zusammenhang ein anderer ist wie in **mincyl_mg**.

Beschreibung für **mincyl_mg**: Seien hierfür: ρ eine obere Schranke für $R_{d-1}(P)$ und $p, q \in P$. Ferner sei $v := p - q$ und $y \neq 0$ ein beliebiger Vektor gleicher Dimension und bezeichne $\alpha = \angle(v, y)$ der Winkel zwischen v und y. Sei y^* die optimale Zylinderachse und $a \in V_d^\epsilon$ ein Richtungsvektor aus der Kugelapproximation.

Ein Zylinder um P mit Achsenrichtung y hat dann mindestens den Radius $r = \frac{d(p,q)\sin\alpha}{2}$. Damit ist eine untere Schranke für den Zylinderradius in Abhängigkeit der Zylinderachse gefunden. Ist $r > \rho$ kann y keine optimale

Achsenrichtung mehr sein. Daraus folgt, dass y^* mit v keinen größeren Winkel bildet, als $\alpha = \alpha(\rho) := \arcsin \min\left\{1, \frac{2\rho}{d(p,q)}\right\}$. Demnach kann y^* nicht in der Kugelkappe um a mit Radius θ_ϵ liegen, falls gilt: $\angle(v,a) > \alpha + \theta_\epsilon$. Ist dies der Fall so kann der Richtungsvektor a verworfen werden. Daraufhin setzt **checkrv** nun ρ auf Null, wenn a verworfen werden kann, und bestimmt ansonsten das größte ρ, ab dem der Richtungsvektor verworfen worden wäre. Gilt $\angle(\rho) + \theta_\epsilon > \angle(v,a) > \theta_\epsilon$, so wird a in diesem Schritt nicht verworfen. Wird später eine kleinere obere Schranke ρ gefunden so verkleinert sich auch $\alpha(\rho)$. Dazu wird zu a ein $\rho_{check}(a)$ berechnet, das bestimmt ab welcher Schranke ρ der Vektor a in **mincyl_mg** ausgelassen werden kann. Dazu muss gelten:

$$\begin{array}{lll} & \alpha(\rho_{check}(a)) & < & \angle(v,a) - \theta_\epsilon \\ \Rightarrow & \arcsin \frac{2\rho_{check}(a)}{d(p,q)} & < & \angle(v,a) - \theta_\epsilon \\ \Rightarrow & \rho_{check}(a) & < & \frac{d(p,q)}{2}\sin(\angle(v,a) - \theta_\epsilon) \end{array}$$

Diese Rechnung gilt solange $\angle(v,a) \in [0, \frac{\pi}{2}]$. Dies ist immer erfüllt, da in der Kugelapproximation nur eine Kugelhälfte betrachtet wird. Gilt $\angle(v,a) < \theta_\epsilon$, so soll $\rho_{check}(a) = 0$ gesetzt werden. Beschreibung für **minslab_mg**: Für zwei Punkte $p, q \in P$ und $v := q - p$, haben zwei Stützhyperebenen an P, die orthogonal zu x stehen, mindestens den Abstand $d(p,q)\cos(\angle(x,v))$. Ist schon eine obere Approximation $\delta \leq d(p,q)$ der Dicke bekannt, so kann man in **minslab_mg** alle Vektoren $a \in V_d^\epsilon$ unberücksichtigt lassen, für welche gilt: $\angle(a,v) < \arccos(\frac{\delta}{d(p,q)}) - \theta_\epsilon$. [11]

Kapitel 4

Zylinderradius in l_∞

Analog zur Betrachtung des Zylinderradius in l_2 betrachten wir nun R_{d-1} in l_∞, mit zugrunde liegender Maximumsnorm.

4.1 Vorbereitung

Sei $\|\cdot\|_\infty$ die Maximumsnorm definiert als $\|u\|_\infty = \max\{u^i : i \in \{1, \ldots, d\}\}$ und sei $\mathbb{B}^d_\infty = \left\{x \in \mathbb{R}^d : \|x\|_\infty \leq 1\right\}$.

Definition 4.1 (Zylinderradius in l_∞)**.** Sei $P \subset \mathbb{R}^d$ eine Punktmenge, $F \subset \mathbb{R}^d$ ein 1–dimensionaler affiner Unterraum und $\rho > 0$. Der Zylinderradius R_{d-1} in l_∞ ist definiert als

$$R_{d-1}(P) = \min\left\{\rho \geq 0 : P \subset F + \rho\mathbb{B}^d_\infty\right\}.$$

Das Finden von $R_{d-1}(P)$ in l_∞ stützt sich auf eine Bemerkung in [1].

Bemerkung 4.1. Da $x \in R_k(P)\mathbb{B} + F \Leftrightarrow (x + R_k(P)\mathbb{B})) \cap F \neq \emptyset$ kann der äußere k–Radius auch wie folgt dargestellt werden:

$$R_k(P) = \min\left\{\begin{array}{l}\rho \geq 0 : \exists F \subset \mathbb{M}\ (d-j)\text{–dim. affiner Teilraum,}\\ \text{sodass } (x + \rho\mathbb{B}) \cap F \neq \emptyset, \forall x \in P\end{array}\right\} \tag{4.1}$$

4.2 Line of Sight

Wir wollen nun einen Algorithmus angeben der einen 1–dimensionalen affinen Unterraum findet, der n Boxen schneidet, oder beweist, dass es keinen solchen geben kann.[9]

4.2.1 Problemstellung

Definition 4.2 (Boxen)**.** Seien $a, b \in \mathbb{R}^d$, dann ist eine Box gegeben durch

$$B(a,b) = \left\{x \in \mathbb{R}^d : a \leq x \leq b\right\}.$$

Gegeben seien n Boxen $B^i = B(a_i, b_i) \subset \mathbb{R}^d$, $i \in \{1, \ldots, n\}$. Finde einen 1–dimensionalen affinen Unterraum $l \subset \mathbb{R}^d$, sodass für alle $i \in \{1, \ldots, n\}$ $l \cap B^i \neq \emptyset$, oder beweise, dass es kein solches l gibt.

Sei $l = u + tv : t \in \mathbb{R}$, mit $u, v \in \mathbb{R}^d$ und B eine Box, so ist offensichtlich $B \cap l \neq \emptyset$ genau dann, wenn ein $t \in \mathbb{R}$ existiert, sodass

$$a^i \leq u^i + tv^i \leq b^i,\ \forall i \in \{1, \ldots, d\}. \tag{4.2}$$

4.2.2 Lösung durch lineare Programmierung

Lemma 4.1. *Seien $B^k = B(a_k, b_k)$ nicht leere Boxen, dann existiert ein 1–dimensionaler affiner Unterraum $l \subset R^d$, sodass $B^k \cap l \neq \emptyset$ für alle $k \in \{1, \ldots, n\}$, genau dann, wenn eine Partition $I_0 \uplus I_+ \uplus I_- = \{1, \ldots, d\}$ und $x, y \in \mathbb{R}^d$ existieren, sodass*

(i) Für alle $i \in I_0$, $\max_{k\in\{1,\ldots,d\}} a_i^k \leq \min_{k\in\{1,\ldots,d\}} b_i^k$

(ii) Für alle $i \in I_+$, $x^i > 0$ und für alle $i \in I_-$, $x^i < 0$

(iii) Für alle $i, j \in I_+$ und für $k \in \{1, \ldots, n\}$

$$a_k^i x^i - y^i \leq b_k^j x^j - y^j.$$

(iv) Für alle $i, j \in I_-$ und für $k \in \{1, \ldots, n\}$

$$a_k^i x^i - y^i \geq b_k^j x^j - y^j.$$

(v) Für alle $i \in I_+$ und $j \in I_-$ und für $k \in \{1, \ldots, d\}$

$$b_k^i x^i - y^i \geq b_k^j x^j - y^j \text{ und } a_k^j x^j - y^j \geq a_k^i x^i - y^i.$$

Wenn eine Partition $I_0 \uplus I_+ \uplus I_- = \{1, \ldots, d\}$ und $x, y \in \mathbb{R}^d$ existieren, dann kann $l = \{u + tv : t \in \mathbb{R}\}$, $u, v \in \mathbb{R}^d$ wie folgt konstruiert werden. Für alle $i \in I_0$

$$\max_{k\in\{1,\ldots,d\}} a_k^i \leq \min_{k\in\{1,\ldots,d\}} b_k^i$$

und $v^i = 0$. Für $i \in I_+ \uplus I_-$ setze

$$u^i = \frac{1}{x^i} \text{ und } v^i = \frac{x^i}{y^i}.$$

Offensichtlich sind einige Ungleichungen in 4.1 strikt. Dies kann man leicht umgehen, indem folgendes lineares Optimierungsproblem löst:

$$\begin{aligned} &max\ \xi \\ -x^i &\leq -\xi && \forall i \in I_+ \\ x^i &\leq -\xi && \forall i \in I_- \\ a_k^i x^i - y^i - b_k^j x^j + y^j &\leq 0 && \forall i, j \in I_+ \\ b_k^j x^j - y^j - a_k^i x^i + y^i &\leq 0 && \forall i, j \in I_- \\ b_k^j x^j - y^j - b_k^i x^i + y^i &\leq 0 && \forall i \in I_+, \forall j \in I_- \\ a_k^i x^i - y^i - a_k^j x^j + y^j &\leq 0 && \forall i \in I_+, \forall j \in I_- \end{aligned} \tag{Megiddo}$$

Für jede Partition müssen, um $x, y \in \mathbb{R}^d$ zu finden, nicht mehr als $2d^2n+d$ lineare Ungleichungen in maximal $2d$ Variablen gelöst werden.

Satz 4.1. *Für ein konstantes d, kann das Problem des Findens einer „line of sight" in $O(n)$ gelöst werden.*

4.3 Algorithmische Lösung

Nun haben wir alle Voraussetzungen, um den optimalen Zylinderradius in l_∞ zu berechnen. Sei $P \subset \mathbb{R}^d$ eine Punktmenge und $\hat{P}_\rho = \left\{x + \rho\mathbb{B}_\infty^d : x \in P\right\}$.

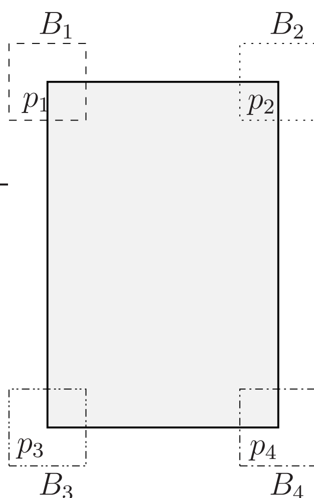

Abbildung 4.1: Megiddo findet keine Linie, die alle Boxen schneidet.

Algorithm 1 Berechne R_{d-1} in l_∞

Require: $P \subset \mathbb{R}^d$ eine endliche Punktmenge
Ensure: $r = R_{d-1}(P)$ in l_∞

1: $upperBound \leftarrow$ obere Schranke von $R_{d-1}(P)$
2: intervall $\leftarrow [0, radius]$
3: aufPunkt $\leftarrow (0, \ldots, 0) \in \mathbb{R}^d$
4: richtung $\leftarrow (1, \ldots, 1) \in \mathbb{R}^d$
5: neueMitte $\leftarrow -1$
6: mitte $\leftarrow \frac{\text{intervall}(1)+\text{intervall}(2)}{2}$
7: **while** neueMitte $\neq$ mitte **do**
8: mitte $\leftarrow \frac{\text{intervall}(1)+\text{intervall}(2)}{2}$
9: $\hat{P}_r \leftarrow \left\{x + r\mathbb{B}^d : x \in P\right\}$
10: Löse Megiddo mit $\hat{P}_r$
11: **if** es gibt einen 1-dim. linearen affinen Unterraum, der alle Boxen in $\hat{P}_r$ schneidet **then**
12: intervall(2) $\leftarrow$ mitte
13: **else**
14: intervall(1) $\leftarrow$ mitte
15: **end if**
neueMitte $\leftarrow \frac{\text{intervall}(1)+\text{intervall}(2)}{2}$
16: **end while**
17: **return** neueMitte

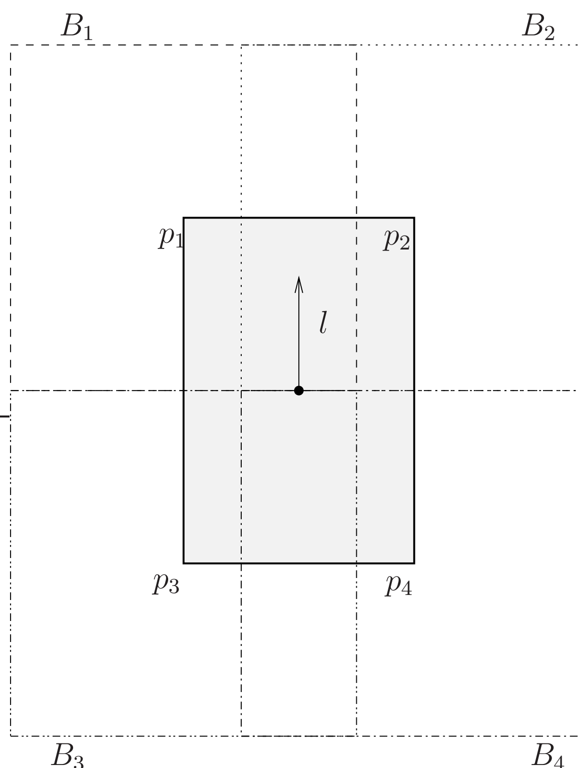

Abbildung 4.2: Megiddo findet eine (nicht eindeutige) Linie, die alle Boxen schneidet.

Offensichtlich liefert uns Megiddo die Antwort, ob es einen 1–dimensionalen affinen Unterraum F gibt, der alle Boxen in $\hat{P}_\rho$ schneidet. Mit einem algorithmischen Ansatz, wie der binären Suche können wir nun leicht den optimalen Zylinderradius berechnen. Folgende Notation in Pseudocode illustriert dieses Vorgehen.

4.3.1 Approximationsvariante

Im eben beschriebenen Algorithmus wird bis auf Rechengenauigkeit überprüft ob $neueMitte \neq Mitte$ gilt.

```
while ( neueMitte ~= Mitte )
   ...
end
```

Man kann sehr viele Schritte der binären Suche auslassen, wenn man anstatt auf Gleichheit zu prüfen folgendes vergleicht: Dazu sei u der linke Intervallrand – für diesen Wert gibt es keine „line of sight“ – und o der rechte Intervallrand – für diesen Wert gibt es eine „line of sight“. Startet man **Megiddo**

mit einem $\epsilon > 0$ so bricht der Algorithmus bereits ab, wenn die Intervallbreite $o - u$ den Wert $\epsilon \cdot u$ unterschreitet. Entsprechend entsteht durch die Approximationsvariante ein weiterer Fehler von $1 + \epsilon$.

```
while ( ((o - u)/u) > eps )
   ...
end
```

Kapitel 5

Codeoptimierung

Matlab ist in seiner Konzeption auf Matrizenrechnung ausgelegt und dementsprechend diesbezüglich optimiert worden. *Matlab* bietet zwar ausreichende Funktionalität um eigene Datenstrukturen zu erstellen, aber dies ist für kleinere Projekte nicht akzeptabel, da dies eine gewisse Schwerfälligkeit in der Entwicklung bedeutet und meistens werden diese Möglichkeiten nicht genutzt, sodass vorwiegend auf *Matlab*–Code zurückgegriffen wird. Bei der Erstellung von Code in dieser Umgebung lassen sich hier einige einfache Regeln angeben, die bei der Erstellung von effizientem Code, wenn möglich angewendet werden sollten. Im Laufe dieses Kapitels werden wir auch einige Strategien beziehungsweise Algorithmen analysieren. Wir verwenden hier die amortisierte Analyse von Algorithmen, da diese meist besser die Laufzeiten in der Praxis wiedergibt, als die höchst theorielastige *worst–case*–Analyse.

5.1 Matrixmultiplikation versus Iterationen

Da, wie bereits erwähnt *Matlab* intern nur mit Matrizen arbeitet und in seiner Konzeption auf das Arbeiten mit Matrixstrukturen ausgelegt ist und diese effizient bearbeitet sind naive `for`–Schleifen durch Matrixmultiplikationen zu ersetzen.

Regel 5.1 (Matrixmultiplikationen sind iterativen Anweisungen vorzuziehen)**.** *Iterative Anweisungen (wie* `for`*–Schleifen) sollten, wenn dies möglich ist durch Matrixmultiplikationen ersetzt werden.*

Beispiel 5.1. *Algorithmus zum Finden des Maximalabstands eines Punktes in einem Körper P zu einem gegebenen Punkt p. Sei $P \subseteq \mathbb{R}^d$ eine endliche Punktmenge. Wir wollen nun zu einem gegebenen Punkt p den Maximalabstand $n = \max_{\hat{p} \in P} \|p - \hat{p}\|_2$ in der euklidischen Norm finden.*

Dies kann man, wie obige Formulierung bereits vermuten lässt, ganz einfach mit `for`*–Schleifen lösen, doch da wir die Fähigkeiten Matlab's im Umgang mit Matrizen nutzen wollen, können wir das Problem leicht umformulieren. Folgender Algorithmus in Pseudocode zeigt, wie sich dies bewerkstelligen lässt:*

Algorithm 2 Berechne $\max_{\hat{p}\in P} \|p - \hat{p}\|_2$

Require: $P \subset \mathbb{R}^d$ eine endliche Punktmenge und einen Punkt $p \in P$
Ensure: $n = \max_{\hat{p}\in P} \|p - \hat{p}\|_2$

1: $X \leftarrow P - \left(p \cdot \underbrace{(1, \dots, 1)}_{\text{Vektor mit } |P|\text{–Spalten}} \right)$
2: $\hat{X} \leftarrow X^T X$
3: $n \leftarrow \sqrt{\max\left\{ diag \hat{X} \right\}}$
4: **return** n

Allerdings sollte man auch mit derlei Optimierungen vorsichtig sein, da solche Transformationen in Matrixoperationen unter Umständen sehr speicherraubend sein können. Wir wollen dies an einem konkreten Beispiel nachvollziehen:
Nehmen wir an wir hätten auf unserem Rechner 1 GB Arbeitsspeicher zur Verfügung. Nehmen wir weiterhin an, unser Betriebssystem verbraucht davon $\frac{1}{3}$. Die Zahlendarstellung in *Matlab* sei `double-precision` und so nimmt eine Zahl 8 Byte an Speicherplatz in Anspruch. Eine einfache Rechnung zeigt, wie groß die Matrix sein darf, die im Arbeitsspeicher gehalten werden kann. Konvertieren wir zunächst die Größe des Arbeitsspeichers in die Bytedarstellung, 1 GB sind demnach $1 \cdot 1024^2$ Bytes, davon verbleiben uns $\left\lfloor \frac{2}{3} \cdot 1024^3 \right\rfloor = 682 \cdot 1024^2$ Bytes für unsere Matrix. Die Anzahl der Zahlen, die wir im verbleibenden Speicher ablegen können sind also $\left\lfloor \frac{682 \cdot 1024^2}{8} \right\rfloor = 85 \cdot 1024^2$. Die Größe der Matrix erhalten wir, indem aus aus dieser Zahl die Wurzel berechnet $\left\lfloor 1024 \cdot \sqrt{85} \right\rfloor = 9044$ wird. Wir können also maximal eine quadratische Matrix der Dimension 9044×9044 erzeugen, die im Arbeitsspeicher gehalten werden kann. Überschreitet die Matrix diese Größe, so tritt durch die Virtualisierung des Speichers, das in moderneren Betriebssystemen übliche Swapping[1] ein. Dieser Vorgang ist im allgemeinen sehr langsam, im Vergleich zu den schnellen Zugriffsmöglichkeiten auf den Arbeitsspeicher. Hier ist die Technik basierend auf `for`–Schleifen der Transformation in Matrixoperationen vorzuziehen.

[1]Seiten des Arbeitsspeichers werden auf den Sekundärspeicher ausgelagert

5.2 A priori Reservierung von Speicher

Die Speicherreservierung ist in *Matlab* ein heikles Thema, da zunächst nicht bekannt ist wie diese intern gehandhabt wird. Wir klassifizieren die folgenden Datenstrukturen als *statisch* und *dynamisch*. Der Unterschied hierbei ist, dass statische Strukturen sich nicht mehr in ihrer Größe ändern dürfen, also immer die selbe Anzahl an Elementen fassen können. Dynamische Strukturen werden bei bedarf einfach vergrößert, wenn man Elemente hinzufügt.

5.2.1 Statische Datenstrukturen

Statische Datenstrukturen sind Strukturen fester Größe, das heißt, die Größe der Struktur ist vorab bekannt und wird im Laufe des Programms nicht mehr verändert. Als einfache Regel, die es im Zuge der effizienten Programmierung zu beachten gilt könnte man folgende nennen:

Regel 5.2 (A priori Reservierung bei bekanntem Speicherplatz)**.** *Ist der Speicherplatz einer Datenstruktur (e.g. eines Arrays oder einer Matrix) bereits vorab bekannt, so ist der Speicher für diese schon vor dessen Nutzung zu reservieren.*

Die Beachtung dieser Regel kann unter Umständen große Laufzeiteinbußen, welche bei sukzessivem Einfügen von Elementen ohne vorheriger Reservierung von Speicher auftreten, wettmachen. Bild 5.1 zeigt, dass eine a priori Reservierung sehr viel Zeit sparen kann, da die dynamische Allokation von Speicher in `Matlab` intern offensichtlich mit einer quadratischen Laufzeit beim Einfügen von n Elementen operiert.

5.2.2 Dynamische Datenstrukturen

Bei dynamischen Datenstrukturen ist die Größe der entsprechenden Struktur vorab noch nicht bekannt. Sie kann also nach belieben verändert werden. Ein typisches Problem hierbei ist, dass zur Handhabung von dynamischen Strukturen statische Strukturen verwendet werden, deren Größe vorab bekannt sein muss. Die folgenden Punkte zeigen, welche Probleme und kritische Operationen es hierbei gibt (es wird nur auf Arrays eingegangen, da hierbei die Probleme recht deutlich sichtbar werden):

- Typisches Problem bei der Verwendung von Arrays: Größe N des Arrays muss deklariert werden und ist damit fest.

- Falls die Anzahl der zu speichernden Einträge $n < N$: Verschwendung des Speicherplatzes.

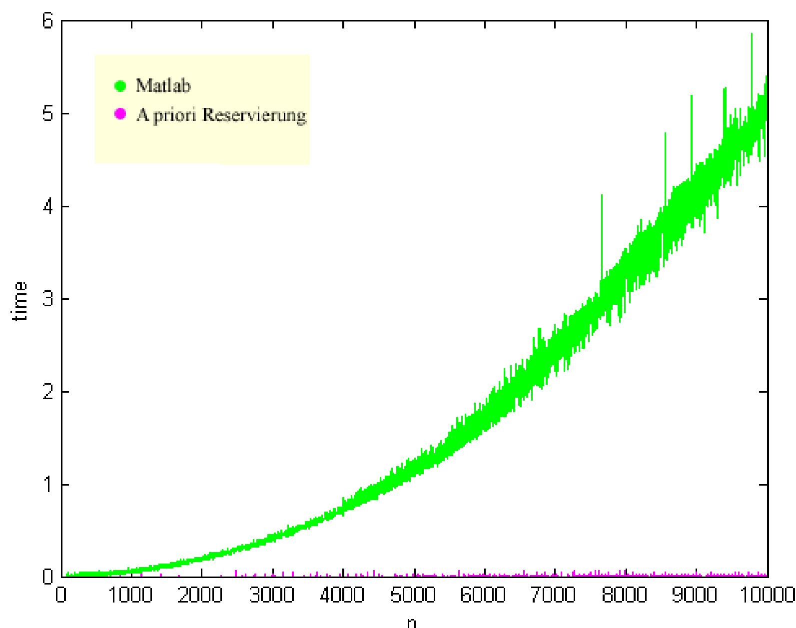

Abbildung 5.1: Vergleich zwischen a priori Reservierung von Speicher und sukzessivem Hinzufügen von Werten.

- Lösung: Dynamische Allokation von Speicherplatz.
- Kritische Operation: Füge ein neues Element in ein Array $\mathcal{A}$ bei $n = N$ ein:
  Verfahre wie folgt:
  - Initialisiere ein neues Array $\mathcal{B}$ der Größe mN mit $m \in \mathbb{N}\backslash\{0\}$.
  - Kopiere $\mathcal{A}$ nach $\mathcal{B}$(Abbildung 5.2 illustriert dieses Vorgehen).
  - Setze den Zeiger von $\mathcal{A}$ auf $\mathcal{B}$.
- Komplexität der Speicherung einer Tabelle $\mathcal{S}$ der Größe n:
  - Im schlechtesten Fall dauert das Einfügen von n Elementen $\Theta(n)$.
  - Das heißt insgesamt $O(n^2)$.

Wir beweisen im Folgenden zwei Sätze zu gängigen Vergrößerungsstrategien. Die *multiplikative Vergrößerungsstrategie* initialisiert ein neues Array $\mathcal{B}$ das um einen Faktor $m \in \mathbb{N}\backslash\{0\}$ größer ist als $\mathcal{A}$. Die Elemente aus $\mathcal{A}$ werden nach $\mathcal{B}$ kopiert und $\mathcal{A}$ wird auf $\mathcal{B}$ gesetzt. Die Vergrößerung wird natürlich

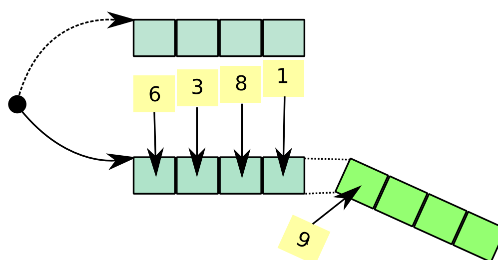

Abbildung 5.2: Verdopplung des Speichers beim Hinzufügen eines neuen Elements.

nur dann durchgeführt, wenn das Array $\mathcal{A}$ voll ist.
Die *additive Vergrößerungsstrategie* initialisiert, wie die *multiplikative* ein um eine Konstante $l \in \mathbb{N} \backslash \{0\}$ größeres Array $\mathcal{B}$. Auch hier werden die Elemente aus $\mathcal{A}$ nach $\mathcal{B}$ kopiert. Die Vergrößerung wird auch hier nur dann vollzogen, wenn das $\mathcal{A}$ bereits voll ist.

Satz 5.1 (Multiplikative Vergrößerungsstrategie)**.** *Es sei $\mathcal{S}$ eine Tabelle, die als dynamisches Array $\mathcal{A}$ (mit multiplikativer Vergrößerungsstrategie) implementiert ist. Die Laufzeit von n Einfügeoperationen in $\mathcal{S}$ für den Fall, dass $\mathcal{S}$ anfangs leer ist und $N = 1$ für die Größe von $\mathcal{A}$ gilt, ist $O(n)$.*

Beweis. Wir verwenden die Bankkontomethode der amortisierten Analyse. Es sei oBdA vorausgesetzt, dass $m = 2$ als Faktor für die Vergrößerung zur Rate gezogen wird. Die Komplexität der Einfügeoperation (ohne Vergrößerung) sei 1. Die Komplexität für die Vergrößerung von k auf $2k$ seien k Schritte (für das Kopieren). Wähle G als eine geeignete Sparstrategie. Zu zeigen: G ist eine geeignete Sparstrategie, das heißt $T(n) = \sum_{i=1}^{n} t_i \leq nG$.
Zunächst ein Beispiel:

$$\sum_{i=1}^{n} t_i = \underbrace{1}_{1} + \overbrace{(1+1)}^{2} + \underbrace{(2+1)}_{3} + \overbrace{1}^{4} + \underbrace{(1+4)}_{5} + \overbrace{1}^{6} + \dots$$

Man kann schnell erkennen, dass

$$\begin{aligned} T(n) &= \sum_{[0\leq i\leq n]} t_i \\ &= n + \sum_{[0\leq i\leq \lfloor \log_2(n-1)\rfloor+1]} 2^i \\ &= n + \frac{1-2^{\lfloor \log_2(n-1)\rfloor+2}}{1-2} \\ &\leq n + 2^{\log_2(n-1)+2} - 1 \\ &= n + 4(n-1) - 1 = 5(n-1) \\ &< 5n \end{aligned}$$

Daraus ergibt sich dass $G = 5$ eine geeignete Sparstrategie ist und somit $T(n) = \sum_{[0\leq i\leq n]} t_i < 5n$. Wir kommen als zum Schluss, dass die amortisierte Komplexität einer Einfügeoperation $\frac{T(n)}{n} \in O(1)$ und damit $T(n) \in O(n)$ (ein etwas weniger formaler Beweis findet sich in [6]). □

Satz 5.2 (Additive Vergrößerungsstrategie)**.** *Sei $l > 0$ eine beliebige additive Vergrößerungskonstante. Sei $\mathcal{S}$ eine Tabelle, die als dynamisches Array $\mathcal{A}$ mit additiver Vergrößerungsstrategie implementiert ist. Die Laufzeit von n Einfügeoperationen ist $O(n^2)$.*

Beweis. Beweis analog zu Satz 5.1. □

Satz 5.1 zeigt also, dass wir bestenfalls mit linearer Laufzeit rechnen können, wenn die Größe des Arrays nicht vorab bekannt ist.
Da schon bei den Tests in Abschnitt 5.2 relativ klar ersichtlich wurde, dass Matlab eine additive Vergrößerungsstrategie anwendet wurde in Matlab ein `Array`–Objekt erstellt welches in amortisiert linearer Zeit beim Ausführen von n Basisoperationen (Einfügen von Elementen) arbeitet. Folgendes Experiment in Bild 5.3 zeigt den Vergleich. Hierbei werden auf der Abszissenachse die Anzahl n der sukzessive hinzugefügten Elemente eingetragen und auf der Ordinatenachse die beanspruchte Zeit.

Bemerkung 5.1. `Matlab` verwendet die schlechtere additive Strategie anstelle der hier vorgestellten multiplikativen Strategie!

5.3 Optimierung über die `Mex`-Schnittstelle

In `Matlab` gibt es neben der Möglichkeit Programme in der hauseigenen Programmiersprache auch noch die Möglichkeit `C` oder `Fortran` Code auszuführen. Dazu kann ein Interface namens `Mex` benutzt werden, das es erlaubt einen

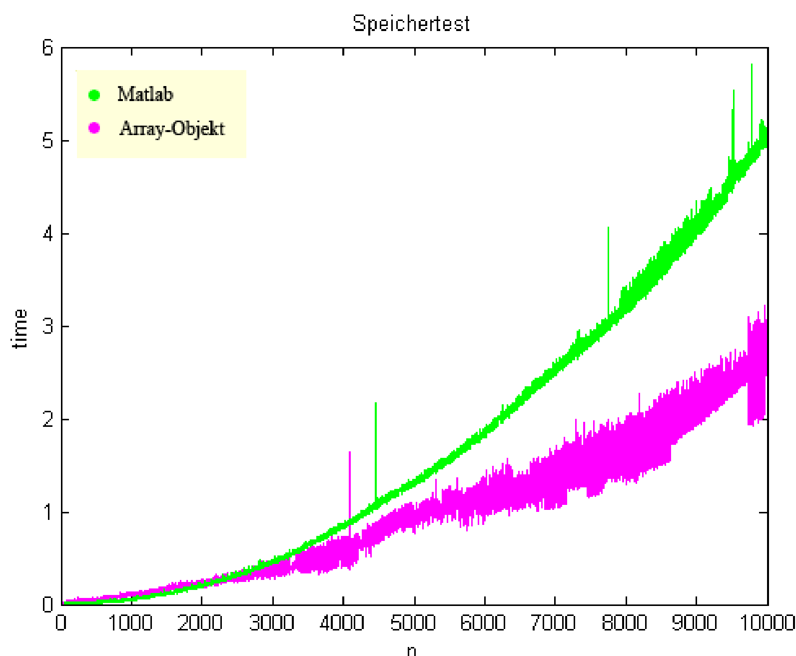

Abbildung 5.3: Vergleich zwischen dem Objekt mit multiplikativer Vergrößerungsstrategie und den internen Routinen von Matlab

Algorithmus in `C` oder `Fortran` zu formulieren. Damit kann durch den Einsatz von Zeigern sehr effizient programmiert werden. Allerdings ist der Code auch etwas fehleranfälliger als `Matlab`–Code, vor allem, wenn man auf die performante Speicherverwaltung von `C` zurückgreift ohne sich auf `Matlabs` Garbage–Collector zu verlassen.
Mittels `Mex`–Schnittstelle kann aber sehr viel Zeit gewonnen werden. Bild 5.4 zeigt, wie unterschiedlich im Zeitverhalten sich der selbe Code, einmal in `Matlab` und einmal in `C` auswirkt.

5.4 Prozessorcache

Matrizen in Matlab werden nicht, wie dies intuitiv erscheinen möchte, in zweidimensionalen Arrays gespeichert, sondern ganz einfach spaltenweise vektorisiert(Abbildung 5.5). Dies bringt uns dazu uns einmal mit der Hardware des Computers auseinanderzusetzen. Moderne Prozessoren sind aufgrund des Leistungsengpasses der Busarchitektur (Abbildung 5.6) mit einem Hochgeschwindigkeitscache ausgestattet. Dieser Cache enthält die Wörter, auf die zuletzt zugegriffen wurde. Alle Speicheranforderungen passieren diesen Ca-

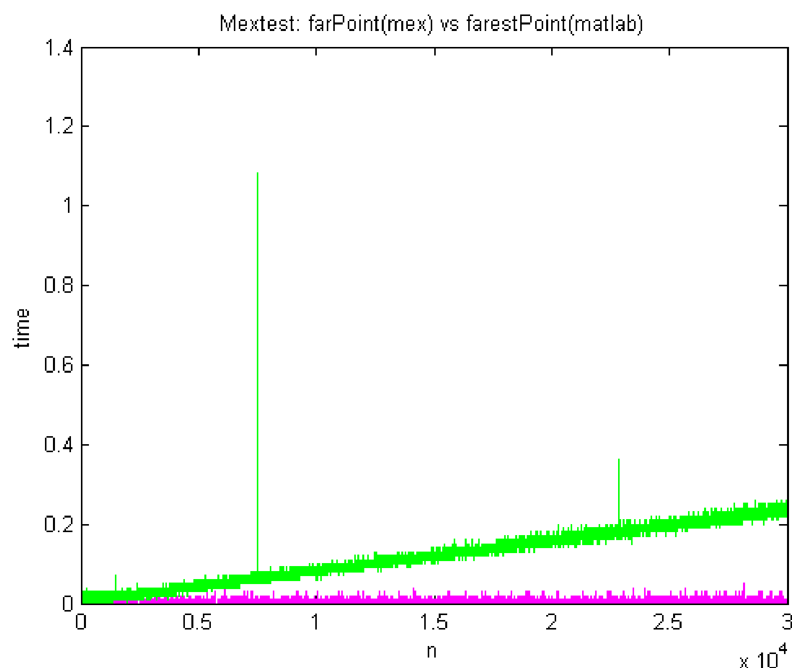

Abbildung 5.4: Vergleich zwischen Matlab- und C-Code.

che. Befindet sich das zu lesende Wort im Cache, so antwortet dieser direkt an die CPU und es erfolgt keine Anforderung über den Bus. Im anderen Fall muss der Speicher natürlich über den Bus angefordert werden. Das Fazit ist, dass, wenn Iterationen durch Arrays getätigt werden, so sollten Sprünge vermieden werden, da der Cache mit ziemlicher Sicherheit die darauffolgende Speicheradresse bereithält. Bei Iterationen durch Matrizen sollte spaltenweise iteriert werden, da Matlab diese spaltenweise vektorisiert, wie man an der Matrixdarstellung in `Mex` sehen kann.

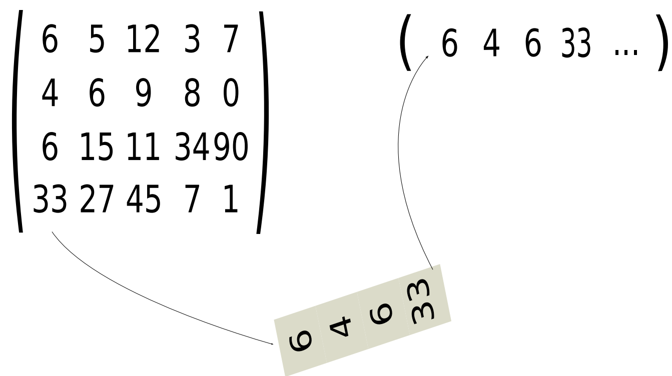

Abbildung 5.5: Vektorisieren einer Matrix.

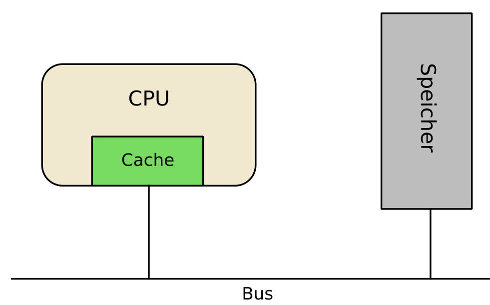

Abbildung 5.6: Busarchitektur mit Prozessor und Speicher

Kapitel 6

Testläufe

In diesem Kapitel werden die Laufzeittests aufgelistet. Alle Tests wurden auf dem gleichen Rechner ausgeführt.
AMD Athlon MP 2600+
2.13 GHz
768 MB RAM
Microsoft Windows XP Professional SP2

Folgende Problemstellungen wurden untersucht:

- Zylinder

Z22	Zylinder mit 22 Punkten auf dem Rand
Z202	Zylinder mit 202 Punkten auf dem Rand
Z2002	Zylinder mit 2002 Punkten auf dem Rand

- Simplexe

S2	2–dimensionales reguläres Simplex nur mit Eckpunkten
S3	3–dimensionales reguläres Simplex nur mit Eckpunkten
S4	4–dimensionales reguläres Simplex nur mit Eckpunkten

- Einheitswürfel

E2	2–dimensionaler Einheitswürfel nur mit Eckpunkten
E3	3–dimensionaler Einheitswürfel nur mit Eckpunkten
E4	4–dimensionaler Einheitswürfel nur mit Eckpunkten

- gestreckter Einheitswürfel mit den Seitenlängen $1, 2, \ldots, 2$

C2	2–dimensionaler gestreckter Einheitswürfel nur mit Eckpunkten
C3	3–dimensionaler gestreckter Einheitswürfel nur mit Eckpunkten
C4	4–dimensionaler gestreckter Einheitswürfel nur mit Eckpunkten

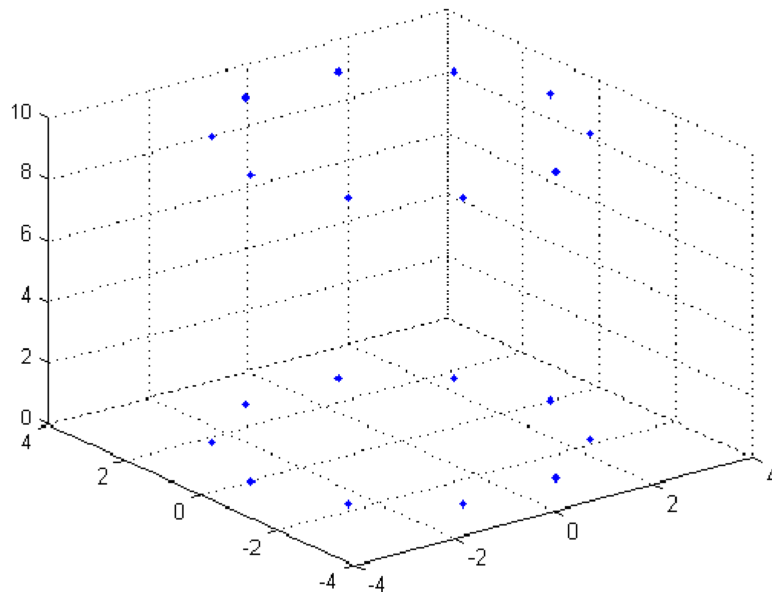

Abbildung 6.1: Zylinder mit 22 Punkten

- andere Körper

UF	2–dimensionaler Einheitswürfel mit 100004 Punkten
CF	C2 mit zusätzlichen 100000 Punkten

Im Folgenden bezeichne:
usR die untere Schranke für den Radius
osR die obere Schranke für den Radius
osD die obere Schranke für den Durchmesser als obere Schranke für die Länge des Zylindersegments.
mincyl: Finde Zylinderradius
minslab: Finde Dicke
mg : Adaptiver Ansatz
SOCP,noSOCP: Löse mit SOCPs oder mit **meb**
SDP,noSDP: Nutze SDP für lokale Schranken oder nicht.

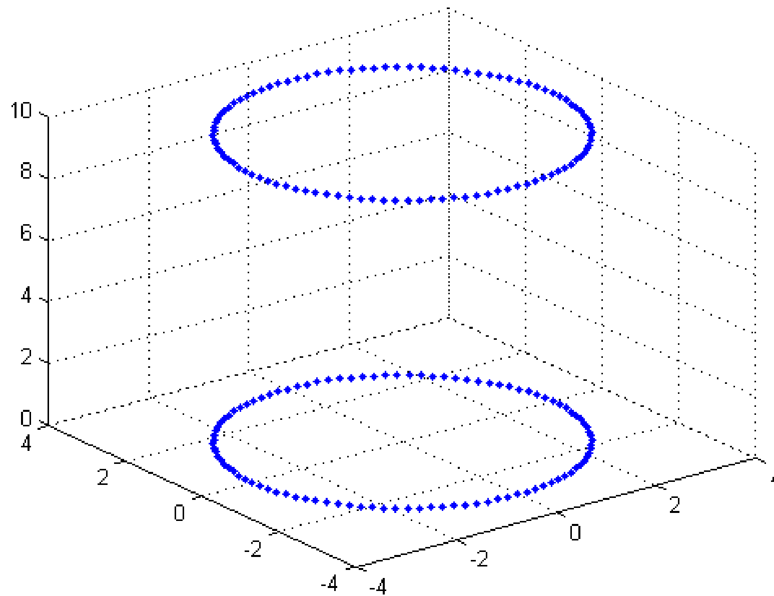

Abbildung 6.2: Zylinder mit 202 Punkten

6.1 Schranken

C2

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	2.2361	0.0313
mebdiam	–	–	2.2361	0.7500
megiddo	0.4995	0.7102	–	0.8594
dreiecke	0.4472	1.7889	3.8730	0.0313
vgl real	0.5	0.5	2	–

C3

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	3	0
mebdiam	–	–	3	0.1250
megiddo	0.9990	1.7397	–	2.25
dreiecke	0.7453	2.9814	5.1962	0.0313
vgl real	1.1180	1.1180	2	–

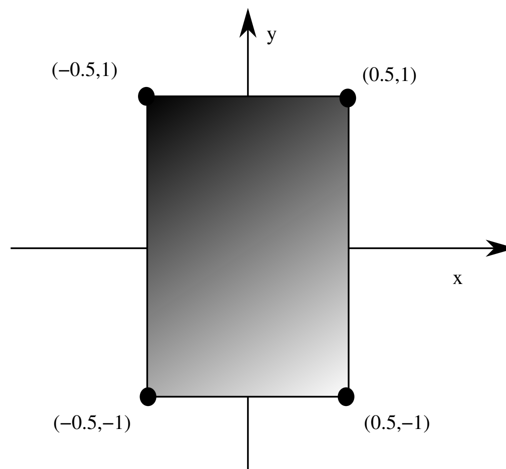

Abbildung 6.3: Gestreckter Einheitswürfel

C4

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	3.6056	0
mebdiam	–	–	3.6056	0.2656
megiddo	0.9990	2.0088	–	44.1875
dreiecke	0.8771	3.5082	6.2450	0.0313
vgl real	1.5	1.5	2	–

S2

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	1	0.0156
mebdiam	–	–	1.1547	0.8281
megiddo	0.3169	0.4520	–	0.9219
dreiecke	0.4330	1.7321	1.7321	0
vgl real	0.4330	0.4330	1	–

S3

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	1	0
mebdiam	–	–	1.2247	0.2031
megiddo	0.3411	0.5954	–	1.5938
dreiecke	0.4330	1.7321	1.7321	0
vgl real	0.5	0.5	0.8660	–

S4

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	1	0
mebdiam	–	–	1.2649	0.2656
megiddo	0.3491	0.7036	–	3.1563
dreiecke	0.4330	1.7321	1.7321	0
vgl real	0.5000	0.5000	0.8660	–

T10

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	11.6619	0
mebdiam	–	–	11.6619	0.2188
megiddo	2.9863	5.2097	–	12.9219
dreiecke	2.5725	10.2899	20.1990	0
vgl real	3	3	10	–

T100

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	11.6619	0.2031
mebdiam	–	–	11.6619	0.2031
megiddo	2.9863	5.2097	–	113.9219
dreiecke	2.5725	10.2899	20.1990	0.0938
vgl real	3	3	10	–

T1000

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	11.6619	281.1719
mebdiam	–	–	11.6619	0.1719
megiddo	2.9863	5.2097	–	2563.3
dreiecke	2.5725	10.2899	20.1990	0.3750
vgl real	3	3	10	–

CF

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	–	$-(error)$
mebdiam	–	–	2.2361	1.5000
megiddo	0.4995	0.7102	–	$1.0033 \cdot 10^5$
dreiecke	0.4472	1.7889	3.8730	15.2344
vgl real	0.5	0.5	2	–

UF

Loeser	*usR*	*osR*	*osD*	*Zeit*
diam	–	–	–	–(*error*)
mebdiam	–	–	1.4142	0.7969
megiddo	0.7102	0.4995	–	98723
dreiecke	0.3536	1.4142	2.4495	3.0938
vgl real	0.5	0.5	1	–

6.2 Zylinder

6.2.1 C2

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5000	3	0.4612	0.0842	0.1719
0.01	0.5000	10	0.4972	0.0056	0.5156
0.001	0.5000	14	0.4998	0.00036251	0.6719
0.0001	0.5000	16	0.5000	0.000091083	0.7656
10^{-8}	0.5000	37	0.5000	$5.5877 \cdot 10^{-9}$	1.6719
10^{-11}	0.5000	58	0.5000	$5.4570 \cdot 10^{-12}$	2.5781

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5000	3	0.5000	$3.6864 \cdot 10^{-11}$	1.1563

mincyl_mg_noSOCP_lokal_noSDP

mit Schranken: dreiecke, diam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5079	153	0.4892	0.0381	13.8594
0.01	0.5010	1281	0.4988	0.0044	115.1719

mincyl_mg_noSOCP_lokal_SDP

mit Schranken: dreiecke, diam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5320	38	0.5000	0.0640	16.6875
0.01	0.5020	606	0.5000	0.0039	264.5625

6.2.2 S2

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.4330	9	0.4195	0.0321	0.4688
0.01	0.4330	15	0.4306	0.0056	0.7969
0.001	0.4330	25	0.4328	$4.1200 \cdot 10^{-4}$	1.3594
0.0001	0.4330	36	0.4330	$2.2829 \cdot 10^{-5}$	1.8750
0.00001	0.4330	40	0.4330	$6.5409 \cdot 10^{-6}$	2.0781
0.000001	0.4330	51	0.4330	$3.5552 \cdot 10^{-7}$	2.5469
0.0000001	0.4330	60	0.4330	$2.0404 \cdot 10^{-8}$	2.8750

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.4330	4	0.4330	$2.4980 \cdot 10^{-13}$	1.4844

mincyl_mg_noSOCP_lokal_noSDP

mit Schranken: dreiecke, diam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.4743	1	0.4330	Inf	0.1406
0.01	0.4369	12	0.4330	0.0954	1.2188
0.001	0.4331	334	0.4330	0.0071	30.3750

mincyl_mg_noSOCP_lokal_SDP

mit Schranken: dreiecke, diam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.4743	2	0.4330	∞	0.8906
0.01	0.4369	17	0.4330	0.0955	6.9844

6.2.3 C3

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.1180	157	1.0688	0.0461	14.5469
0.01	1.1180	809	1.1078	0.0092	65.8125

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.1181	24	1.1180	$2.6376 \cdot 10^{-5}$	12.6875

mincyl_mg_noSOCP_lokal_SDP

mit Schranken: dreiecke, diam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.2133	264	0.1180	0.0852	162.2500

6.2.4 S3

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5005	588	0.4704	0.0639	57.4375
0.01	0.5000	1680	0.4955	0.0091	159.8750

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5029	24	0.5000	0.0058	10.9375

6.2.5 T10

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	59	2.8751	0.0434	6.6094
0.01	3.0000	336	2.9760	0.0081	36.9375
0.001	3.0000	1612	2.9976	0.00080299	165.9219

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	9	3.0000	$1.1696 \cdot 10^{-5}$	7.2344

6.2.6 T100

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	67	2.8751	0.0432	15.1563
0.01	3.0000	331	2.9760	0.0080	74.4375
0.001	3.0000	1580	2.9976	0.00079917	342.7031

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	11	3.0000	$2.6573 \cdot 10^{-6}$	42.0781

6.2.7 T1000

mincyl_mg_noSDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	70	2.8757	0.0432	113.1875
0.01	3.0000	326	2.9763	0.0080	520.8125

mincyl_mg_SDP

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	3.0000	13	3.0000	$5.2947 \cdot 10^{-6}$	499.7656

6.2.8 UF

mincyl_mg_noSDP

mit Schranken: dreiecke

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5000	4	0.4612	0.0842	463.7969
0.01	0.5000	14	0.4972	0.0056	1676.5

mincyl_mg_noSOCP_lokal_noSDP

mit Schranken: dreiecke, mebdiam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5079	88	0.4831	0.0514	71.3906
0.01	0.5010	856	0.4982	0.0056	686.9531

6.2.9 CF

mincyl_mg_noSDP

mit Schranken: dreiecke

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5000	3	0.4612	0.0842	579.1250
0.01	0.5000	8	0.4972	0.0056	1109.1
0.001	0.5000	14	0.4998	0.00036251	2126.7

mincyl_mg_noSOCP_lokal_noSDP

mit Schranken: dreiecke, mebdiam

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.5079	153	0.4892	0.0381	124.00
0.01	0.5010	1282	0.4988	0.0044	1026.6
0.001	0.5001	12100	0.4998	0.00054725	9718.4

6.3 Dicke

6.3.1 C2

minslab

zieleps	*r*	*time*
0.1	1.000	0.1719
0.01	1.000	0.2813
0.001	1.000	0.9844
0.0001	1.000	3.8438

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.000	4	0.9223	0.0842	0.1250
0.01	1.000	8	0.9944	0.0056	0.1719
0.001	1.000	16	0.9996	$3.6251 \cdot 10^{-4}$	0.2969
0.0001	1.000	19	0.9999	$9.1083 \cdot 10^{-5}$	0.3438

6.3.2 S2

minslab

zieleps	*r*	*time*
0.1	0.8660	0.1563
0.01	0.8660	0.3125
0.001	0.8660	1.0156
0.0001	0.8660	3.9531

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.8660	11	0.8391	0.0321	0.2344
0.01	0.8660	15	0.8612	0.0056	0.2656
0.001	0.8660	26	0.8657	$4.1200 \cdot 10^{-4}$	0.4688
0.0001	0.8660	36	0.8660	$2.2829 \cdot 10^{-5}$	0.6250
0.00001	0.8660	42	0.8660	$6.5409 \cdot 10^{-6}$	0.7031
0.000001	0.8660	53	0.8660	$3.5751 \cdot 10^{-7}$	0.9063
0.0000001	0.8660	67	0.8660	$2.2790 \cdot 10^{-8}$	1.0938

6.3.3 C3

minslab

zieleps	*r*	*time*
0.1	1.0000	8.9219
0.01	1.0000	102.4375

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.0000	87	0.9586	0.0432	1.6094
0.01	1.0000	389	0.9921	0.0080	6.2655
0.001	1.0000	1879	0.9992	$7.9916 \cdot 10^{-4}$	30.0313

6.3.4 S3

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.7071	104	0.6578	0.0749	1.8125
0.01	0.7071	208	0.7026	0.0064	3.4688
0.001	0.7071	374	0.7066	$7.2484 \cdot 10^{-4}$	6.0000

6.3.5 T10

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	5.7063	206	5.4072	0.0553	6.5156
0.01	5.7063	907	5.6612	0.0080	29.4063
0.001	5.7063	3552	5.7018	$7.9916 \cdot 10^{-4}$	104.1875

6.3.6 T100

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	5.9970	231	5.6416	0.0630	701.5625

6.3.7 CF

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.01	1.000	10	0.9944	0.0054	2606.7

6.3.8 UF

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	1.000	4	0.9223	0.0842	1198.6
0.01	1.000	8	0.9786	0.0219	1833.1

6.3.9 S4

minslab_mg

zieleps	*r*	*iter*	*lower*	*eps*	*time*
0.1	0.6455	12209	0.5961	0.0828	263.0781

Kapitel 7

Analyse

In diesem Kapitel werden die Tests bewertet und Vorschläge erarbeitet welche Verfahren unter welchen Bedingungen vorzuziehen sind.

Es hat sich herausgestellt, dass Megiddo sich aus zwei Gründen nicht eignet obere und untere Schranken für den Zylinderradius zu berechnen. Zum einen ist die Approximationsgüte mit $\sqrt{d}$ relativ schlecht, zum anderen ist die Laufzeit (auch in der Approximationsvariante) ziemlich groß.

Der Ansatz keine SOCPs zu lösen scheitert in den meisten Fällen, da um eine genaue Approximation zu bekommen (ϵ klein) die Zahl der Löseraufrufe immens ansteigt und dadurch der Vorteil des schnelleren Lösers (meb anstelle von SOCP-Solvern) zunichte gemacht wird. In 2D und bei vielen Punkten ist jedoch der Vorteil groß genug um bis zu einer Fehlertoleranz von einem Prozent schneller zu sein wie die Variante mit SOCPs.

Bei der Lösung mit SDP ergab sich, dass die erste SDP–Formulierung stets der Zweiten vorzuziehen ist. Die zweite liefert zwar öfter ein bessere Schranke, jedoch ist diese Verbesserung jeweils so gering, dass der Vorteil durch bessere Schranken durch die größere Laufzeit durch die SDP–Berechnung zunichte gemacht. Die erste SDP–Formulierung eignet sich jedoch nur bei Körpern mit relativ wenigen Punkten.

7.1 Schranken

7.1.1 2–dimensional

untere Schranke für den Zylinderradius

Anzahl der Punkte	empfohlene Schranke
wenige	**dreiecke**
viele	**dreiecke**

obere Schranke für den Durchmesser

Anzahl der Punkte	empfohlene Schranke
wenige	**diam**
viele	**meb_diam**

7.1.2 3–dimensional

untere Schranke für den Zylinderradius

Anzahl der Punkte	empfohlene Schranke
wenige	**dreiecke**
viele	**dreiecke**

obere Schranke für den Durchmesser

Anzahl der Punkte	empfohlene Schranke
wenige	**diam**
viele	**meb_diam**

7.1.3 $>$3–dimensional

untere Schranke für den Zylinderradius

Anzahl der Punkte	empfohlene Schranke
wenige	**dreiecke**
viele	**dreiecke**

obere Schranke für den Durchmesser

Anzahl der Punkte	empfohlene Schranke
wenige	**diam**
viele	**meb_diam**

7.2 Dicke

Das adaptive Verfahren **minslab_mg** ist in jedem Fall vorzuziehen.

7.3 Zylinderradius

Alle vorgestellten Verfahren zur Berechnung des Zylinderradius sind adaptive Verfahren. Hierfür bezeichne AP den Fall, dass vorab bekannt ist, dass der Koordinatenursprung auf der Zylinderachse liegt. Ist dies nicht der Fall, so

lässt sich P so transformieren, dass wieder AP vorliegt, jedoch verschlechtert sich dabei die Güte–Garantie so stark, dass zumindest bei dieser Transformation es keinen Sinn macht das Verfahren anzuwenden.

7.3.1 2–dimensional

Punkte	empfohlener Löser (AP)	empfohlener Löser
wenige	**mincyl_mg_SDP**	**mincyl_mg**
viele	**mincyl_mg_noSOCP_noSDP**	**mincyl_mg_noSOCP_noSDP**

7.3.2 3–dimensional

Punkte	empfohlener Löser (AP)	empfohlener Löser
wenige	**mincyl_mg_SDP**	**mincyl_mg**
viele	**mincyl_mg_noSDP**	**mincyl_mg_noSDP**

7.3.3 >3–dimensional

Punkte	empfohlener Löser (AP)	empfohlener Löser
wenige	**mincyl_mg_SDP**	**mincyl_mg**
viele	**mincyl_mg_noSDP**	**mincyl_mg_noSDP**

Kapitel 8

Weiteres Verbesserungspotential

An welchen Stellen gibt es für die Vorgestellten Algorithmen Verbesserungspotential? Grundsätzlich gilt für alle Algorithmen, dass bessere Schranken auch eine bessere Laufzeit impliziert.

8.1 Heuristiken

In allen mincyl_* und minslab_* Programmen gibt es einen Parameter, der das Maß der Epsilon–Verkleinerung festlegt. In allen mincyl_mg_noSOCP_* gibt es einen Parameter, der das Maß die Mindest–Theta–Verfeinerung festlegt. In allen mincyl_mg_noSOCP_* gibt es einen Parameter, der die Mindest–Quotient–Vergrößerung in der Fixpunktiteration festlegt.

Für all diese Parameter gibt lassen sich sicherlich, eventuell für spezielle Fälle, bessere Werte finden.

8.2 Verwendete Programme

Der Einsatz anderer, schnellerer Löser bringt selbstverständlich einen Vorteil. In diesem Projekt wurde stets SEDUMI verwendet um SOCPs zu lösen. Anstelle von meb hätte es auch einen anderen schnelleren Solver gegeben, der nicht ausprobiert wurde da er sich, zumindest nicht so einfach, in Matlab einbinden lässt.

Bereits bei der Berechnung der Kugelapproximation entsteht ein Fehler, da von Simplexen ausgehend die Kugelkappen erzeugt werden. Ein besseres Verfahrenskonzept würde die Zahl der Richtungsvektoren verkleinern und damit die Laufzeit aller Programme in diesem Projekt verbessern.

Im Fall des verankerten Zylinders, also wenn SDPs verwendet werden, müsste der Core–Set–Ansatz vielversprechend sein, da die SDPs mit wenigen

Punkten wesentlich performanter sind wie mit vielen Punkten.

8.3 Programmiersprache

Eine relativ kleine Verbesserung ist zu erwarten, wenn man alle Programme in `C` umsetzt und dabei eine bessere Speicherverwaltung nutzt. Allerdings ist fraglich ob der Aufwand den Nutzen rechtfertigt. Für den konkreten Einsatz, zum Beispiel in der Medizin, sollten die Programme jedoch in eine gängige Programmiersprache wie `C` umgesetzt werden, damit der Code besser in vorhandene Systeme eingebettet werden kann.

8.4 Programmiertechnik

Matlab in der jetzigen Version unterstützt nur das Einprozessor–Rechnermodell, demnach ist verteiltes Rechnen nicht möglich. Alle vorgestellten Programme ließen sich jedoch pro Verfeinerungsschritt um einen Faktor gleich der Anzahl der Richtungsvektoren verschnellern, wenn genügend Prozessoren beziehungsweise Prozessorkerne zur Verfügung stehen. Zur Begründung: Für eine feste Kugelapproximation könnten für jeden Richtungsvektor die Berechnungen konfliktfrei parallel ausgeführt werden, da sie keine gemeinsamen Daten verändern. Es bleibt abzuwarten wie genau die Programmierschnittstelle dafür aussehen wird; es wurde jedoch versucht die Programme so zu schreiben, dass sie diesbezüglich sehr einfach abänderbar sind. Auf den ersten Blick mag es heutzutage absurd wirken von so vielen Rechnerkernen zu sprechen, allerdings spricht Intel in seiner Roadmap [7] davon, dass die soeben begonnen Multi–Core–Era in wenigen Jahren, voraussichtlich 2010, von der Many–Core–Era abgelöst wird. In der Many–Core–Era prognostiziert Intel, dass die Prozessoren mit hunderten oder sogar tausenden threads gleichzeitig umgehen können. Dadurch wird klar welches Potential im verteilen Rechnen, für die vorgestellten Programme, steckt.

Literaturverzeichnis

[1] R. Brandenberg. Course: Computational convexity, 2005. www.ma.tum.de/~brandenb.

[2] R. Brandenberg. Persönliche korrespondenz, 2006.

[3] T. M. Chan. Faster core-set construction and data-stream algorithms in fixed dimensions. 2005.

[4] T.M. Chan. Approximating the diameter width smallest enclosing cylinder, and minimum -width annulus. 2002.

[5] C.A. Duncan, M. T. Goodrich, and A. Ramos. Efficient approximation algorithms for computational metrology. *In Proceeding of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[6] M. T. Goodrich and R. Tamassia. *Algorithm Design - Foundations, Analysis and Internet Examples*. Wiley, 2002.

[7] Intel. Platform 2015: Intel processor and platform evolution for the next decade. 2006.

[8] P. Kumar, J.S.B.Mitchell, and A. Yildirim. Minimal enclosing ball, 2005. http://www.compgeom.com/meb/.

[9] N. Megiddo. Finding a line of sight thru boxes in d-spaces in linear time. *IBM Research Division*, 1996.

[10] P.Kumar, J.S.B. Mitchell, and E.A. Yildirim. Computing core-sets and approximate smallest enclosing hypersheres in high dimensions.

[11] S. Rittsteiger. Algorithmen zu containment problemen. Master's thesis, TU München, 2005.