Eine modifizierte Barrieremethode für konvexe quadratische Optimierungsprobleme

Diplomarbeit

von

Sven Oliver Krumke

Betreuer: Prof. Dr. Josef Stoer

Institut für Angewandte Mathematik und Statistik
Bayerische Julius–Maximilians–Universität Würzburg

März 1994

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Vorwort 3

1.2 Grundlegende Definitionen 8

2 Klassische Lösungsverfahren und Abwandlungen **11**

2.1 Logarithmische Barrieremethoden 11

2.2 Multiplikatorverfahren 14

3 Modifizierte Barrierefunktionen **18**

3.1 Motivation und grundlegende Eigenschaften 18

3.2 Modifizierte Barrierefunktionen für quadratische Programme 22

3.3 Hauptergebnisse von Polyak über Modifizierte Barrierefunktionen . . . 30

3.4 Barrierefunktionen mit Shift 45

4 Die Modifizierte Barrieremethode **54**

4.1 Herleitung des Verfahrens 54

4.2 Grundlegende Eigenschaften des Algorithmus 56

5 Die Konvergenz des Verfahrens **62**

5.1 Polyaks Konvergenzresultate 63

5.2 Powells Ergebnisse für lineare Programme 65

5.3 Einige Resultate aus der konvexen Analysis 66

5.4 Konvergenz der dualen Folge 74

5.5 Konvergenzverhalten der primären Folge 84

5.6 Konvergenzrate der dualen Folge für lineare Programme 91

6 Implementierung der Modifizierten Barrieremethode 96

6.1 Vom Modellalgorithmus zum implementierbaren Algorithmus 96

6.1.1 Numerische Minimierung der Modifizierten Barrierefunktionen . 97

6.1.2 Abbruchkriterien 98

6.1.3 Wahl der Startvektoren 100

6.1.4 Wahl der Barriereparameter 102

6.2 Numerische Ergebnisse 107

6.2.1 Klee–Minty Probleme 110

6.2.2 Zuordnungs–Probleme 115

6.2.3 Zufällige lineare Programme 120

6.2.4 Least–Squares Probleme 123

6.2.5 Zufällige quadratische Programme 127

6.2.6 Zusammenfassung der Ergebnisse 131

7 Fazit und Ausblick 134

A Beschreibung des Programmprojektes 137

A.1 Konzept der Implementierung 138

A.2 Benutzung des Programms `modbfm` 142

A.3 Der Problemgenerator `gen` 144

B Quelltexte 146

B.1 Die Matrix– und Vektorklasse: `matrix.h` 146

B.2 Der Parser für die Konfigurationsdatei 198

B.2.1 Lexikalische Analyse: `lexer.lex` 198

B.2.2 Syntaktische Analyse: `parser.y` 201

B.3 Das Hauptmodul: `main.c` 204

C Symbole und Notationen 237

Literaturverzeichnis 239

Kapitel 1

Einleitung

1.1 Vorwort

Die Simplexmethode galt lange Zeit als das effektivste Verfahren zur Lösung von linearen Programmen. Obwohl sie im schlechtesten Fall eine exponentielle Anzahl von Schritten zur Lösung benötigt (siehe etwa [22]), tritt dieser Fall in der Praxis äußerst selten auf, und die Simplexmethode ist in der Regel ein sehr effizientes Lösungsverfahren.

Das Interesse an alternativen Methoden zur Lösung von linearen Programmen wurde durch eine Veröffentlichung von Karmarkar [20] geweckt. Karmarkar benutzte in seinem Verfahren projektive Transformationen und zeigte, daß es für lineare Programme eine polynomiale Komplexität besitzt, die weit besser als jede bis dahin bekannte Schranke war. Im Gegensatz zur Simplexmethode nutzt Karmarkars Verfahren die kombinatorische Natur des zu lösenden linearen Programms nicht direkt aus. Es erzeugt, ausgehend von einem Startpunkt im relativen Inneren der zulässigen Punktmenge, eine Folge, die gegen eine Optimallösung des linearen Programms konvergiert. Befindet man sich nahe genug an der Optimallösung, so kann man diese durch einen einfachen Rundungsprozeß aus dem laufenden Iterationspunkt exakt gewinnen.

Bereits kurz nach der Arbeit von Karmarkar erschienen in verschiedenen Zeitschriften Artikel, die behaupteten, das Verfahren von Karmarkar sei in jedem Falle deutlich schneller als jede existierende Implementierung der Simplexmethode. Obwohl diese Behauptungen niemals vollständig und in aller Strenge verifiziert wurden, lenkte dies doch die Aufmerksamkeit auf Alternativen zur Simplexmethode, insbesondere auf Innere–Punkte–Verfahren.

Wenige Jahre nach der Veröffentlichung von Karmarkar zeigte sich eine Implementierung des dualen affinen Verfahrens ([1]) gegenüber einer speziellen Implementierung der Simplexmethode (MINOS 4.0) für alle NETLIB–Testprobleme überlegen.

Seit dieser Zeit ist die Entwicklung im Bereich der Innere–Punkte–Verfahren sehr schnell vorangegangen, und es wurden zahlreiche Algorithmen entwickelt. Oftmals wurden die diesen Verfahren zugrundeliegenden Ideen auch auf konvexe quadratische Programme ausgedehnt. Gill et. al. stellten in [11] einen Zusammenhang zwischen dem Verfahren von Karmarkar und der klassischen logarithmischen Barrieremethode (vgl. etwa [7],[9]) zur Lösung von nichtlinearen Optimierungsproblemen her. Dies führte zu der Erkenntnis, daß Karmarkars Verfahren nur ein Spezialfall einer weit größeren Klasse von logarithmischen Barrieremethoden ([12], [13]) ist. Damit stieg das Interesse an diesen aus der Mode gekommenen Barrieremethoden.

Problematisch bei vielen Barrieremethoden ist die numerische Stabilität in der Praxis. Um die Konvergenz des Verfahrens zu sichern, ist es erforderlich, daß ein Straf– bzw. Barriereparameter gegen Unendlich divergiert.

Im Gegensatz dazu kann es bei Verfahren, die auf erweiterten Lagrangefunktionen (Augmented Lagrangians) basieren, oftmals unnötig sein, einen Parameter über alle Schranken wachsen zu lassen, da Approximationen für einen (passenden) optimalen Lagrangevektor erzeugt werden.

R. Polyak stellte in [27] *Modifizierte Barrierefunktionen* (Modified Barrierfunctions) vor. Seine Idee ist es, mit diesen Funktionen die Vorzüge der klassischen Barrierefunktionen und der erweiterten Lagrangefunktionen zu verbinden. Aufbauend auf den Modifizierten Barrierefunktionen entwickelte Polyak die *Modifizierte Barrieremethode* für nichtlineare Optimierungsprobleme [27] sowie für lineare Programme [28]. Die Modifizierte Barrieremethode ist kein Innere–Punkt–Verfahren im ursprünglichen Sinne, da durch eine monotone Transformation der Nebenbedingungen die zulässige Menge vergrößert wird und somit während der Iteration auch (für das Ausgangsproblem) unzulässige Punkte erzeugt werden. Ein positiver Nebeneffekt der Transformation der Nebenbedingungen ist, daß die Modifizierte Barrieremethode im Gegensatz zu den herkömmlichen Barrieremethoden keinen Startpunkt im Inneren der Menge der zulässigen Lösungen des Ausgangsproblems benötigt.

In [27] zeigt Polyak die Konvergenz des Verfahrens für konvexe (und nichtkonvexe) Optimierungsprobleme unter der Voraussetzung, daß folgende Bedingungen erfüllt sind: (1) Es existiert eine eindeutige Optimallösung x^*, (2) es gilt strikte Komplementarität, (3) die hinreichenden Bedingungen zweiter Ordnung sind in x^* erfüllt, (4) die Gradienten der in x^* aktiven Restriktionen sind linear unabhängig, (5) die Menge der zulässigen Lösungen des Problems ist beschränkt, (6) der Barriereparameter der Modifizierten Barrierefunktionen wird „genügend groß“ gewählt.

Für lineare Programme setzt der Konvergenzbeweis von Polyak in [28] wiederum die Bedingungen (1), (2) und (4) bis (6) voraus. Die Voraussetzung (3) wird jedoch durch die Forderung nach primär und dual nichtdegenerierten Optimallösungen ersetzt.

M. J. D. Powell zeigt in [29], daß man zumindest für lineare Programme die obigen Voraussetzungen stark abschwächen kann. Nach seinen Ergebnissen reicht bereits die Beschränktheit der zulässigen Punktmenge aus, um bei der Modifizierten Barrieremethode Konvergenz zu erhalten.

Ziel dieser Diplomarbeit ist es, die Modifizierte Barrieremethode sowohl in der Theorie als auch der Praxis zu untersuchen, wenn sie auf konvexe quadratische Programme angewendet wird. Dies umfaßt unter anderem den Spezialfall der linearen Programme.

Im nächsten Abschnitt definieren wir dazu formal das konvexe quadratische Problem (QP), für das wir die Modifizierte Barrieremethode untersuchen werden, und geben einige grundlegende Definitionen.

Im Kapitel 2 wiederholen wir kurz die klassische logarithmische Barrieremethode und ihre Abwandlungen für lineare bzw. konvexe quadratische Programme. Wiederholt wird auch die Methode der erweiterten Lagrangefunktion. Diese hat, wie bereits oben erwähnt, neben der klassischen logarithmische Barrieremethode ihren Teil zu Polyaks Idee der Modifizierten Barrieremethode beigesteuert. Wir stellen auch in aller Kürze das Exponential–Multiplikatorverfahren vor. Dieses Verfahren, welches von Bertsekas vorgeschlagen und von Tseng und Bertsekas in [37] untersucht wurde, ist ebenfalls in gewissen Sinne mit der Modifizierten Barrieremethode verwandt. Daher lassen sich einige Beweisideen für dieses Verfahren in der ein oder anderen Form auf die Modifizierte Barrieremethode übertragen.

Kapitel 3 beschäftigt sich mit den Modifizierten Barrierefunktionen, so wie sie von Polyak in [27] bzw. [28] eingeführt wurden. Es werden ihre grundlegenden Eigenschaften hergeleitet und die Beweise von Polyak aus [27] für den Spezialfall des in Kapitel 1 vorgestellten quadratischen Programms (QP) geführt. Dies umfaßt neben den eigentlichen Modifizierten Barrierefunktionen auch die Barrierefunktionen mit Shift, die sich als ein weiterer Spezialfall ergeben. Ergänzend zu den Ergebnissen von Polyak, finden wir noch einige weitere Eigenschaften der Modifizierten Barrierefunktionen für den Fall, daß die Menge der Optimallösungen von (QP) nichtleer und beschränkt ist.

In Kapitel 4 wird dann die Modifizierte Barrieremethode vorgestellt. Das Verfahren erzeugt, wie auch die Methode der erweiterten Lagrangefunktion, neben einer *primären Folge* auch eine Folge von Lagrangemultiplikator–Schätzungen, die wir als *duale Folge* bezeichnen. Mit Hilfe der Karush–Kuhn–Tucker Bedingungen für (QP) motivieren wir den Multiplikator–Update, den Polyak bei der Modifizierten Barrieremethode benutzt. Wir übertragen einige Resultate von Powell aus [29] über die wesentlichen Eigenschaften des Verfahrens vom linearen auf den quadratischen Fall.

Kapitel 5 ist der Konvergenzanalyse der Modifizierten Barrieremethode gewidmet. Zunächst werden mit Hilfe der Ergebnisse aus Kapitel 2 die Konvergenzresultate von

Polyak gezeigt. Danach zitieren wir die Ergebnisse von Powell aus [29] über die Konvergenz des Verfahrens für lineare Programme.

Der Rest des Kapitels enthält die *neuen* Ergebnisse, die im Rahmen dieser Arbeit gefunden werden konnten. Einige Beweise benutzen die Techniken, die in [37] zur Analyse der Exponential–Multiplikatormethode angewendet werden, sind ansonsten aber eigene Resultate.

Wir zeigen die Konvergenz der dualen Folge gegen eine Optimallösung des (Lagrange–) dualen Problems unter der einzigen Voraussetzung, daß die Menge der Optimallösungen von (QP) nichtleer und beschränkt ist. Die eben genannte Voraussetzung wird benötigt, um überhaupt die Definiertheit des Verfahrens zu sichern. Daher werden alle unsere Resultate unter dieser Grundvoraussetzung bewiesen. Wir finden einige Konvergenzeigenschaften der primären Folge. Bei strikt konvexer Zielfunktion konvergiert die primäre Folge gegen die Optimallösung des quadratischen Programmes (QP). Ist die Zielfunktion nicht strikt konvex, aber die Optimallösung eindeutig und existiert ein strikt komplementärer Lagrangemultiplikator, so ist dies ebenfalls hinreichend für die Konvergenz. Wir zeigen ebenfalls, daß, falls die die Zielfunktion linear ist, oder die Barriereparameter gegen Unendlich divergieren, die Forderung nach einem strikt komplementären Lagrangemultiplikator redundant ist, man also auch dann Konvergenz erhält, wenn nur die Optimallösung eindeutig ist. Im folgenden leiten wir dann noch weitere Eigenschaften der primären Folge her.

Im letzten Abschnitt des Kapitels 5 untersuchen wir einige Konvergenzraten in bezug auf die duale Folge für den Fall, daß die Zielfunktion von (QP) linear ist, d.h. daß sich das quadratische Optimierungsproblem (QP) auf ein lineares Programm reduziert. Diese Analyse ist vor allen Dingen deswegen interessant, da für eine lineare Zielfunktion das (Lagrange–) duale Problem dem klassischen dualen linearen Programm entspricht und unsere Ergebnisse aus Kapitel 5 die Konvergenz der dualen Folge gegen eine Optimallösung dieses Problems unter schwachen Voraussetzungen zeigen.

Die Resultate aus Kapitel 5 stärken die Resultate von Polyak und ergänzen die Untersuchungen von Powell. Wir können viele Voraussetzungen, die Polyak für seine Konvergenzbeweise benötigt, fallen lassen oder zumindest durch schwächere Voraussetzungen ersetzen.

Kapitel 6 beschäftigt sich mit der Implementierung der Modifizierten Barrieremethode. Wir besprechen die wesentlichen Gedanken, die nötig sind, damit man vom Modellalgorithmus der Modifizierten Barrieremethode zu einem Algorithmus gelangt, den man implementieren kann. Dies umfaßt unter anderem die Wahl der Startpunkte und geeignete Abbruchkriterien. Neben zwei von Polyak in [27] und [28] vorgestellten Varianten der Modifizierten Barrieremethode schlagen wir eine weitere Version vor. Diese kann mit einem unzulässigen Startvektor initialisiert werden und benötigt keine inneren

Punkte.

Anschließend stellen wir die numerischen Ergebnisse vor, die durch das Lösen von Testproblemen mit Hilfe einer Implementierung der Modifizierten Barrieremethode (in Form des Programms `modbfm`) gewonnen werden konnten, und diskutieren diese.

Im Kapitel 7 werden Abschlußbetrachtungen angestellt und einige Ausblicke gegeben.

Der Anhang A enthält eine Beschreibung des Konzeptes der Implementierung und Informationen zur Bedienung des Programms `modbfm`. Im Anhang B sind die wichtigsten Quelltexte für das Programm `modbfm` abgedruckt.

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Josef Stoer herzlich bedanken, der das interessante Thema stellte, mich aber weitgehend an denjenigen Aspekten arbeiten ließ, die mich vornehmlich interessierten.

Mein Dank gilt auch Herrn Dr. Florian Jarre, der mir wichtige Literaturstellen aufzeigte und wertvolle Ratschläge gab.

1.2 Grundlegende Definitionen

Wir betrachten in dieser Arbeit das quadratische Optimierungsproblem

$$\begin{array}{lll} \text{(QP)} & \min & \frac{1}{2}x^TQx + c^Tx \\ & x \in \mathbb{R}^n: & Ax \geq b, \end{array}$$

wobei Q eine positiv semidefinite $n \times n$–Matrix und A eine $m \times n$–Matrix mit den Zeilen $a_1^T, \ldots, a_m^T$ ist. Mit

$$S = \{x \in \mathbb{R}^n : Ax \geq b\} = \{x \in \mathbb{R}^n : a_j^Tx - b_j \geq 0, j = 1, \ldots, m\}$$

bezeichnen wir die Menge der zulässigen Lösungen und mit S^* die Menge der Optimallösungen von (QP). Zum Zweck der einfacheren Notation setzen wir

$$q(x) := \frac{1}{2}x^TQx + c^Tx.$$

Sofern die Menge S^* der Optimallösungen von (QP) nichtleer ist — dies werden wir bei der Analyse der Modifizierten Barrierefunktionen und –methode stets voraussetzen — verwenden wir q^* für den optimalen Funktionswert, d.h.

$$q^* := \inf\{q(x) : x \in S\} = \inf\{q(x) : Ax \geq b\}.$$

Die Lagrangefunktion zu (QP) ist durch

$$L(x, \lambda) = \begin{cases} \frac{1}{2}x^TQx + c^Tx + \sum\limits_{j=1}^{m} \lambda_j(b_j - a_j^Tx) & \text{, falls } \lambda \geq 0 \\ -\infty & \text{, sonst} \end{cases} \tag{1.1}$$

gegeben. Wegen der positiven Semidefinitheit von Q (und der Linearität der Nebenbedingungen) ist (QP) ein konvexes Optimierungsproblem. Nach dem Satz von Kuhn und Tucker (siehe etwa [36, Theorem 6.7.8]) ist ein Vektor $x^* \in \mathbb{R}^n$ daher genau dann Optimallösung von (QP), wenn es ein $\lambda^* \in \mathbb{R}^m$, $\lambda^* \geq 0$ gibt, so daß folgende Bedingungen erfüllt sind:

$$Qx^* + c - A^T\lambda^* = 0 \tag{1.2a}$$

$$b - Ax^* \leq 0 \tag{1.2b}$$

$$\lambda_j^*(b_j - a_j^Tx^*) = 0 \quad \text{für } j = 1, \ldots, m. \tag{1.2c}$$

Die Gleichung (1.2a) ist gleichbedeutend mit $\nabla_x L(x^*, \lambda^*) = 0$.

Man kann zu (QP) ein duales quadratisches Programm assoziieren. Es lautet:

$$\begin{array}{lll} \text{(QPD)} & \max & -\frac{1}{2}x^TQx + b^T\lambda \\ & x, \lambda: & A^T\lambda - Qx = c \\ & & \lambda \geq 0 \end{array}$$

Wir wollen an dieser Stelle ein wichtiges Dualitätsresultat für das obige Paar von quadratischen Programmen nennen. Sei

$$\eta(x,\lambda) := -\frac{1}{2}x^TQx + b^T\lambda.$$

Es gilt dann der folgende Satz:

Satz 1.2.1 (Dualitätssatz von Dorn)

(i) *Für beliebige zulässige Lösungen x von* (QP) *und (x',λ') von* (QPD) *gilt:* $q(x) \;\geq\; \eta(x',\lambda')$.

(ii) *Wenn* (QP) *eine Optimallösung x^* besitzt, dann existiert ein λ^*, so daß (x^*,λ^*) Optimallösung von* (QPD) *ist. Es gilt: $q(x^*) = \eta(x^*,\lambda^*)$.*

(iii) *Besitzt* (QPD) *eine Optimallösung (x^*,λ^*), so besitzt* (QP) *eine Optimallösung $\hat{x}$ und es gilt: $q(\hat{x}) = \eta(x^*,\lambda^*)$.*

(iv) *Sei x zulässig für* (QP) *und (x',λ') zulässig für* (QPD)*. Gilt $q(x) = \eta(x',\lambda')$, so ist x optimal für* (QP) *und (x',λ') optimal für* (QPD)*. Darüberhinaus gilt $q(x) = \eta(x',\lambda')$ genau dann, wenn*

$$(\lambda')^TAx = b^T\lambda' = c^Tx + (x')^TQx \qquad \text{und} \qquad Qx = Qx'$$

gelten.

Beweis: [36, Theorem 5.9.11]. □

Wir wollen nun im folgenden noch kurz auf einige grundlegende Notationen eingehen, die wir in dieser Arbeit verwenden. Alle wichtigen Notationen und Symbole sind darüberhinaus im Anhang C samt einer kurzen Erläuterung aufgeführt.

Für einen Vektor bezeichnen wir mit dem entsprechenden Großbuchstaben die Diagonalmatrix mit den Einträgen des Vektors auf der Diagonalen: ist z.B. $x \in \mathbb{R}^n$ ein Vektor, so sei $X = \mathrm{diag}(x_1,\ldots,x_n)$. Alle Ungleichungen zwischen Vektoren oder Vektoren und Skalaren sind komponentenweise zu verstehen. Die Mengen $\mathbb{R}^m_+$ und $\mathbb{R}^m_{++}$ sind durch

$$\begin{aligned}\mathbb{R}^m_+ &:= \{y \in \mathbb{R}^n : y \geq 0\}\\ \mathbb{R}^m_{++} &:= \{y \in \mathbb{R}^n : y > 0\}\end{aligned}$$

definiert. Mit e bezeichnen wir den Vektor aus lauter Einsen, $e = (1,\ldots,1)^T$, mit $I^{r,r}$ die $r\times r$ Einheitsmatrix. Wir definieren für eine Matrix B die Grenzennorm (auch: natürliche Norm, lub–Norm) $\mathrm{lub}_\infty(B)$ zu der Vektornorm $\|.\|_\infty$ durch

$$\mathrm{lub}_\infty(B) := \sup_{x\neq 0}\frac{\|Bx\|_\infty}{\|x\|_\infty}.$$

Für die natürliche Matrixnorm lub_∞ gilt dann

$$\mathrm{lub}_\infty(B) = \max_i \sum_{j=1}^{n} |b_{ij}|.$$

Einen Beweis der letzten Gleichung findet man unter anderem in [35, Abschnitt 4.4] oder [34, Abschnitt 1.2.1]. Letztendlich steht log in dieser Arbeit für den natürlichen Logarithmus.

Kapitel 2

Klassische Lösungsverfahren und Abwandlungen

Dieses Kapitel stellt kurz einige Verfahren vor, die in engem Zusammenhang mit der Modifizierten Barrieremethode stehen, welche in Kapitel 4 eingeführt werden wird. Sowohl die klassische logarithmische Barrieremethode aus Abschnitt 2.1 als auch die Methode der erweiterten Lagrangefunktion aus Abschnitt 2.2 sind quasi als „Vorfahren“ der Modifizierten Barrieremethode zu sehen.

2.1 Logarithmische Barrieremethoden

In diesem Abschnitt soll auf die klassische logarithmische Barrieremethode und auf ihre Abwandlungen für lineare und quadratische Programme eingegangen werden.

Die Ursprünge der logarithmischen Barrieremethoden für lineare und quadratische Programme stammen aus der Theorie der nichtlinearen Optimierungsprobleme. Die klassische logarithmische Barrieremethode wurde von Frisch [9] für nichtlineare Optimierungsprobleme vorgeschlagen, und unter anderem von Fiacco und Cormick in [7] untersucht. Für das ungleichungsrestringierte Optimierungsproblem

$$\begin{array}{lll} \text{(P)} & \min & f(x) \\ & x \in S = & \{x \in \mathbb{R}^n : g_j(x) \le 0, j = 1, \ldots, m\}, \end{array}$$

betrachtet man zum Parameter $k > 0$ die Frisch–Funktion

$$F(x,k) = \begin{cases} f(x) - \frac{1}{k} \sum\limits_{j=1}^{m} \log(-g_j(x)) & \text{falls } x \in \operatorname{int} S \\ +\infty & \text{sonst} \end{cases} \tag{2.1}$$

Wir setzen hier f und g_j $(j = 1, \ldots, m)$ als zweimal stetig differenzierbar voraus. Falls f und die g_j $(j = 1, \ldots, m)$ zusätzlich auch konvex sind sowie die Menge S der zulässigen

Lösungen von (P) beschränkt mit nichtleerem Inneren ist, so ist die Funktion $F(.,k)$ strikt konvex auf $\operatorname{int} S$ und besitzt ein eindeutig bestimmtes Minimum $x(k) \in \operatorname{int} S$. Es gilt $\nabla F(x(k),k) = 0$. Die Menge $\{x(k) : k > 0\}$ nennt man dann den Lösungspfad (Trajectory) für das Problem (P). Das ursprüngliche Problem (P) wird nun in eine Folge von unrestringierten Problemen

$$(\mathrm{P}_{\mathbf{k}^\nu}) \quad \min F(x,k^\nu) \equiv \min f(x) - \frac{1}{k^\nu}\sum_{j=1}^{m} \log(-g_j(x))$$

umgewandelt, wobei $(k^\nu)_\nu$ eine strikt monoton steigende Folge von positiven Zahlen mit $\lim\limits_{\nu\to\infty} k^\nu = +\infty$ ist. Die klassische Barrieremethode zur Lösung von (P) läßt sich dann wie folgt formulieren:

Algorithmus 2.1.1 (Klassische Barrieremethode)

(1) *Wähle eine Folge* $(k^\nu)_\nu$ *mit* $0 < k^0 < k^1 < \cdots$ *und* $\lim\limits_{\nu\to\infty} k^\nu = +\infty$.

(2) *Wähle* x^0 *mit* $g_j(x^0) < 0$ *für* $j = 1,\ldots,m$.

(3) *Setze* $\nu := 1$.

(4) *Bestimme* $x^\nu = \arg\ \min\{F(x,k^\nu) : x \in \mathbb{R}^n\}$.

(5) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (4).

Fiacco und McCormick zeigen in [7, Kapitel 3], daß die Folge $(x^\nu)_\nu$, welche vom Algorithmus 2.1.1 geliefert wird, unter Standardvoraussetzungen an die Regularität der Lösungen von (P) gegen eine Optimallösung x^* konvergiert.

Natürlich muß bei der numerischen Realisierung der klassischen Barrieremethode der Schritt (4) durch eine endliche Prozedur ersetzt werden. Dies geschieht, indem man das Newton–Verfahren (eventuell mit Line–Search) zur Lösung des nichtlinearen Gleichungssystems

$$\nabla_x F(x,k^\nu) = 0 \tag{2.2}$$

benutzt. Als Startpunkt der Newton–Iteration wählt man $x^{\nu,0} = x^{\nu-1}$ und erzeugt dann rekursiv eine Folge $(x^{\nu,j})_j$ durch $x^{\nu,j+1} = x^{\nu,j} + \zeta(x^{\nu,j},k^\nu)$, wobei $\zeta(x^{\nu,j},k^\nu)$ der Newton–Schritt zur Lösung des nichtlinearen Gleichungssystems (2.2) im Punkt $x^{\nu,j}$ ist. Der Newton–Schritt $\zeta = \zeta(x^{\nu,j},k^\nu)$ ergibt sich als Lösung des linearen Gleichungssystems

$$\nabla_{xx} F(x^{\nu,j},k^\nu)\zeta = -\nabla_x F(x^{\nu,j},k^\nu). \tag{2.3}$$

Eine spezielle Eigenschaft der Frisch–Funktion, wie auch anderer klassischer Barrierefunktionen, ist ihre Unbeschränktheit in einer Umgebung des Randes ∂S der Menge

S der zulässigen Lösungen von (P). Dies ermöglicht es uns, wie oben angedeutet, den Lösungsprozeß in einem inneren Punkt $x^0 \in \operatorname{int} S$ zu starten und dabei im Inneren von S zu bleiben, ohne besonders auf die Nebenbedingungen zu achten. Es können dazu die geläufigen Techniken für unrestringierte Probleme (evtl. mit geringen Modifikationen[1]) verwendet werden. Dies ist eine besonders wünschenswerte Eigenschaft und stellt einen großen Vorteil der klassischen Barrierefunktionen dar.

Die eben genannten Vorzüge werden aber dann zum Nachteil, wenn sich der Lösungsprozeß dem Rand ∂S nähert. Im allgemeinen ist bei einer Optimallösung x^* mindestens eine Nebenbedingung aktiv. Nun existiert dann weder die Frisch–Funktion $F(.,k)$ noch ihre Ableitung $\nabla_x F(.,k)$ in der Optimallösung x^* des Problems (P). Für x nahe bei x^* und/oder große k^ν ist das Gleichungssystem (2.3) sehr schlecht konditioniert (vgl. etwa [23] oder [38]). Dies bedingt ernsthafte numerische Schwierigkeiten in der Endphase des Algorithmus und verringert dann in der Praxis die Konvergenzgeschwindigkeit.

Die (QP)– bzw. (LP)–Barrieremethode (siehe etwa [30]) greift die Ideen der klassischen logarithmischen Barrieremethode auf. Diese beiden Verfahren beschäftigen sich mit der Lösung des konvexen quadratischen Programms

$$\begin{array}{lll} \text{(QP)} & \min & \frac{1}{2}x^T Q x + c^T x \\ & x \in \mathbb{R}^n : & Ax \geq b, \end{array}$$

aus Abschnitt 1.2, bzw. des linearen Programms

$$\begin{array}{lll} \text{(LP)} & \min & c^T x \\ & x \in \mathbb{R}^n : & Ax \geq b, \end{array}$$

welches ein Spezialfall von (QP) ($Q = 0$) ist. Auch sie verwenden die Frisch–Funktion, die im Fall von (QP) die Form

$$F(x,k) = \frac{1}{2}x^T Q x + c^T x - \frac{1}{k}\sum_{j=1}^{m} \log(a_j^T x - b_j)$$

besitzt. Die beiden Verfahren folgen approximativ dem Lösungspfad $\{x(k) : k > 0\}$, der bereits in den klassischen Algorithmus 2.1.1 einging. Allerdings besteht ein wesentlicher Unterschied zwischen der (QP)– bzw. (LP)–Barrieremethode und der klassischen logarithmischen Barrieremethode: Die (QP)– und (LP)–Barrieremethode verwenden jeweils nur einen Newton–Schritt pro Barriereparameter. Im folgenden ist die (QP)–Barrieremethode algorithmisch formuliert:

[1] wie etwa einer Line–Search

Algorithmus 2.1.2 ((QP)–Barrieremethode)

(1) *Wähle* $x^0 \in \{x : Ax > b\}$.

(2) *Wähle* $k^0 > 0$ *und definiere die Folge* $(k^\nu)_\nu$ *durch*

$$k^{\nu+1} = \frac{k^\nu}{1 - 1/(41\sqrt{m})}, \quad \nu = 0, 1, 2, \ldots$$

(Dann gilt $\lim\limits_{\nu\to\infty} k^\nu = +\infty$*).*

(3) *Setze* $\nu := 1$.

(4) *Setze* $x^{\nu+1} := x^\nu - \nabla^2_{xx} F(x^\nu, k^\nu)^{-1} \nabla_x F(x^\nu, k^\nu)^{-1}$

(5) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (4).

Falls der Funktionswert $q(x^\nu)$ der laufenden Iterierten genügend nahe am optimalen Funktionswert q^* ist, so kann man aus x^ν durch einen Rundungsprozeß eine Optimallösung erhalten. Dabei bedeutet „genügend nahe“ eine Abschätzung der Form

$$q(x^\nu) - q^* \leq 2^{-O(L)}. \tag{2.4}$$

L ist hier die Anzahl der Bits, die nötig sind, um die Problemdaten von (QP) darzustellen. Renegar und Shub zeigen in [30], daß der Algorithmus 2.1.2 mit einem Abbruchkriterium der Form (2.4) in $O(\sqrt{m} \cdot L)$ Schritten terminiert. Die Komplexität der (QP)–Barrieremethode bzw. der (LP)–Barrieremethode macht dieses Verfahren äußerst interessant. Da aber die Barriereparameter sehr groß werden müssen, bevor man das Verfahren abbrechen kann, treten in der Praxis ähnliche numerische Probleme auf wie bei der klassischen logarithmischen Barrieremethode.

2.2 Multiplikatorverfahren

Wir betrachten wieder das Optimierungsproblem

$$\begin{array}{lll} \text{(P)} & \min & f(x) \\ & x \in S = & \{x \in \mathbb{R}^n : g_j(x) \leq 0, j = 1, \ldots, m\}, \end{array}$$

wobei wir wieder f und die g_j ($j = 1, \ldots, m$) als zweimal stetig differenzierbare Funktionen voraussetzen. Dieser Abschnitt befaßt sich mit zwei sogenannten *Multiplikatorverfahren* zur Lösung des Optimierungsproblems (P): der Methode der erweiterten Lagrangefunktion und dem Exponential–Multiplikatorverfahren. Multiplikatorverfahren erzeugen in der Iteration neben einer Folge von Vektoren, die gegen eine Optimallösung

x^* konvergieren soll, auch eine Folge von Approximationen an einen zu x^* passenden optimalen Lagrangemultiplikator. Dadurch wird es oftmals unnötig, einen Straf– bzw. Barriereparameter gegen Unendlich divergieren zu lassen.

Buys und Rockafellar [32] führten die folgende erweiterte Lagrangefunktion (Augmented Lagrangian) zum Problem (P) ein:

$$\begin{aligned}\Phi(x,\lambda,k) &= f(x)+\frac{k}{2}\sum_{j=1}^{m}\left(\max\left\{0,g_j(x)+\frac{\lambda_j}{k}\right\}\right)^2-\frac{k}{2}\sum_{j=1}^{m}\lambda_j^2\\ &= f(x)+\sum_{j=1}^{m}\begin{cases}\lambda_j g_j(x)+\frac{k}{2}g_j^2(x) & \text{, falls } g_j(x)\geq -\frac{\lambda_j}{k}\\ -\frac{k}{2}\lambda_j^2 & \text{, falls } g_j(x)< -\frac{\lambda_j}{k}\end{cases}\end{aligned} \tag{2.5}$$

Die Funktion (2.5) und ihr Gradient $\nabla_x\Phi(.,\lambda,k)$ sind als Funktionen von x stetig. Allerdings existieren Unstetigkeiten in der zweiten Ableitung an solchen Punkten x, für die $g_j(x)=-\frac{\lambda_j}{k}$ für ein $j\in\{1,\ldots,m\}$ gilt.

Es seien f und die g_j, $j=1,\ldots,m$ konvex. Sei x^* eine Optimallösung von (P) und $\lambda^*\geq 0$ ein zugehöriger Lagrangemultiplikator. Unter bestimmten Regularitätbedingungen (im wesentlichen der Eindeutigkeit von x^* und λ^*) kann man zeigen, daß für genügend großen Strafparameter $k>0$ die Hessematrix $\nabla^2_{xx}\Phi(x^*,\lambda^*,k)$ positiv definit, d.h. $\Phi(.,\lambda^*,k)$ in einer Umgebung von x^* strikt konvex, ist. Wegen $\nabla_x\Phi(x^*,\lambda^*,k)=0$ ist x^* dann ein striktes Minimum von $\Phi(.,\lambda^*,k)$.

Wiederum wird das ursprüngliche Problem (P) in eine Sequenz von unrestringierten Problemen umgewandelt. Die Methode der erweiterten Lagrangefunktion (Augmented Lagrangian Method) formuliert sich dann wie folgt:

Algorithmus 2.2.1 (Methode der erweiterten Lagrangefunktion)

(1) *Wähle* $k>0$.

(2) *Wähle* $\lambda^0\in\mathbb{R}^m$ *mit* $\lambda^0\geq 0$.

(3) *Setze* $\nu:=0$.

(4) *Zu gegebenem* λ^ν *bestimme* $x^\nu\in\arg\ \min\{\Phi(x,\lambda^\nu,k):x\in\mathbb{R}^n\}$

(5) *Setze* $\lambda_j^{\nu+1}=\lambda_j^\nu-\min\{\lambda_j^\nu,-kg_j(x)\}$ *für* $j=1,\ldots,m$.

(6) *Setze* $\nu:=\nu+1$ *und gehe zu Schritt* (4).

Die Update–Formel in Schritt (5) geht auf Powell zurück. Unter Standardvoraussetzungen konvergieren die Folgen $(x^\nu)_\nu$, $(\lambda^\nu)_\nu$, die vom Verfahren 2.2.1 geliefert werden, mindestens linear gegen x^* bzw. λ^*, sofern der Strafparameter $k>0$ genügend groß und λ^0 genügend nahe bei λ^* gewählt werden. Weiterhin kann man durch ausreichend große Wahl von k erreichen, daß die lineare Konvergenz beliebig schnell wird.

Ein Problem, das bei der Methode der erweiterten Lagrangefunktion auftaucht, wurde oben bereits angedeutet: Die Funktionen $\Phi(., \lambda^\nu, k)$ besitzen Unstetigkeiten in der zweiten Ableitung. Dies führt zu Problemen, wenn eine Variante des Newton–Verfahrens zur (approximativen) Lösung der unrestringierten Probleme in Schritt (4) verwendet wird.

Bertsekas entwickelte ein Multiplikatorverfahren, welches auf der Funktion

$$\mathcal{E}(x, \lambda, k) = f(x) + \frac{1}{k} \sum_{j=1}^{m} \lambda_j \left(e^{k \cdot g_j(x)} - 1 \right) \tag{2.6}$$

basiert. Die Funktion (2.6) ist als Funktion von x bei festem $\lambda \in \mathbb{R}^m_+$ und $k > 0$ im Gegensatz zu den klassischen erweiterten Lagrangefunktionen zweimal stetig differenzierbar. Der folgende Algorithmus stellt das Exponential–Multiplikatorverfahren (Exponential Multiplier Method) dar:

Algorithmus 2.2.2 (Exponential–Multiplikatorverfahren)

(1) *Wähle* $\kappa > 0$ *und eine Folge* $(k^\nu)_\nu$ *mit* $0 < \kappa \leq k^\nu$ *für* $\nu = 0, 1, 2, \ldots,$

(2) *Wähle* $\lambda^0 \in \mathbb{R}^m$ *mit* $\lambda^0 > 0$.

(3) *Setze* $\nu := 0$.

(4) *Zu gegebenem* λ^ν *und* k^ν *bestimme* $x^\nu \in \arg\ \min\{\mathcal{E}(x, \lambda^\nu, k^\nu) : x \in \mathbb{R}^n\}$

(5) *Setze* $\lambda_j^{\nu+1} = \lambda_j^\nu \cdot e^{k^\nu g_j(x^\nu)}$ *für* $j = 1, \ldots, m$.

(6) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (4).

Seien wieder f und die g_j ($j = 1, \ldots, m$) konvexe, zweimal stetig differenzierbare Funktionen. Bertsekas zeigt in [37] Konvergenzeigenschaften des Verfahrens unter den Voraussetzungen, daß die Menge der Optimallösungen von (P) beschränkt ist und ein Punkt $\bar{x}$ mit $g_j(\bar{x}) < 0$ für alle nicht affinen g_j existiert. Er beweist, daß die Folge $(x^\nu)_\nu$, die vom Algorithmus 2.2.2 geliefert wird, in einem ergodischen Sinne gegen eine Optimallösung x^* konvergiert. Ergodische Konvergenz bedeutet in diesem Zusammenhang, daß jeder Häufungspunkt der Folge $(y^\nu)_\nu$ mit

$$y^\nu = \frac{k^0 x^0 + \cdots + k^\nu x^\nu}{k^0 + \cdots + k^\nu}$$

eine Optimallösung von (P) ist. Weiterhin zeigt Bertsekas, daß die Folge $(\lambda^\nu)_\nu$ gegen die Optimallösung eines zu (P) dualen Problems konvergiert. Für lineare Programme analysiert er in [37] auch einige Konvergenraten des Verfahrens.

Oben wurde bereits ein Vorteil des Exponential–Multiplikatorverfahrens gegenüber der klassischen Methode der erweiterten Lagrangefunktionen erwähnt: Die Funktion $\mathcal{E}$ aus (2.6) ist als Funktion von x zweimal stetig differenzierbar. Als Folge davon ist das Newton–Verfahren zur Lösung der unrestringierten Teilprobleme in der Regel effektiver einsetzbar.

Kapitel 3

Modifizierte Barrierefunktionen

In diesem Kapitel werden die Modifizierten Barrierefunktionen zunächst allgemein für das konvexe Programm

$$\begin{array}{lll} (\mathrm{P}) & \min & f(x) \\ & x \in \mathbb{R}^n : & g_j(x) \leq 0, \quad j = 1, \ldots, m \end{array}$$

(f, g_j konvex, zweimal stetig differenzierbar, $j = 1, \ldots, m$) eingeführt und ihre grundlegenden Eigenschaften und Unterschiede zu den klassischen Barrierefunktionen vorgestellt. Es stellt sich dabei heraus, daß die Modifizierten Barrierefunktionen als klassische Lagrangefunktionen für ein zu (P) äquivalentes Problem betrachtet werden können.

Danach wird die Modifizierte Barrierefunktion zum quadratischen Optimierungsproblem (QP) angegeben und es werden weitere wichtige Eigenschaften für diesen Spezialfall erarbeitet.

3.1 Motivation und grundlegende Eigenschaften

Wir bezeichnen mit $S = \{x \in \mathbb{R}^n : g_j(x) \leq 0, j = 1, \ldots, m\}$ die Menge der zulässigen Lösungen für (P). Sei $k > 0$ beliebig. Da für $k > 0$ die Beziehungen

$$g_j(x) \leq 0 \Leftrightarrow 1 - kg_j(x) \geq 1 \Leftrightarrow \log(1 - kg_j(x)) \geq 0 \Leftrightarrow k^{-1} \log(1 - kg_j(x)) \geq 0$$

gelten, folgt, daß wir (P) äquivalent auch folgendermaßen schreiben können:

$$\begin{array}{lll} (\mathrm{P}^{(\mathbf{k})}) & \min & f(x) \\ & x \in \mathbb{R}^n : & -k^{-1} \log(1 - kg_j(x)) \leq 0, \quad j = 1, \ldots, m. \end{array}$$

Die klassische Lagrangefunktion L_k zum Problem ($\mathrm{P}^{(\mathrm{k})}$) lautet dann

$$L_k(x,\lambda) = \begin{cases} f(x) - k^{-1} \sum_{j=1}^{m} \lambda_j \log(1 - kg_j(x)) & \text{, falls } \lambda \geq 0 \\ -\infty & \text{, sonst} \end{cases}$$

$$= \begin{cases} f(x) - k^{-1} \sum_{j=1}^{m} \lambda_j \log(1 - kg_j(x)) & \text{, falls } x \in \operatorname{int} S_k \text{ und } \lambda \geq 0 \\ +\infty & \text{, falls } x \notin \operatorname{int} S_k \text{ und } \lambda \geq 0 \\ -\infty & \text{, sonst.} \end{cases}$$

Dabei ist

$$S_k := \{x \in \mathbb{R}^n : 1 - kg_j(x) \geq 0, j = 1,\dots,m\} = \{x \in \mathbb{R}^n : g_j(x) \leq \frac{1}{k}, j = 1,\dots,m\}.$$

Dies zeigt, daß die Modifizierte Frisch–Funktion $\mathcal{F}(x,\lambda,k) : \mathbb{R}^n \times \mathbb{R}^m_+ \to (-\infty, +\infty]$, die durch

$$\mathcal{F}(x,\lambda,k) = \begin{cases} f(x) - \dfrac{1}{k} \sum_{j=1}^{m} \lambda_j \log(1 - kg_j(x)) & \text{, falls } x \in \operatorname{int} S_k \\ +\infty & \text{, sonst} \end{cases} \tag{3.1}$$

definiert ist, als klassische Lagrangefunktion für ($\mathrm{P}^{(\mathrm{k})}$) gedeutet werden kann. Man beachte dabei, daß die logarithmischen Strafterme $\log(1 - kg_j(x))$ $(j = 1,\dots,m)$ bereits für $x \in \operatorname{int} S_k$ definiert und endlich sind. Die Idee der Modifizierten Barrierefunktionen beruht also darauf, den zulässigen Bereich des Problems (P) durch eine monotone Transformation der Nebenbedingungen zu vergrößern und dann die gewöhnliche Lagrangefunktion zum äquivalenten Problem zu assoziieren.

Die Vergrößerung des zulässigen Bereichs durch die Abschwächung der Nebenbedingungen, wie dies hier bei der Menge S_k vollzogen wurde, ist eine seit längerem bekannte Technik. Sie wurde bereits bei den *äußeren Straffunktionen* (siehe z.B. [7, Kapitel 4]) angewandt.

Wir werden nun einige wichtige Eigenschaften der Modifizierten Barrierefunktionen herleiten. Wir nehmen dazu an, daß (P) eine Optimallösung x^* mit zugehörigem Lagrangemultiplikator $\lambda^* \geq 0$ besitzt, d.h.

$$\nabla f(x^*) + \sum_{j=1}^{m} \lambda_j^* \nabla g_j(x^*) = 0 \tag{3.2a}$$

$$g_j(x^*) \leq 0 \text{ für } j = 1,\dots,m \tag{3.2b}$$

$$\lambda_j^* g_j(x^*) = 0 \text{ für } j = 1,\dots,m. \tag{3.2c}$$

Wir behaupten, daß wegen der Komplementarität (3.2c) auch

$$\lambda_j^* \log(1 - kg_j(x^*)) = 0 \text{ für } j = 1,\dots,m$$

folgt. In der Tat: Ist nämlich $\lambda_j^* = 0$, so ist die Aussage klar, da der logarithmische Term wegen der Zulässigkeit von x^* stets endlich ist. Ist $\lambda_j^* > 0$, so hat man wegen der Komplementaritätsbedingung (3.2c) dann $g_j(x^*) = 0$, also $1 - kg_j(x^*) = 1$, und daher $\log(1 - kg_j(x^*)) = 0$. Folglich gilt für alle $k > 0$, daß

$$\mathcal{F}(x^*, \lambda^*, k) = f(x^*) - \frac{1}{k}\sum_{j=1}^{m} \lambda_j^* \log(1 - kg_j(x^*)) = f(x^*). \tag{3.3}$$

Betrachten wir nun den Gradienten der Modifizierten Frisch–Funktion $\mathcal{F}(., \lambda^*, k)$ an der Stelle x^*. Zunächst gilt allgemein für alle $x \in \operatorname{int} S_k$ und $\lambda \geq 0$:

$$\nabla_x \mathcal{F}(x, \lambda, k) = \nabla_x f(x) + \sum_{j=1}^{m} \frac{\lambda_j}{1 - kg_j(x)} \nabla_x g_j(x).$$

Für den Fall, daß $x = x^*$ und $\lambda = \lambda^*$, gilt dann wegen der Komplementarität (3.2c), daß $\frac{\lambda_j^*}{1-kg_j(x^*)} = \lambda_j^*$ für $j = 1, \ldots, m$ ist. Daher ergibt sich aus der letzten Rechnung die Identität

$$\nabla_x \mathcal{F}(x^*, \lambda^*, k) = \nabla_x f(x^*) + \sum_{j=1}^{m} \lambda_j^* \nabla_x g_j(x^*) = 0. \tag{3.4}$$

Das letzte Gleichheitszeichen ist dabei eine Konsequenz aus (3.2a). Zuletzt berechnen wir noch die Hessematrix von $\mathcal{F}$. Es gilt für $x \in \operatorname{int} S_k$, $\lambda \geq 0$ und $k > 0$:

$$\nabla_{xx}^2 \mathcal{F}(x, \lambda, k) = \nabla_{xx}^2 f(x) + k \sum_{j=1}^{m} \lambda_j \frac{\nabla_x g_j(x) \nabla_x g_j(x)^T}{(1 - kg_j(x))^2} + \sum_{j=1}^{m} \frac{\lambda_j}{1 - kg_j(x)} \nabla_{xx}^2 g_j(x)$$

Aus dieser Darstellung ersehen wir, daß für festes $x \in \operatorname{int} S_k$ und $\lambda \geq 0$ die Hessematrix $\nabla_{xx}^2 \mathcal{F}(x, \lambda, k)$ positiv semidefinit ist, da wir f und die g_j als konvexe Funktionen vorausgesetzt hatten. Insbesondere ist dann $\mathcal{F}(., \lambda^*, k)$ als Funktion von x konvex auf $\operatorname{int} S_k$. Die Konvexität von $\mathcal{F}(., \lambda^*, k)$ impliziert dann zusammen mit (3.4), daß die Funktion $\mathcal{F}(., \lambda^*, k)$ ihr globales Minimum in x^* annimmt und dort den Funktionswert $f(x^*)$ besitzt, d.h.

$$f(x^*) = \mathcal{F}(x^*, \lambda^*, k) = \min_{x \in \operatorname{int} S_k} \mathcal{F}(x, \lambda^*, k) = \min_{x \in \mathbb{R}^n} \mathcal{F}(x, \lambda^*, k),$$

und somit

$$x^* \in \arg \min_{x \in \operatorname{int} S_k} \mathcal{F}(x, \lambda^*, k) = \arg \min_{x \in \mathbb{R}^n} \mathcal{F}(x, \lambda^*, k).$$

Nehmen wir nun für einen Moment an, daß für das Karush–Kuhn–Tucker Paar x^*, λ^* strikte Komplementarität gilt. Die Nebenbedingungen seien dabei bereits so umnumeriert, daß

$$\begin{aligned} g_j(x^*) = 0, \quad \lambda_j^* > 0, \quad j = 1, \ldots, p \quad \text{und} \\ g_j(x^*) < 0, \quad \lambda_j^* = 0, \quad j = p+1, \ldots, m. \end{aligned}$$

Ferner nehmen wir momentan an, daß in x^* die hinreichenden Bedingungen zweiter Ordnung erfüllt sind, d.h. daß die Hessematrix

$$\nabla^2_{xx} L_{(P)}(x^*, \lambda^*) = \nabla^2_{xx} f(x^*) + \sum_{j=1}^{m} \lambda_j^* \nabla^2_{xx} g_j(x^*)$$

der klassischen Lagrangefunktion

$$L_{(P)}(x, \lambda) = \begin{cases} f(x) + \sum\limits_{j=1}^{m} \lambda_j g_j(x) & \text{, falls } \lambda \geq 0 \\ -\infty & \text{, sonst} \end{cases}$$

zum ursprünglichen Problem (P) positiv definit auf dem Unterraum

$$Z = \{s : s^T \nabla_x g_j(x^*) = 0, j = 1, \ldots, p\}$$

ist und die Gradienten $\{\nabla_x g_j(x^*) : j = 1, \ldots, p\}$ der in x^* aktiven Restriktionen linear unabhängig sind. Dann gilt

$$\begin{aligned} \nabla^2_{xx} \mathcal{F}(x^*, \lambda^*, k) &= \nabla^2_{xx} f(x^*) + k \sum_{j=1}^{m} \lambda_j^* \frac{\nabla g_j(x^*) \nabla g_j(x^*)^T}{(1 - k g_j(x^*))^2} + \sum_{j=1}^{m} \frac{\lambda_j^*}{1 - k g_j(x^*)} \nabla^2_{xx} g_j(x^*) \\ &= \nabla^2_{xx} f(x^*) + k \sum_{j=1}^{p} \nabla_x g_j(x^*) \nabla_x g_j(x^*)^T + \sum_{j=1}^{m} \lambda_j^* \nabla^2_{xx} g_j(x^*) \\ &= \nabla^2_{xx} f(x^*) + \sum_{j=1}^{m} \lambda_j^* \nabla^2_{xx} g_j(x^*) + k \sum_{j=1}^{p} \nabla_x g_j(x^*) \nabla_x g_j(x^*)^T \\ &= \nabla^2_{xx} L_{(P)}(x^*, \lambda^*) + k \sum_{j=1}^{r} \nabla_x g_j(x^*) \nabla_x g_j(x^*)^T. \end{aligned} \tag{3.5}$$

Da nach unserer Annahme $\nabla^2_{xx} L_{(P)}(x^*, \lambda^*)$ positiv semidefinit auf $\mathbb{R}^n$ (wegen der Konvexität von f und der g_j) und außerdem positiv definit auf Z ist, folgt aus (3.5), daß die Matrix $\nabla^2_{xx} \mathcal{F}(x^*, \lambda^*, k)$ auf ganz $\mathbb{R}^n$ positiv definit ist. Aus Stetigkeitsgründen ist dann auch $\nabla^2_{xx} \mathcal{F}(x, \lambda^*, k)$ für x nahe bei x^* positiv definit. Daher ist $\mathcal{F}(., \lambda^*, k)$ strikt konvex in einer Umgebung von x^*, und x^* ist das *eindeutig bestimmte* globale Minimum von $\mathcal{F}(., \lambda^*, k)$:

$$x^* = \arg \min_{x \in \text{int } S_k} \mathcal{F}(x, \lambda^*, k) = \arg \min_{x \in \mathbb{R}^n} \mathcal{F}(x, \lambda^*, k). \tag{3.6}$$

In diesem Fall ermöglicht es uns die Kenntnis des optimalen Lagrangemultiplikators λ^*, die Lösung des Optimierungsproblem (P) auf die Lösung eines einzigen (unrestringierten) Problems (3.6) zu reduzieren.

3.2 Modifizierte Barrierefunktionen für quadratische Programme

Betrachten wir nun das konvexe quadratische Programm (QP) aus Abschnitt 1.2:

$$\text{(QP)} \quad \begin{array}{ll} \min & \frac{1}{2}x^TQx + c^Tx \\ x \in \mathbb{R}^n: & Ax \geq b, \end{array}$$

mit positiv semidefiniter Matrix Q. Die Modifizierte Barrierefunktion zu (QP) lautet dann

$$\mathcal{F}(x, \lambda, k) = \frac{1}{2}x^TQx + c^Tx - \frac{1}{k}\sum_{j=1}^{m} \lambda_j \log(k \cdot (a_j^Tx - b_j) + 1).$$

Um wichtige Eigenschaften der Modifizierten Barrierefunktion $\mathcal{F}(x, \lambda, k)$ herzuleiten, zeigen wir zunächst das folgende

Lemma 3.2.1 *Sei $K \subset \mathbb{R}^n$ konvex, abgeschlossen und unbeschränkt. Dann gibt es einen Vektor $v \neq 0$, so daß $x + \alpha v \in K$ für alle $x \in K$ und alle $\alpha \geq 0$.*

Beweis: Zu $\nu \in \mathbb{N}$ gibt es wegen der Unbeschränktheit von K ein $y^\nu \in K$ mit $\|y^\nu\|_2 \geq \nu$. Sei $v^\nu = y^\nu/\|y^\nu\|_2$. Dann ist $\|v^\nu\|_2 = 1$ und die Folge $(v^\nu)_\nu$ somit in der kompakten Menge $M = \{s : \|s\|_2 = 1\}$ enthalten. Daher enthält $(v^\nu)_\nu$ eine konvergente Teilfolge $(v^{\nu_\ell})_\ell$, etwa $\lim\limits_{\ell\to\infty} v^{\nu_\ell} = v \in M$. Wir zeigen nun, daß für $x \in K$ und $\alpha \geq 0$ gilt: $x + \alpha v \in K$. Man betrachte die Folge $(w^\ell)_\ell$ mit $w^\ell = \frac{\alpha}{\|v^{\nu_\ell}\|_2}v^{\nu_\ell} + \left(1 - \frac{\alpha}{\|v^{\nu_\ell}\|_2}\right)x$. Für großes ℓ gilt wegen $\|v^{\nu_\ell}\|_2 \to \infty$, daß $0 \leq \frac{\alpha}{\|v^{\nu_\ell}\|_2} \leq 1$, so daß dann wegen der Konvexität von K gilt: $w^\ell \in K$. Wir schreiben w^ℓ anders als $w^\ell = x + \alpha\frac{v^{\nu_\ell}}{\|v^{\nu_\ell}\|_2} - \alpha\frac{x}{\|v^{\nu_\ell}\|_2}$. Daraus wird ersichtlich, daß $\lim\limits_{\ell\to\infty} w^\ell = x + \alpha v$. Wie oben gesehen, ist $w^\ell \in K$ für großes ℓ. Somit folgt $x + \alpha v \in \bar{K} = K$, da K nach Voraussetzung abgeschlossen ist. Dies zeigt die Behauptung. $\square$

Damit sind die Voraussetzungen geschaffen, um den folgenden Satz zu zeigen, der die Ergebnisse von [29, Lemma 2.1] auf den quadratischen Fall überträgt:

Satz 3.2.2 *Sei $S = \{x \in \mathbb{R}^n : Ax \geq b\} = \{x \in \mathbb{R}^n : a_j^Tx - b_j \geq 0, j = 1, 2, \ldots, m\}$ nichtleer und beschränkt. Dann gelten folgende Aussagen:*

(i) *Für jedes $k > 0$ ist $S_k = \{x : a_j^Tx - b_j \geq -\frac{1}{k}, j = 1, 2, \ldots, m\}$ nichtleer, konvex, beschränkt mit nichtleerem Inneren* int S_k.

(ii) *Für jedes feste $\lambda \in \mathbb{R}^m$ mit $\lambda > 0$ und jedes $k > 0$ ist die Modifizierte Frisch–Funktion*

$$\mathcal{F}(x, \lambda, k) = \frac{1}{2}x^T Qx + c^T x - \frac{1}{k}\sum_{j=1}^{m} \lambda_j \log(k \cdot (a_j^T x - b_j) + 1)$$

als Funktion von x strikt konvex in S_k.

(iii) *Für jedes $\lambda \in \mathbb{R}^m$ mit $\lambda > 0$ und $k > 0$ existiert ein eindeutiges $x(\lambda, k) \in \operatorname{int} S_k$ mit*

$$x(\lambda, k) = \operatorname{argmin}\{\mathcal{F}(x, \lambda, k) : x \in S_k\}.$$

Es gilt $\nabla_x \mathcal{F}(x(\lambda, k), \lambda, k) = 0$.

Beweis:

(i) Die Aussagen, daß S_k nichtleer und konvex ist, sind trivial. Auch $\operatorname{int} S_k \neq \emptyset$ folgt sofort aus $S \subset \operatorname{int} S_k$ und $S \neq \emptyset$.

Wir nehmen an, S_k sei nicht beschränkt. Nach Lemma 3.2.1 existiert dann wegen der Konvexität und Abgeschlossenheit von S_k ein $v \neq 0$, so daß $x + \alpha v \in S_k$ für alle $x \in S_k$ und alle $\alpha \geq 0$, also $k(a_j^T(x + \alpha v) - b_j) + 1 \geq 0$ für $j = 1, \ldots, m$. Daher gilt für $j = 1, \ldots, m$, daß $k(a_j^T x - b_j) + 1 + k\alpha a_j^T v \geq 0$ für alle $x \in S_k$ und alle $\alpha \geq 0$. Daraus folgt dann wiederum

$$a_j^T v \geq 0 \text{ für } j = 1, \ldots, m. \tag{3.7}$$

Wäre nämlich $a_{j_0}^T v < 0$ für ein $j_0 \in \{1, \ldots, m\}$, so könnte man für festes $x \in S_k$ ein $\alpha > 0$ finden, so daß $k\alpha a_{j_0}^T v < k(a_{j_0}^T x - b_{j_0}) + 1$, also $k(a_{j_0}^T(x + \alpha v) - b_j) + 1 < 0$, im Widerspruch zu $x + \alpha v \in S_k$.

Sei nun $x' \in S$ beliebig. Dann gilt wegen (3.7) für $j = 1, \ldots, m$ und alle $\alpha \geq 0$:

$$a_j^T(x' + \alpha v) - b_j = \underbrace{a_j^T x' - b_j}_{\geq 0} + \alpha \underbrace{a_j^T v}_{\geq 0} \geq 0,$$

also $x' + \alpha v \in S$. Damit folgt aber, daß S selbst unbeschränkt ist im Widerspruch zur Voraussetzung. Folglich muß S_k beschränkt sein.

(ii) Für $x \in \operatorname{int} S_k$ ist $\mathcal{F}(., \lambda, k)$ sicherlich zweimal stetig differenzierbar. Es gilt

$$\nabla_x \mathcal{F}(x, \lambda, k) = Qx + c - \sum_{j=1}^{m} \lambda_j \frac{a_j}{k(a_j^T x - b_j) + 1}, \tag{3.8}$$

sowie

$$\nabla_{xx}^2 \mathcal{F}(x, \lambda, k) = Q + k \sum_{j=1}^{m} \lambda_j \frac{a_j a_j^T}{[k(a_j^T x - b_j) + 1]^2}. \tag{3.9}$$

Aus der Theorie der konvexen Funktionen ist bekannt, daß $\mathcal{F}(.,\lambda,k)$ auf int S_k strikt konvex ist, falls $\nabla^2_{xx}\mathcal{F}(x,\lambda,k)$ positiv definit für $x \in \text{int } S_k$ ist. Sei $v \in \mathbb{R}^n$. Dann gilt

$$v^T\nabla^2_{xx}\mathcal{F}(x,\lambda,k)v = v^TQv + k\sum_{j=1}^{m}\lambda_j\frac{(a_j^Tv)^2}{[k(a_j^Tx-b_j)+1]^2} \geq 0,$$

da Q positiv semidefinit und $\lambda > 0$ ist. Wäre nun $v^T\nabla^2_{xx}\mathcal{F}(x,\lambda,k)v = 0$, so folgte wegen $\lambda_j > 0$ sofort $a_j^Tv = 0$ für $j = 1,\ldots,m$, was wie oben der Beschränktheit von S widerspricht.

(iii) $\mathcal{F}(.,\lambda,k)$ ist stetig auf int S_k. Da $\mathcal{F}(x,\lambda,k)$ für $x \to \partial S_k$ gegen Unendlich strebt und S_k beschränkt ist, folgt, daß $\mathcal{F}(.,\lambda,k)$ ein Minimum $x(\lambda,k)$ in int S_k besitzt. Dieses erfüllt dann notwendigerweise $\nabla_x\mathcal{F}(x(\lambda,k),\lambda,k) = 0$. Wegen der strikten Konvexität von $\mathcal{F}(.,\lambda,k)$ ist das Minimum eindeutig.

Damit ist der Satz bewiesen. □

Man möchte auch für den Fall, daß S nicht beschränkt ist, ähnliche Resultate wie diejenigen in Satz 3.2.2 haben. Es ist sofort klar, daß S_k auch in diesem Fall eine konvexe Menge mit nichtleerem Inneren ist, sofern nur S selbst nichtleer war. Aus (3.9) wird auch sofort ersichtlich, daß $\mathcal{F}(.,\lambda,k)$ wieder konvex, nicht jedoch notwendigerweise strikt konvex ist. Man ist versucht, wie folgt zu schließen: Da $\mathcal{F}(.,\lambda,k)$ auf int S_k stetig und konvex ist und für $x \to \partial S_k$ gegen $+\infty$ strebt, existiert ein (jetzt möglicherweise nicht eindeutiges) $x(\lambda,k) \in \arg\ \min\{\mathcal{F}(x,\lambda,k) : x \in S_k\}$, so daß $\nabla_x\mathcal{F}(x(\lambda,k),\lambda,k) = 0$. Diese an für sich plausibel erscheindende Argumentation ist aber so nicht korrekt. Man betrachte dazu das folgende Beispiel:

Beispiel 3.1

$$\begin{array}{ll}(\text{QP}) & \min 0 \\ & x \in \mathbb{R}: \quad x \geq 0\end{array}$$

Die Modifizierte Barrierefunktion zu (QP) ist dann durch

$$\mathcal{F}(x,\lambda,k) = -\frac{1}{k}\lambda\log(kx+1)$$

gegeben. Für $x \to \infty$ strebt $\mathcal{F}(x,\lambda,k)$ für festes $\lambda > 0$ und $k > 0$ gegen $-\infty$. Folglich besitzt $F(.,\lambda,k)$ kein Minimum auf $S_k = \{x : x \geq -\frac{1}{k}\}$. Die Abbildung 3.1 zeigt den Graphen von $\mathcal{F}(.,\lambda,k)$ für $\lambda = 1$ und $k = 2$. ◇

Nun ist im letzten Beispiel die Zielfunktion des Optimierungsproblems (QP) konstant. Im nächsten Beispiel fügen wir daher eine Dimension hinzu, um einen nicht ganz so pathologischen Fall zu erhalten:

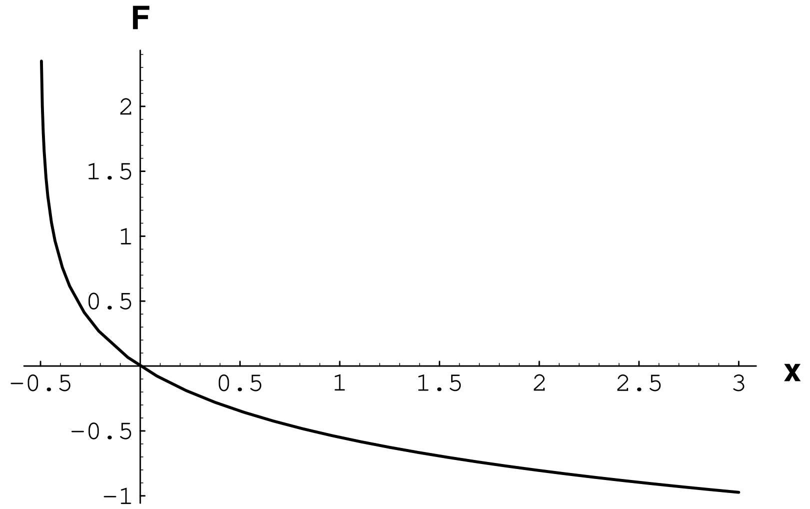

Abbildung 3.1:

Beispiel 3.2

$$\text{(QP)} \quad \min x_1 \\ x \in \mathbb{R}^2 : \quad x \geq 0$$

Die Menge S der für (QP) zulässigen Lösungen ist der positive Quadrant, die Menge der Optimallösungen ist $S^* = \{(0, x_2) : x_2 \geq 0\}$. In der Abbildung 3.2 sind die Menge der zulässigen und optimalen Lösungen für (QP) veranschaulicht.

Die Modifizierte Barrierefunktion lautet dann

$$\mathcal{F}(x_1, x_2, \lambda_1, \lambda_2, k) = x_1 - \frac{1}{k}\lambda_1 \log(kx_1 + 1) - \frac{1}{k}\lambda_2 \log(kx_2 + 1).$$

Die Abbildung 3.3 zeigt den Graphen der Modifizierten Barrierefunktion $\mathcal{F}(., \lambda_1, \lambda_2, k)$ für $\lambda_1 = 3$, $\lambda_2 = 3$ und $k = 2$.

Für festes x_1 strebt die Modifizierte Barrierefunktion gegen $-\infty$, wenn x_2 gegen $+\infty$ strebt. Dies läßt sich auch weder durch die die Wahl von $\lambda > 0$ noch der von $k > 0$ ändern. Daher existiert nicht einmal ein endliches x, welches $\mathcal{F}(., \lambda, k)$ auf $S_k = \{x : x \geq -\frac{1}{k}\}$ minimiert. Insbesondere existiert kein x, so daß der Gradient von $\mathcal{F}(., \lambda, k)$ an der Stelle x verschwindet. $\diamond$

Die beiden Beispiele zeigen, daß ein so einfacher Schluß wie unter der Voraussetzung der Beschränktheit von S nicht ohne weiteres möglich ist. Offenbar kann man nicht alle Voraussetzungen an die Struktur von (QP) fallenlassen, d.h. man kann die Beschränktheit von S nicht ersatzlos streichen. Deshalb werden wir versuchen, die Beschränktheit von S durch eine etwas schwächere Voraussetzung zu ersetzen. In beiden Beispielen

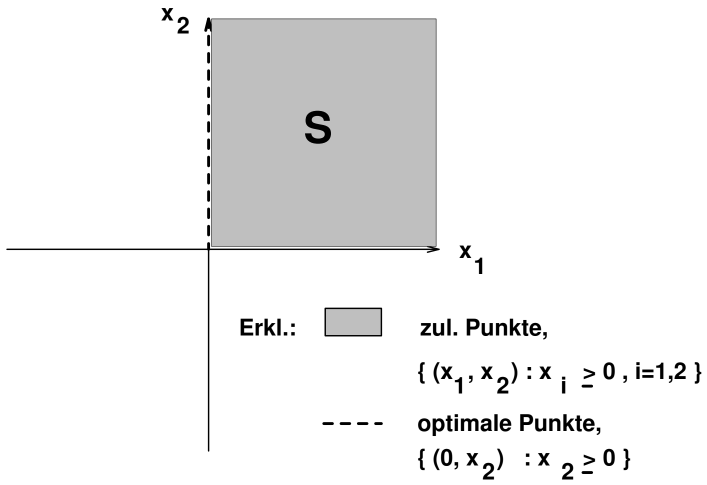

Abbildung 3.2:

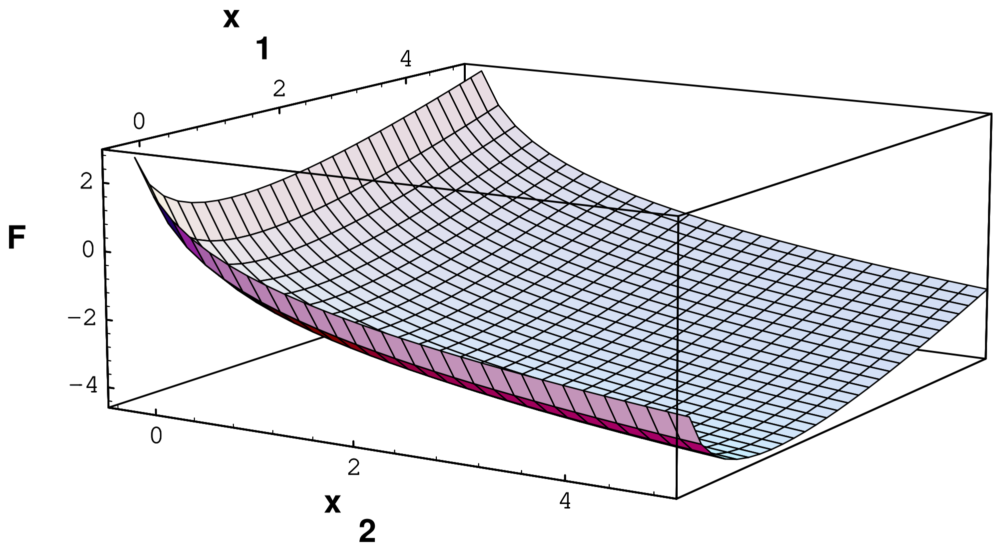

Abbildung 3.3:

ist die Menge der Optimallösungen unbeschränkt. Daher werden wir voraussetzen, daß die Menge S^* der Optimallösungen von (QP) beschränkt und nichtleer ist. In diesem Zusammenhang beweisen Fiacco und McCormick in [7] den folgenden

Satz 3.2.3 *Seien f und g_j $(j = 1, \ldots, m)$ konvex und sei die Menge S^* der Optimallösungen des konvexen Programms*

$$\begin{array}{lll} \text{(P)} & \min & f(x) \\ & x \in \mathbb{R}^n : & g_j(x) \leq 0, \quad j = 1, \ldots, m \end{array}$$

nichtleer und beschränkt. Dann ist für beliebige Konstanten $\omega_0, \omega_1, \ldots, \omega_m$ die Menge

$$\{x \in \mathbb{R}^n : f(x) \leq \omega_0, g_j(x) \leq \omega_j, j = 1, \ldots, m\}$$

eine (möglicherweise leere) kompakte Teilmenge des $\mathbb{R}^n$.

Beweis: [7, Corollary 20]. □

Mit Hilfe dieses Ergebnisses zeigen wir als nächstes eine Hilfsaussage.

Lemma 3.2.4 *Sei die Menge S^* der Optimallösungen von* (QP) *beschränkt und nichtleer. Dann besitzt für jedes $k > 0$ und $\lambda \geq 0$ die Modifizierte Barrierefunktion $\mathcal{F}(., \lambda, k)$ zu* (QP) *beschränkte Niveaumengen, d.h. für jedes $c \in \mathbb{R}$ ist*

$$\mathrm{lev}_c := \{x \in S_k : \mathcal{F}(x, \lambda, k) \leq c\}$$

eine beschränkte (und somit kompakte) Teilmenge von $\mathrm{int}\, S_k$.

Beweis: Wir führen den Beweis durch Widerspruch. Zunächst ist klar, daß lev_c bereits eine Teilmenge von $\mathrm{int}\, S_k$ ist, da $\mathcal{F}(., \lambda, k)|_{\partial S_k} \equiv +\infty$. Wäre die Behauptung des Lemmas nun falsch, so existierte ein $c \in \mathbb{R}$ und eine Folge $(v^\ell)_\ell$ mit folgenden Eigenschaften:

(i) $\mathcal{F}(v^\ell, \lambda, k) \leq c < +\infty$

(ii) $\lim\limits_{\ell \to \infty} \|v^\ell\|_2 = +\infty$

(iii) $v^\ell \in S_k$

(iv) $a_j^T v^\ell - b_j > -\frac{1}{k}$, d.h. $v^\ell \in \mathrm{int}\, S_k$

Dabei ergibt sich die strikte Ungleichung in der letzten Aussage aus der ersten Eigenschaft und aus der Tatsache, daß $\mathcal{F}(x,\lambda,k) = +\infty$ für jedes x gilt, das nur $a_j^T x - b_j = -1/k$ für ein j erfüllt. Sei nun $M := \{x \in S_k : q(x) \leq q^* + \frac{1}{2}\}$. M ist nichtleer, denn offenbar ist jede Optimallösung von (QP) ein Element von M. Nach Satz 3.2.3 ist M kompakt. Nach elementaren Sätzen der Topologie ist mit M ist auch der topologische Rand ∂M von M kompakt.

Wegen der Beschränktheit von M gilt wegen (ii): $v^\ell \notin M$ für $\ell > N$, wobei $N > 0$ eine geeignete Konstante ist. Sei x^* eine beliebige Optimallösung von (QP) und sei $\ell > L$ beliebig. Da $x^* \in M$, $v^\ell \notin M$ und M beschränkt ist, existiert auf der Verbindungsstrecke zwischen x^* und v^ℓ ein $\xi^\ell = t^\ell v^\ell + (1-t^\ell)x^*$ $(0 < t^\ell < 1)$ mit $\xi^\ell \in \partial M$. Wir behaupten nun, daß t^ℓ für $\ell \to \infty$ gegen 0 strebt. In der Tat ergibt sich aus der umgekehrten Dreicksungleichung

$$\|\xi^\ell\|_2 \geq t^\ell \|v^\ell\|_2 - (1-t^\ell)\|x^*\|_2,$$

d.h.

$$0 < t^\ell \leq \frac{\|\xi^\ell\|_2 + (1-t^\ell)\|x^*\|_2}{\|v^\ell\|_2} \leq \frac{\omega_\partial + 2\|x^*\|_2}{\|v^\ell\|_2}, \tag{3.10}$$

wobei $\omega_\partial = \max\{\|\xi\|_2 : \xi \in \partial M\}$. Man beachte, daß $\omega_\partial < +\infty$ wegen der Kompaktheit von ∂M. Da der am weitesten rechts stehende Ausdruck in (3.10) für $\ell \to \infty$ gegen 0 strebt, muß auch t^ℓ dann gegen 0 streben.

Wegen der Konvexität von q gilt ferner:

$$q(\xi^\ell) = q(t^\ell v^\ell + (1-t^\ell)x^*) \leq t^\ell q(v^\ell) + (1-t^\ell)q(x^*) = t^\ell q(v^\ell) + (1-t^\ell)q^*,$$

also

$$q(v^\ell) \geq \frac{q(\xi^\ell) - (1-t^\ell)q^*}{t^\ell} = \frac{q^* + \frac{1}{2} - (1-t^\ell)q^*}{t^\ell} = q^* + \frac{1}{2t^\ell}. \tag{3.11}$$

Dabei haben wir für das erste Gleichheitszeichen benutzt, daß wegen $\xi^\ell \in \partial M$ gilt: $q(\xi^\ell) = q^* + 1/2$. Weiterhin folgt aus der Definition von $\xi^\ell = t^\ell v^\ell + (1-t^\ell)x^*$, daß

$$\begin{aligned} a_j^T v^\ell - b_j &= \frac{a_j^T \xi^\ell - (1-t^\ell)a_j^T x^*}{t^\ell} - b_j \\ &= a_j^T x^* - b_j + \frac{a_j^T(\xi^\ell - x^*)}{t^\ell} \\ &\leq a_j^T x^* - b_j + \frac{1}{t^\ell}\|a_j\|_2 \cdot \|\xi^\ell - x^*\|_2. \end{aligned}$$

Die letzte Abschätzung ergibt sich durch Anwendung der Cauchy–Schwarz–Ungleichung auf das Skalarprodukt $a_j^T(\xi^\ell - x^*)$. Wegen der Kompaktheit von ∂M ist $\sup\{\|\xi - x^*\|_2 : \xi \in \partial M\} < +\infty$. Da nach Konstruktion $\xi^\ell \in \partial M$, folgt daher aus der letzten Rechnung mit

$$\omega := (\max\{\|a_j\|_2 : j = 1, \ldots, m\}) \cdot (\sup\{\|\xi - x^*\|_2 : \xi \in \partial M\}) < +\infty$$

für $\ell > N$ die Ungleichung

$$a_j^T y^\ell - b_j \leq a_j^T x^* - b_j + \frac{1}{t^\ell}\omega, \quad j = 1, \ldots, m. \tag{3.12}$$

Mit Hilfe von (3.11) und (3.12) folgt dann

$$\begin{aligned}
\mathcal{F}(v^\ell, \lambda, k) &= q(v^\ell) - \frac{1}{k}\sum_{j=1}^m \lambda_j \log(k(a_j^T v^\ell - b_j) + 1) \\
&\overset{(3.11)}{\geq} q^* + \frac{1}{2t^\ell} - \frac{1}{k}\sum_{j=1}^m \lambda_j \log(k(a_j^T v^\ell - b_j) + 1) \\
&\overset{(3.12)}{\geq} q^* + \frac{1}{2t^\ell} - \frac{1}{k}\sum_{j=1}^m \lambda_j \log(k(a_j^T x^* - b_j + \frac{\omega}{t^\ell}) + 1) \\
&= q^* + \frac{1}{t^\ell}\left(\frac{1}{2} - \frac{1}{k}\sum_{j=1}^m \lambda_j t^\ell \log(k(a_j^T x^* - b_j + \frac{\omega}{t^\ell}) + 1)\right) \qquad (3.13)
\end{aligned}$$

Der nächste Schritt ist nun zu zeigen, daß

$$\lim_{\ell\to\infty} t^\ell \log(k(a_j^T x^* - b_j + \frac{\omega}{t^\ell}) + 1) = 0 \tag{3.14}$$

für $j = 1, \ldots, m$ gilt. Dies folgt aus $\lim\limits_{s\to\infty} \frac{1}{s}\log s = 0$ und $\lim\limits_{\ell\to\infty} t^\ell = 0$. Die Gleichung (3.14) und erneut $t^\ell \to 0$, ermöglichen es uns, ein $N' > N$ zu finden, so daß folgende Aussagen für $\ell > N'$ richtig sind:

$$\frac{1}{2} - \frac{1}{k}\sum_{j=1}^m \lambda_j t^\ell \log(k(a_j^T x^* - b_j + \frac{\omega}{t^\ell})) > \frac{1}{4}$$

und

$$\frac{1}{t^\ell} > 4\,(c - q^*)\,.$$

Dann aber gewinnen wir aus (3.13), daß für $\ell > N'$ die Ungleichung

$$\mathcal{F}(v^\ell, \lambda, k) > q^* + \frac{1}{t^\ell}\frac{1}{4} > q^* + 4(c - q^*)\frac{1}{4} = c$$

erfüllt ist, im Widerspruch zu $\mathcal{F}(v^\ell, \lambda, k) \leq c$ für *alle* ℓ. $\square$

Mit Hilfe des letzten Lemmas kann man doch noch (fast) alle Aussagen von Satz 3.2.2 für den Fall retten, daß zwar S möglicherweise unbeschränkt, die Menge der Optimallösungen von (QP) jedoch beschränkt ist.

Satz 3.2.5 *Die Menge S^* der Optimallösungen von* (QP) *sei beschränkt und nichtleer. Dann gilt:*

(i) *Für jedes $k > 0$ ist $S_k = \{x : a_j^T x - b_j \geq -\frac{1}{k}, j = 1, 2, \ldots, m\}$ nichtleer, konvex mit nichtleerem Inneren* int S_k.

(ii) *Für jedes feste $\lambda \in \mathbb{R}^m$ mit $\lambda > 0$ und jedes $k > 0$ ist die Modifizierte Frisch–Funktion*

$$\mathcal{F}(x, \lambda, k) = \frac{1}{2} x^T Q x + c^T x - \frac{1}{k} \sum_{j=1}^{m} \lambda_j \log(k \cdot (a_j^T x - b_j) + 1)$$

als Funktion von x konvex in S_k.

(iii) *Für jedes $\lambda \in \mathbb{R}^m$ mit $\lambda > 0$ und $k > 0$ existiert $x(\lambda, k) \in \operatorname{int} S_k$ mit*

$$x(\lambda, k) \in \arg\ \min\{\mathcal{F}(x, \lambda, k) : x \in S_k\}$$

und es gilt $\nabla_x \mathcal{F}(x(\lambda, k), \lambda, k) = 0$.

Beweis: Die Aussagen (i) und (ii) folgen aus den entsprechenden Beweisen für Satz 3.2.2. Zu zeigen ist daher nur die Behauptung (iii). Da S_k nichtleer ist, gibt es ein $x^0 \in S_k$. Nach Lemma 3.2.4 ist

$$M = \operatorname{lev}_{\mathcal{F}(x^0, \lambda, k)} = \{x \in S_k : \mathcal{F}(x, \lambda, k) \leq \mathcal{F}(x^0, \lambda, k)\}$$

eine kompakte Teilmenge von int S_k. Wegen $x^0 \in M$ ist M nichtleer. Da $\mathcal{F}(., \lambda, k)$ auf int S_k und somit insbesondere auf M stetig ist, existiert ein $x(\lambda, k) \in \arg\ \min\{\mathcal{F}(x, \lambda, k) : x \in M\}$. Offenbar ist $x(\lambda, k)$ dann auch globales Minimum von $\mathcal{F}(., \lambda, k)$ auf S_k. Da $x(\lambda, k) \in M \subseteq \operatorname{int} S_k$, folgt notwendigerweise: $\nabla_x \mathcal{F}(x(\lambda, k), \lambda, k) = 0$. □

3.3 Hauptergebnisse von Polyak über Modifizierte Barrierefunktionen

In diesem Abschnitt wird zunächst das Hauptergebnis von R. Polyak aus [28] über die Modifizierten Barrierefunktionen bei Anwendung auf das lineare Programm

$$\begin{array}{lll} \text{(LP)} & \min & c^T x \\ & x \in \mathbb{R}^n : & Ax \geq b \end{array}$$

vorgestellt, das sich als Spezialfall aus (QP) für $Q = 0$ ergibt. Danach werden die Ergebnisse von Polyak aus [27] für allgemeine konvexe Probleme auf (QP) übertragen und für diesen Fall bewiesen. Der Beweis aus [27] kann durch die spezielle Struktur von (QP) an einigen Stellen übersichtlicher gestaltet werden.

Betrachen wir zunächst den linearen Fall, d.h. $Q = 0$. Dann reduziert sich (QP) zu (LP) und das zu (QP) duale quadratische Problem (QPD) auf das lineare Programm

$$\begin{array}{lrcl} \text{(LPD)} \quad \max & b^T\lambda & & \\ x \in \mathbb{R}^m : & A^T\lambda & = & c \\ & \lambda & \geq & 0. \end{array}$$

In der Literatur wird (LPD) oftmals als das primäre lineare Programm bezeichnet, während man (LP) meistens das duale lineare Programm nennt. Die Modifizierte Barrierefunktion zu (LP) ist durch

$$\mathcal{F}(x, \lambda, k) = c^T x - \frac{1}{k}\sum_{j=1}^{m} \lambda_j \log(k \cdot (a_j^T x - b_j) + 1)$$

gegeben. Polyak setzt für seine Ergebnisse in [28] voraus, daß sowohl (LP) als auch (LPD) nichtentartet sind, d.h. er macht die

Voraussetzung 1 *Für das primär–duale Paar* (LP), (LPD) *von linearen Programmen gelte:*

(i) *Die Optimallösungen x^* von* (LP) *und λ^* von* (LPD) *erfüllen:*

$$\begin{array}{lll} a_j^T x^* = b_j, & \lambda_j^* > 0, & j = 1, \ldots, n \\ a_j^T x^* > b_j, & \lambda_j^* = 0, & j = n+1, \ldots, m \end{array} \tag{3.15}$$

(ii) $\mathrm{rg} A = n$ *und* $\mathrm{rg}(a_1, \ldots, a_n) = n$.

Man kann zeigen, daß unter dieser Voraussetzung x^* und λ^* eindeutige Lösung von (LP) bzw. (LPD) sind. Unter der Voraussetzung 1 bezeichnet Polyak

$$\sigma = \min\Big(\min\{\lambda_j^*, j = 1, \ldots, n\}, \min\{a_j^T x^* - b_j, j = n+1, \ldots, m\}\Big) > 0$$

als das Maß an Nichtdegeneriertheit von (LP) und (LPD). Um die Lesbarkeit der Darstellung zu vereinfachen, führen wir die Notation

$$\Delta(x, k) = \mathrm{diag}(k(a_1^T x - b_1) + 1, \ldots, k(a_m^T x - b_m) + 1) \tag{3.16}$$

ein. Für $k_0, \delta, \varepsilon > 0$ sei die Menge $D(\lambda^*, k_0, \delta, \varepsilon)$ durch

$$D(\lambda^*, k_0, \delta, \varepsilon) = \Big\{(\lambda, k) \in \mathbb{R}_+^m \times \mathbb{R}_+ : \varepsilon \leq \lambda_j, j = 1, \ldots, n, \|\lambda^* - \lambda\|_\infty \leq k\delta, k \geq k_0\Big\}$$

definiert. An dieser Stelle erscheint es als nützlich, etwas mehr auf die Menge $D(\lambda^*, k_0, \delta, \varepsilon)$ einzugehen und die zunächst recht komplex anmutende Definition auch anschaulich und geometrisch zu erläutern. Sei dazu j fest und betrachten wir die Menge

$$D_j = D_j(\lambda^*, k_0, \delta, \varepsilon) = \Big\{(\lambda_j, k) : \exists \lambda = (\lambda_1, \ldots, \lambda_m)^T : (\lambda, k) \in D(\lambda^*, k_0, \delta, \varepsilon)\Big\}.$$

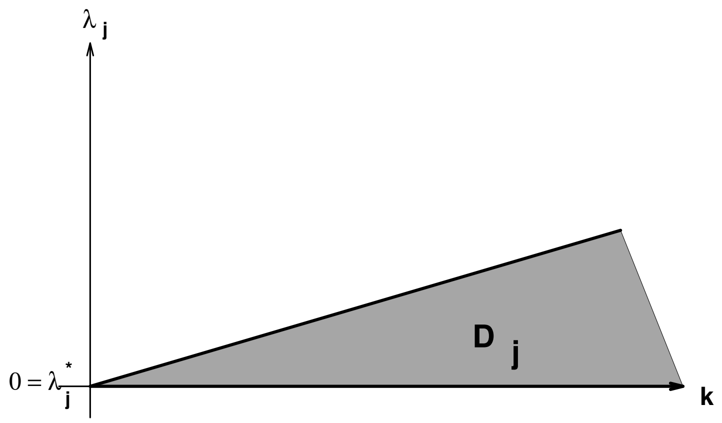

Abbildung 3.4:

Die Menge D_j ist eine Projektion der Menge $D(\lambda^*, k_0, \delta, \varepsilon)$ auf die j–te Komponente der Vektoren λ und auf den Barriereparameter k. Es bietet sich jetzt an, eine Unterscheidung zu treffen, ob $1 \leq j \leq n$ oder $n < j \leq m$ gilt. Betrachten wir zunächst den Fall, daß $n < j \leq m$, d.h. den Fall der nichtaktiven Indizes. Dann schreibt sich die Menge D_j als:

$$\begin{aligned} D_j &= \{(\lambda_j, k) : \lambda_j \geq 0, k \geq k_0 \text{ und } |\lambda_j - \lambda_j^*| \leq k\delta\} \\ &= \{(\lambda_j, k) : \lambda_j \geq 0, k \geq k_0 \text{ und } |\lambda_j| \leq k\delta\}. \end{aligned} \tag{3.17}$$

Die letzte Gleichheit folgt dabei daraus, daß j ein nichtaktiver Index ist, und somit $\lambda_j^* = 0$ gelten muß. Betrachtet man die Beziehungen in (3.17), so sieht man, daß D_j nichts anderes als ein Kegel mit Scheitelpunkt $(0, 0)$ ist. Die Abbildung 3.4 stellt D_j graphisch dar.

Wenden wir uns nun den aktiven Indizes $j \in \{1, \ldots, n\}$ zu. Dann besitzt D_j die Form

$$D_j = \left\{(\lambda_j, k) : \lambda_j \geq \varepsilon, k \geq k_0 \text{ und } |\lambda_j - \lambda_j^*| \leq k\delta\right\}. \tag{3.18}$$

Wiederum enthält (3.18) eine Kegelbedingung. Wegen $\lambda_j \geq \varepsilon$ wird jedoch ein Teil des Kegels weggeschnitten. In der Abbildung 3.5 wird D_j für die aktiven Indizes $j = 1, \ldots, n$ veranschaulicht.

Zusammenfassend kann man also sagen, daß in der Menge $D(\lambda^*, k_0, \delta, \varepsilon)$ an die einzelnen Koordinateneinträge der Elemente λ eine Art Kegelbedingung gestellt wird. Grob gesprochen ist (λ, k) genau dann ein Element von $D(\lambda^*, k_0, \delta, \varepsilon)$, wenn k größer als der Schwellenwert k_0 ist und λ „nahe genug“ bei λ^* liegt, wobei „nahe genug“ von k und der Konstanten $\delta > 0$ abhängt.

Polyak zeigt nun in [28, Theorem 4] den folgenden

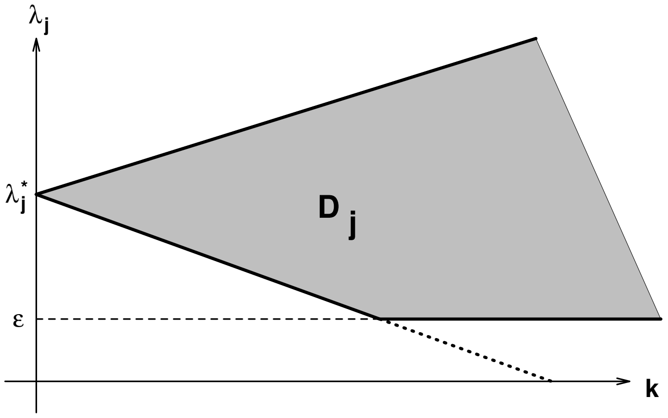

Abbildung 3.5:

Satz 3.3.1 *Für das primär–duale Paar von linearen Programmen* (LP), (LPD) *sei die Voraussetzung 1 erfüllt. Dann gibt es* $k_0 > 0$, $\delta > 0$, *so daß für alle* $0 < \varepsilon < \sigma$ *und alle* $(\lambda, k) \in D(\lambda^*, k_0, \delta, \varepsilon)$ *folgende Aussagen richtig sind:*

(i) *Es existiert ein eindeutiges* $\hat{x}(\lambda, k) = \arg\ \min\{\mathcal{F}(x, \lambda, k) | x \in \mathbb{R}^n\}$ *mit*

$$\nabla_x \mathcal{F}(x(\lambda, k), \lambda, k)) = 0.$$

(ii) *Definiert man*

$$\hat{\lambda}(\lambda, k) = \Delta(\hat{x}(\lambda, k), k)^{-1} \lambda, \tag{3.19}$$

so gibt es eine Konstante $c_0 > 0$ *(unabhängig von* $k \geq k_0$*), so daß für das Paar* $\hat{x}(\lambda, k), \hat{\lambda}(\lambda, k)$ *die Abschätzung*

$$\max\left\{\|\hat{x}(\lambda, k) - x^*\|_\infty, \|\hat{\lambda}(\lambda, k) - \lambda^*\|_\infty\right\} \leq \frac{c_0}{k}\|\lambda - \lambda^*\|_\infty \tag{3.20}$$

gilt.

(iii) *Es gilt* $(\hat{\lambda}(\lambda, k), k) \in D(\lambda^*, k_0, \delta, \varepsilon)$.

Wir wollen nun einen analogen Satz für den konvexen quadratischen Fall, d.h. für positiv semidefinites $Q \neq 0$, beweisen, wobei wir dafür auf die Ergebnisse von Polyak aus [27] zurückgreifen. Wie bereits in Abschnitt 3.2 angegeben, lautet die Modifizierte Barrierefunktion für (QP)

$$\mathcal{F}(x, \lambda, k) = \frac{1}{2}x^T Q x + c^T x - \frac{1}{k}\sum_{j=1}^{m} \lambda_j \log(k \cdot (a_j^T x - b_j) + 1).$$

Leider benötigt der Beweis ähnlich wie im linearen Fall eine Nichtentartungsvoraussetzung.

Voraussetzung 2 (QP) *besitze eine Optimallösung x^* mit zugehörigem Lagrangevektor $\lambda^* \geq 0$. Es gelte*

$$\begin{aligned} r_j(x^*) = a_j^T x - b_j = 0, \quad \lambda_j^* > 0, \quad j = 1, \ldots, p \\ r_j(x^*) = a_j^T x - b_j > 0, \quad \lambda_j^* = 0, \quad j = p+1, \ldots, m \end{aligned}$$

und die hinreichenden Bedingungen zweiter Ordnung seien in x^* erfüllt, d.h. Q sei positiv definit auf dem Unterraum $Z = \{s : a_j^T s = 0, j = 1, \ldots, p\}$. Ferner seien die Vektoren $\{a_j, j = 1, \ldots, p\}$ linear unabhängig.

Die Voraussetzung 2 impliziert wiederum, daß x^* eindeutige Optimallösung von (QP) ist (vgl. z.B. [25, S. 316]). Darüberhinaus ist λ^* eindeutig bestimmter Lagrangevektor zu x^*. Um dies zu sehen, nehmen wir an, daß $\tilde{\lambda}$ ein weiterer optimaler Lagrangemultiplikator zu x^* sei. Wegen der Komplementarität (1.2c) folgt dann $\tilde{\lambda}_j = 0 = \lambda_j^*$ für $j = p+1, \ldots, m$. Nun verwenden wir die Karush–Kuhn–Tucker Bedingung (1.2a):

$$Qx^* + c = A^T \lambda^* = \sum_{j=1}^{m} \lambda_j^* a_j = \sum_{j=1}^{p} \lambda_j^* a_j \tag{3.21}$$

$$Qx^* + c = A^T \tilde{\lambda} = \sum_{j=1}^{m} \tilde{\lambda}_j a_j = \sum_{j=1}^{p} \tilde{\lambda}_j a_j \tag{3.22}$$

Subtrahiert man (3.22) von (3.21), so ergibt sich

$$\sum_{j=1}^{p} (\lambda_j^* - \tilde{\lambda}_j) \cdot a_j = 0,$$

und aus der linearen Unabhängigkeit der Vektoren $a_1, \ldots, a_p$ folgt dann $\lambda_j^* = \tilde{\lambda}_j$ für $j = 1, \ldots, p$. Insgesamt folgt also $\lambda^* = \tilde{\lambda}$, und dies zeigt, daß der Lagrangevektor λ^* zu x^* eindeutig bestimmt ist.

In völliger Analogie zum linearen Fall sei für $\delta > 0$, $\varepsilon > 0$ und $k_0 > 0$ die Menge $D(\lambda^*, k_0, \delta, \varepsilon)$ durch

$$D(\lambda^*, k_0, \delta, \varepsilon) = \left\{ (\lambda, k) \in \mathbb{R}_+^m \times \mathbb{R}_+ : \varepsilon \leq \lambda_j, j = 1, \ldots, p, \|\lambda - \lambda^*\|_\infty \leq k\delta, k \geq k_0 \right\}$$

definiert. Zur Vereinfachung der Darstellung und zur Verbesserung der Lesbarkeit (vor allem der Beweise) werden wir einige Notationen einführen.

Für einen Vektor $v \in \mathbb{R}^m$ sei $v_{(p)} = (v_1, \ldots, v_p)^T$, $v_{(m-p)} = (v_{p+1}, \ldots, v_m)^T$ und $V = \operatorname{diag}(v_1, \ldots, v_n)$. Ist $B = (b_{ij})_{1 \leq i,j \leq m}$ eine $m \times m$ Matrix, so sei $B_{(p)} = (b_{ij})_{1 \leq i,j \leq p}$ und $B_{(m-p)} = (b_{ij})_{p+1 \leq i,j \leq m}$. Wir partitionieren die Matrix A als $A = \binom{\bar{A}_p}{\bar{A}_{m-p}}$, wobei $\bar{A}_p^T = (a_1, \ldots, a_p)$ und $\bar{A}_{m-p}^T = (a_{p+1}, \ldots, a_m)$. Weiterhin sei die Diagonalmatrix $\Delta(x, k)$ durch (3.16) genauso wie im linearen Fall definiert. Analog zum linearen Fall werden wir unter der Voraussetzung 2

$$\sigma = \min \left(\min\{\lambda_j^*, j = 1, \ldots, p\}, \min\{a_j^T x^* - b_j, j = p+1, \ldots, m\} \right) > 0 \tag{3.23}$$

als das Maß an Nichtdegeneriertheit von (QP) bezeichnen. Wir werden sehen, daß σ ein entscheidender Faktor für die Größe der Konstanten ist, die beim Beweis des Analogons zu Satz 3.3.1 auftauchen. Wir zeigen nun zunächst das folgende Lemma:

Lemma 3.3.2 *Sei $(B_k)_k$ eine Folge von invertierbaren $\ell \times \ell$ Matrizen. Es gelte $\lim_{k\to\infty} B_k = B_\infty$ mit einer invertierbaren Matrix B_∞. Dann gilt $\mathrm{lub}_\infty(B_k^{-1}) \leq c$ für eine Konstante $c > 0$.*

Beweis: Wegen $\lim_{k\to\infty} B_k = B$ gilt $B_k = B + H_k$ mit $\lim_{k\to\infty} H_k = 0$. Wegen $\lim_{k\to\infty} H_k = 0$ können wir dann ein $K > 0$ finden, so daß für $k > K$ gilt: $\mathrm{lub}_\infty(BH_k) < \frac{1}{2}$. Daher erhalten wir mit Hilfe der Neumannschen Reihe

$$(I + BH_k)^{-1} = \sum_{\ell=0}^{\infty} (BH_k)^\ell$$

für $k > K$ die folgende Ungleichung:

$$\mathrm{lub}_\infty(I + BH_k)^{-1} \leq \frac{1}{1 - \mathrm{lub}_\infty(BH_k)} \leq \frac{1}{1 - \frac{1}{2}} = 2. \tag{3.24}$$

Schreiben wir B_k^{-1} als

$$B_k^{-1} = (B + H_k)^{-1} = B^{-1}(I + BH_k)^{-1}. \tag{3.25}$$

so ergibt sich aus (3.24) und (3.25) für $k > K$ die Abschätzung:

$$\mathrm{lub}(B_k^{-1}) \leq 2 \cdot \mathrm{lub}_\infty(B^{-1}).$$

Setzen wir $c' := \max_{k \leq K} \mathrm{lub}_\infty(B_k^{-1})$, so folgt die Behauptung des Lemmas mit $c := \max\{c', 2\mathrm{lub}_\infty(B^{-1})\}$. $\square$

Nun zeigen wir das Analogon zu Satz 3.3.1 für das quadratische Programm (QP). Der Beweis entspricht im wesentlichen dem Beweis von [27, Theorem 1]:

Satz 3.3.3 *Für das quadratische Programm (QP) sei die Voraussetzung 2 erfüllt. Dann gibt es $k_0 > 0$ und $\delta > 0$, so daß für alle $0 < \varepsilon < \min\{\lambda_j^* : j = 1, \ldots, p\}$ und $(\lambda, k) \in D(\lambda^*, k_0, \delta, \varepsilon)$ folgende Aussagen richtig sind:*

(i) *Es existiert $\hat{x}(\lambda, k) \in \arg\ \min\{\mathcal{F}(x, \lambda, k) : x \in \mathbb{R}^n\}$ mit $\nabla_x \mathcal{F}(\hat{x}(\lambda, k), \lambda, k)) = 0$.*

(ii) *Sei $\hat{\lambda}(\lambda, k) = \Delta(\hat{x}(\lambda, k), k)^{-1}\lambda$. Dann gilt die Ungleichung*

$$\max\{\|\hat{x}(\lambda, k) - x^*\|_\infty, \|\hat{\lambda}(\lambda, k) - \lambda^*\|_\infty\} \leq \frac{c_1}{k}\|\lambda - \lambda^*\|_\infty \tag{3.26}$$

mit einer Konstanten c_1, die nicht von $k \geq k_0$ abhängt.

(iii) *Es gilt* $(\hat{\lambda}(\lambda, k), k) \in D(\lambda^*, k_0, \delta, \varepsilon)$.

Beweis: Sei $u = (u_1, \ldots, u_m)^T \in \mathbb{R}^m$. Für $\delta > 0$ und $\varepsilon_0 > 0$ definieren wir die Funktion $h : \bar{B}(x^*, \varepsilon_0) \times \bar{B}(\lambda^*_{(p)}, \varepsilon_0) \times \bar{B}(0, \delta) \to \mathbb{R}^n$ durch

$$h(x, u, k) = k \sum_{j=p+1}^{m} \frac{u_j}{k(a_j^T x - b_k) + 1} \cdot a_j = k \sum_{j=p+1}^{m} \frac{u_j}{k r_j(x) + 1} \cdot a_j.$$

Offenbar gilt für alle $k > 0$, daß $h(x^*, 0, k) = 0$. Ferner ist wegen

$$D_x h(x, u, k) = k^2 \,(a_{p+1}, \ldots, a_m) \cdot \Delta^2_{(m-p)}(x, k) \cdot \begin{pmatrix} u_{p+1} & & 0 \\ & \ddots & \\ 0 & & u_m \end{pmatrix} \cdot \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix}$$

auch $D_x h(x^*, 0, k) = 0$ für alle $k > 0$. Wir betrachten des weiteren nun die Funktion $\Phi : \mathbb{R}^{n+p+m+1} \to \mathbb{R}^{n+p}$, definiert durch

$$\begin{aligned} \Phi(x, \lambda_1, \ldots, \lambda_p, u, k) &= \begin{pmatrix} Qx + c - \sum\limits_{j=1}^{p} \lambda_j a_j - h(x, u, k) \\ k^{-1} \Delta_{(p)}(x, k)^{-1} (k \cdot u_{(p)} + \lambda^*_{(p)}) - k^{-1} \lambda_{(p)} \end{pmatrix} \\ &= \begin{pmatrix} \varphi_1(x, \lambda_1, \ldots, \lambda_p, u, k) \\ \varphi_2(x, \lambda_1, \ldots, \lambda_p, u, k) \end{pmatrix}, \end{aligned}$$

auf $\bar{B}(x^*, \varepsilon_0) \times \bar{B}(\lambda_{(p)}, \varepsilon_0) \times \bar{B}(0, \delta) \times (0, +\infty)$. Zunächst zeigen wir, daß für alle k die Gleichung

$$\Phi(x^*, \lambda^*_{(p)}, 0, k) = 0$$

erfüllt ist. In der Tat, da das Paar x^*, λ^* die Karush–Kuhn–Tucker Bedingungen (1.2a) bis (1.2c) erfüllt, gilt insbesondere

$$\begin{aligned} 0 = Qx^* + c - \sum_{j=1}^{m} \lambda_j^* a_j &= Qx^* + c - \sum_{j=1}^{p} \lambda_j^* a_j - \sum_{j=p+1}^{m} \overbrace{\lambda_j^*}^{=0} a_j \\ &= Qx^* + c - \sum_{j=1}^{p} \lambda_j^* a_j \end{aligned}$$

Wie oben bemerkt, ist $h(x^*, 0, k) = 0$, und damit hat man:

$$\varphi_1(x^*, \lambda^*_{(p)}, 0, k) = Qx^* + c - \sum_{j=1}^{p} \lambda_j^* a_j - h(x^*, 0, k) = 0. \tag{3.27}$$

Weiterhin ist $r_j(x^*) = 0$ für $j = 1, \ldots, p$, also

$$\Delta_{(p)}(x^*, k) = \operatorname{diag}(k r_1(x^*) + 1, \ldots, k r_p(x^*) + 1) = I^{p,p}.$$

Daraus ergibt sich sofort, daß

$$\varphi_2(x^*, \lambda^*_{(p)}, 0, k) = k^{-1}\Delta_{(p)}(x^*, k)^{-1}\lambda^*_{(p)} - k^{-1}\lambda^*_{(p)} = k^{-1}I^{p,p}\lambda^*_{(p)} - k^{-1}\lambda^*_{(p)} = 0. \quad (3.28)$$

Nimmt man nun (3.27) und (3.28) zusammen, so erhält man das gewünschte Ergebnis: $\Phi(x^*, \lambda^*_{(p)}, 0, k) = 0$. Wir berechnen $B_k = D_{x,\lambda_{(p)}}\mathcal{F}(x^*, \lambda_{(p)}, 0, k)$. Es gilt zunächst

$$\begin{aligned} D_{x,\lambda_{(p)}}\varphi_1(x, \lambda_{(p)}, u, k) &= (Q - D_x h(x, u, k); -a_1, -a_2, \ldots, -a_p) \\ &= \left(Q - D_x h(x, u, k); -\bar{A}_p^T\right). \end{aligned}$$

Nutzen wir $D_x h(x^*, 0, k) = 0$ aus, so sehen wir, daß

$$D_{x,\lambda_{(p)}}\varphi_1(x^*, \lambda^*_{(p)}, 0, k) = (Q; -\bar{A}_p) \quad (3.29)$$

gilt. Weiterhin ist

$$D_{x,\lambda_{(p)}}\varphi_2(x, \lambda_{(p)}, u, k) = \left(-\Delta_{(p)}(x, k)^{-2}(kU_{(p)} + \Lambda_{(p)})\bar{A}_p; -k^{-1}I^{p,p}\right).$$

Aus $\Delta_{(p)}(x^*, k) = I^{p,p}$ folgt daher:

$$D_{x,\lambda_{(p)}}\varphi_2(x^*, \lambda^*_{(p)}, 0, k) = (-\Lambda^*_{(p)}\bar{A}_p; -k^{-1}I^{p,p}). \quad (3.30)$$

Damit ergibt sich aus (3.29) und (3.30)

$$\begin{aligned} B_k &= D_{x,\lambda_{(p)}}\Phi(x^*, \lambda^*_{(p)}, 0, k) \\ &= \left(\begin{array}{c|c} Q & -\bar{A}_p^T \\ \hline -\Lambda^*_{(p)}\bar{A}_p & -k^{-1}I^{p,p} \end{array}\right). \\ &= \left(\begin{array}{c|cccc} Q & -a_1 & -a_2 & \cdots & -a_p \\ \hline -\lambda_1^* a_1^T & & & & \\ -\lambda_2^* a_2^T & & -k^{-1}I^{p,p} & & \\ \vdots & & & & \\ -\lambda_p^* a_p^T & & & & \end{array}\right) \end{aligned}$$

Wir wollen das implizite Funktionentheorem anwenden. Dazu müssen wir zunächst zeigen, daß B_k nichtsingulär ist. Sei $B_k\binom{v}{w} = 0$ mit $v \in \mathbb{R}^n$ und $w \in \mathbb{R}^p$. Dann folgt

$$Qv - \sum_{j=1}^{p} w_j a_j = 0 \quad (3.31)$$

und

$$-\lambda_j^* a_j^T v - k^{-1} w_j = 0 \quad \text{für } j = 1, \ldots, p. \tag{3.32}$$

Wir multiplizieren die Gleichung (3.31) mit v^T und erhalten somit

$$v^T Q v - \sum_{j=1}^{p} w_j v^T a_j = 0. \tag{3.33}$$

Aus (3.32) ergibt sich $w_j = -k\lambda_j^* a_j^T v$, $j = 1, \ldots, p$. Wenn wir dies in die Gleichung (3.33) einsetzen, folgt

$$v^T Q v + k \sum_{j=1}^{p} \lambda_j^* (a_j^T v)^2 = 0. \tag{3.34}$$

Nun ist Q positiv semidefinit und $\lambda_j^*(a_j^T v)^2 \geq 0$ für $j = 1, \ldots, p$. Daher zeigt (3.34), daß $v^T Q v = 0$ und wegen $\lambda_{(p)}^* > 0$ darüberhinaus auch $a_j^T v = 0$ für $j = 1, \ldots, p$ gelten muß. Die letzte Aussage ist dabei nach Definition des Unterraumes Z äquivalent zu $v \in Z$. Nach Voraussetzung ist Q positiv definit auf Z. Wie eben gesehen, gilt sowohl $v \in Z$ als auch $v^T Q v = 0$ und daher folgt sofort $v = 0$.

Aus $w_j = -k\lambda_j^* a_j^T v$, $j = 1, \ldots, p$ ergibt sich dann auch $w = 0$. Somit ist $\binom{v}{w} = \binom{0}{0}$ und dies zeigt die Nichtsingularität von B_k.

Sei $K := \{0 \in \mathbb{R}^m\} \times [k_0, k_1]$, wobei $k_1 > k_0$ beliebig groß, aber fest gewählt sei. Dann ist K kompakt. Wir haben gesehen, daß für jedes $k \in [k_0, k_1]$ die Matrix B_k nichtsingulär ist. Mit Hilfe des zweiten impliziten Funktionentheorems (siehe etwa [2, S. 12]) können wir daher wie folgt schließen: Es existieren $\varepsilon_0 > 0$, $\delta > 0$ und C^1–Funktionen $x(u,k), \lambda_{(p)}(u,k)$, die eindeutig in einer Umgebung $M(K,\delta) = \{(u,k) : \|u\|_\infty \leq \delta, k \in [k_0, k_1]\}$ von K definiert sind, so daß

$$\begin{aligned}
x(0,k) &= x^*, && (3.35a)\\
\lambda_{(p)}(0,k) &= \lambda^*, && (3.35b)\\
\Phi(x(u,k), \lambda_{(u)}(u,k), u, k) &= 0, && (3.35c)\\
\|x(u,k) - x^*\|_\infty &\leq \varepsilon_0, && (3.35d)\\
\|\lambda_{(p)}(u,k) - \lambda_{(p)}^*\|_\infty &\leq \varepsilon_0 && (3.35e)
\end{aligned}$$

für *alle* $(u,k) \in M(u,\delta)$ gilt. Sind $\delta > 0$ und $\varepsilon_0 > 0$ ausreichend klein, so gilt zudem

$$\lambda_{(p)}(u,k) \geq \varepsilon \quad \text{für } (u,k) \in M(K,\delta). \tag{3.35f}$$

Wir definieren nun die Vektorfunktion $\lambda_{(m-p)}(u,k) = \Delta_{(m-p)}(x(u,k))^{-1} \lambda_{(m-p)}$ und ferner

$$\lambda(u,k) = \begin{pmatrix} \lambda_{(p)}(u,k) \\ \lambda_{(m-p)}(u,k) \end{pmatrix}. \tag{3.36}$$

Man beachte, daß $\lambda_{(p)}(u,k)$ durch das implizite Funktionentheorem geliefert wird. Wir werden sehen, daß die Definition von $\lambda_{(m-p)}(u,k)$ natürlich ist. Sei

$$u = \frac{\lambda - \lambda^*}{k},$$

dann besagt unser bisheriges Ergebnis, daß es für $\|\lambda - \lambda^*\|_\infty = k\|u\|_\infty \leq k\delta$ eindeutige C^1–Funktionen $x(\frac{\lambda-\lambda^*}{k}, k)$, $\lambda_{(p)}(\frac{\lambda-\lambda^*}{k}, k)$ gibt, so daß (3.35a) bis (3.35e) erfüllt sind. Wir schreiben nun die Gleichung (3.35c) komponentenweise unter Benutzung von $\Phi = \binom{\varphi_1}{\varphi_2}$ aus. Wir erhalten

$$\begin{aligned}
0 &= \varphi_1(x(u,k), \lambda_{(p)}(u,k), u, k) \\
&= Qx(u,k) + c - \sum_{j=1}^{p} \lambda_j(u,k) a_j - h(x(u,k), u, k) && (3.37) \\
&= Qx(u,k) + c - \sum_{j=1}^{p} \lambda_j(u,k) a_j - \sum_{j=p+1}^{m} (\lambda_j - \overbrace{\lambda_j^*}^{=0}) \frac{a_j}{k(a_j^T x(u,k) - b_k) + 1} \\
&= Qx(u,k) + c - \sum_{j=1}^{p} \lambda_j(u,k) a_j - \sum_{j=p+1}^{m} \frac{\lambda_j}{k(a_j^T x(u,k) - b_k) + 1} \cdot a_j && (3.38)
\end{aligned}$$

sowie

$$\begin{aligned}
0 &= \varphi_2(x(u,k), \lambda_{(p)}(u,k), u, k) \\
&= k^{-1}\Delta_{(p)}(x(u,k),k)^{-1}(k \cdot u_{(p)} + \lambda^*_{(p)}) - k^{-1}\lambda_{(p)}(u,k) && (3.39) \\
&= k^{-1}\Delta_{(p)}(x(u,k),k)^{-1}(k \cdot (\lambda_{(p)} - \lambda^*_{(p)})/k + \lambda^*_{(p)}) - k^{-1}\lambda_{(p)}(u,k) \\
&= k^{-1}\Delta_{(p)}(x(u,k),k)^{-1}\lambda_{(p)} - k^{-1}\lambda_{(p)}(u,k). && (3.40)
\end{aligned}$$

Aus (3.40) folgt nun sofort, daß

$$\lambda_{(p)}(u,k) = \Delta_{(p)}(x(u,k),k)^{-1}\lambda_{(p)}. \tag{3.41}$$

Setzt man dieses Ergebnis in (3.38) ein, so ergibt sich

$$Qx(u,k) + c - \sum_{j=1}^{p} \frac{\lambda_j}{k(a_j^T x(u,k) - b_j) + 1} \cdot a_j - \sum_{j=p+1}^{m} \frac{\lambda_j}{k(a_j^T x(u,k) - b_j) + 1} \cdot a_j = 0. \tag{3.42}$$

Im Hinblick auf (3.8) ist dies nichts anderes als

$$\nabla_x \mathcal{F}(x(u,k), \lambda, k) = 0. \tag{3.43}$$

Mit $\hat{x}(\lambda,k) := x(\frac{\lambda-\lambda^*}{k}, k)$ folgt nun die erste Aussage des Satzes. In der Tat ist wegen (3.43)

$$\hat{x}(\lambda,k) \in \arg\min\{\mathcal{F}(x,\lambda,k) : x \in \mathbb{R}^n\}.$$

Die Gleichung (3.41) zeigt darüberhinaus, daß, wie angekündigt, die Definition von $\lambda_{(m-p)}$ natürlich ist. Damit läßt sich (3.36) auch als

$$\lambda(u,k) = \Delta(x(u,k),k)^{-1}\lambda \tag{3.44}$$

schreiben. Wir wollen nun $\|\lambda(u,k)-\lambda^*\|_\infty$ und $\|x(u,k)-x^*\|_\infty$ abschätzen. Es ist

$$\|\lambda(u,k)-\lambda^*\|_\infty = \max\left\{\|\lambda_{(p)}(u,k)-\lambda^*_{(p)}\|_\infty, \|\lambda_{(m-p)}(u,k)-\lambda^*_{(m-p)}\|_\infty\right\},$$

und daher schätzen wir zunächst $\|\lambda_{(m-p)}(u,k)-\lambda^*_{(m-p)}\|_\infty$ ab. Ist $\delta > 0$ klein genug, so folgt wegen $x(0,k) = x^*$ und der Stetigkeit der affinen Funktionen $r_j(x) = a_j^T x - b_j, j = p+1,\ldots,m$, daß für alle $\|u\|_\infty \le \delta$ gilt:

$$r_j(x(u,k)) \ge \frac{1}{2} r_j(x(0,k)) = \frac{1}{2}(a_j^T x^* - b_j) \ge \frac{\sigma}{2}. \tag{3.45}$$

Daher gilt dann für $j = p+1,\ldots,m$, daß

$$\begin{aligned}
\left|\lambda_j(u,k)-\lambda_j^*\right| &= \left|\frac{\lambda_j}{k\cdot(a_j^T x(u,k)-b_j)+1} - \overbrace{\lambda_j^*}^{=0}\right| = \left|\lambda_j \cdot \frac{1}{k\cdot(a_j^T x(u,k)-b_j)+1}\right| \\
&\le \left|\frac{\lambda_j}{k(\frac{\sigma}{2}+1)}\right| \\
&\le \frac{2}{\sigma}\cdot\lambda_j\frac{1}{k} \\
&= \frac{1}{k}\cdot\frac{2}{\sigma}\cdot\left|\lambda_j-\lambda_j^*\right|.
\end{aligned}$$

Folglich ist daher

$$\|\lambda_{(m-p)}(u,k)-\lambda^*_{(m-p)}\|_\infty \le \frac{1}{k}\cdot\frac{2}{\sigma}\cdot\|\lambda_{(m-p)}-\lambda^*_{(m-p)}\|_\infty. \tag{3.46}$$

Um eine Abschätzung für $\|\lambda_{(p)}(u,k)-\lambda_{(p)}\|_\infty$ und $\|x(u,k)-x^*\|_\infty$ herzuleiten, bemerken wir zunächst, daß nach der Kettenregel gilt:

$$\begin{aligned}
\begin{pmatrix} x(u,k) & - & x^* \\ \lambda_{(p)}(u,k) & - & \lambda^*_{(p)} \end{pmatrix} &= \begin{pmatrix} x(u,k) & - & x(0,k) \\ \lambda_{(p)}(u,k) & - & \lambda_{(p)}(0,k) \end{pmatrix} \\
&= \int_0^1 \begin{pmatrix} D_u x(\tau u,k) \\ D_u\lambda_{(p)}(\tau u,k) \end{pmatrix} u\, d\tau.
\end{aligned} \tag{3.47}$$

Haben wir daher einmal gezeigt, daß für alle $k \ge k_0$ und $\|u\| \le \delta$ die Abschätzung

$$\max\left\{\mathrm{lub}_\infty(D_u x(u,k)), \mathrm{lub}_\infty(D_u\lambda_{(p)}(u,k))\right\} \le \omega \tag{3.48}$$

für eine von k unabhängige Konstante ω gilt, dann folgt auch aus (3.47), daß die Ungleichung

$$\max\{\|x(u,k)-x^*\|_\infty, \|\lambda_{(p)}(u,k)-\lambda^*_{(p)}\|_\infty\} \le \omega\|u\|_\infty = \frac{\omega}{k}\|\lambda-\lambda^*\|_\infty \tag{3.49}$$

erfüllt ist. Im Hinblick auf (3.46) folgt dann mit $c_1 := \max\{2/\sigma, \omega\}$ sofort:

$$\max\{\|x(u,k)-x^*\|_\infty, \|\lambda(u,k)-\lambda^*\|_\infty\} \le \frac{c_1}{k}\|\lambda-\lambda^*\|_\infty,$$

bzw. mit $u = \frac{\lambda-\lambda^*}{k}$, $\hat{x}(\lambda,k) = x(\frac{\lambda-\lambda^*}{k},k)$ und $\hat{\lambda}(\lambda,k) = \lambda(\frac{\lambda-\lambda^*}{k},k)$

$$\max\{\|\hat{x}(\lambda,k)-x^*\|_\infty, \|\lambda(\lambda,k)-\lambda^*\|_\infty\} \le \frac{c_1}{k}\|\lambda-\lambda^*\|_\infty. \tag{3.50}$$

und damit die Aussage (ii) des Satzes. Es verbleibt also, eine Ungleichung der Form (3.48) herzuleiten. Dazu differenzieren wir (3.37) und (3.39) nach u. Aus (3.37) erhalten wir dann

$$Q\cdot D_u x(u,k) - \begin{pmatrix} a_1 & a_2 & \cdots & a_p \end{pmatrix}\cdot D_u\lambda_{(p)}(u,k) = \frac{d}{du}\left[h(x(u,k),u,k)\right], \tag{3.51}$$

während sich aus (3.39)

$$\begin{aligned} &- \begin{pmatrix} ku_1+\lambda_1^* & & 0 \\ & \ddots & \\ 0 & & ku_p+\lambda_p^* \end{pmatrix} \cdot \Delta_{(p)}(x(u,k),k)^{-2} \cdot \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} \cdot D_u x(u,k) - k^{-1} D_u\lambda_{(p)}(u,k) \\ = \; & -\Delta_{(p)}(x(u,k),k)^{-1}\cdot(I^{p,p};0^{p,m-p}) = -\left(\Delta_{(p)}(x(u,k),k)^{-1};0^{p,m-p}\right) \end{aligned}$$

ergibt. Aus der letzten Gleichung ergibt sich durch Multiplikation aller Seiten mit der nichtsingulären Matrix $\Delta_{(p)}(x(u,k),k)^2$ von links

$$\begin{aligned} &-\left(k\cdot U_{(p)}+\Lambda^*_{(p)}\right)\cdot \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} \cdot D_u x(u,k) - k^{-1}\Delta_{(p)}(x(u,k),k)^2\cdot D_u\lambda_{(p)}(u,k) \\ = \; & -\left(\Delta_{(p)}(x(u,k),k);0^{p,m-p}\right), \end{aligned} \tag{3.52}$$

da die auftretenden Diagonalmatrizen kommutieren. Aus den Gleichungen (3.51) und (3.52), folgt, daß $D_u x(u,k)$ und $D_u\lambda_{(p)}(u,k)$ Lösung des folgenden linearen Gleichungssystems sind:

$$\left(\begin{array}{c|c} Q & -a_1 \quad -a_2 \quad \cdots \quad -a_p \\ \hline -\left(k \cdot U_{(p)} + \Lambda^*_{(p)}\right) \cdot \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} & -k^{-1}\Delta_{(p)}(x(u,k),k)^2 \end{array}\right) \cdot \begin{pmatrix} D_u x(u,k) \\ D_u \lambda_{(p)}(u,k) \end{pmatrix}$$

$$= \begin{pmatrix} \frac{d}{du}\left[h(x(u,k),u,k)\right] \\ -\left(\Delta_{(p)}(x(u,k),k); 0^{p,m-p}\right) \end{pmatrix}.$$

Mit den Abkürzungen

$$B(u,k) = \left(\begin{array}{c|c} Q & -a_1 \quad -a_2 \quad \cdots \quad -a_p \\ \hline -\left(k \cdot U_{(p)} + \Lambda^*_{(p)}\right) \cdot \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} & -k^{-1}\Delta_{(p)}(x(u,k),k)^2 \end{array}\right),$$

$$N(u,k) = \begin{pmatrix} \frac{d}{du}\left[h(x(u,k),u,k)\right] \\ -\left(\Delta_{(p)}(x(u,k),k); 0^{r,m-r}\right) \end{pmatrix}$$

schreibt sich dieses Gleichungssystem dann als

$$B(u,k) \cdot \begin{pmatrix} D_u x(u,k) \\ D_u \lambda_{(p)}(u,k) \end{pmatrix} = N(u,k). \tag{3.53}$$

Wir sehen, daß sich die Matrix $B(u,k)$ aus (3.53) für $u = 0 \in \mathrm{I\!R}^m$ auf die Matrix $B_k = D_{x,\lambda_{(p)}}\Phi(x^*, \lambda_{(p)}, 0, k)$ reduziert. Insbesondere ist $B(0,k) = B_k$ also nichtsingulär. Wir wollen nun als nächstes zeigen, daß $\mathrm{lub}_\infty(B_k^{-1}) \le c$ für eine Konstante $c > 0$ und *alle* $k \geq k_0$ gilt. Dazu betrachten wir $B_\infty = \lim_{k\to\infty} B_k$. Es gilt

$$B_\infty = \lim_{k\to\infty} B_k = \left(\begin{array}{c|c} Q & -a_1 \quad -a_2 \quad \cdots \quad -a_p \\ \hline \begin{array}{c} -\lambda_1^* a_1^T \\ -\lambda_2^* a_2^T \\ \vdots \\ -\lambda_p^* a_p^T \end{array} & 0 \end{array}\right).$$

Wir behaupten, daß auch B_∞ nichtsingulär ist. Der Beweis dieser Tatsache verläuft analog zum Beweis der Nichtsingularität von B_k weiter oben im Beweis. Ist nämlich $B_\infty\binom{v}{w} = 0$, so folgt wieder

$$Qv - \sum_{j=1}^{p} w_j a_j = 0 \tag{3.54}$$

und

$$\lambda_j^* a_j^T v = 0 \text{ für } j = 1, \ldots, p. \tag{3.55}$$

Multipliziert man (3.54) mit v^T und nutzt aus, daß wegen $\lambda_j^* > 0$ und (3.55) auch $a_j^T v = 0$ für $j = 1, \ldots, p$ gelten muß, so hat man $v^T Q v = 0$.

Da Q nach Voraussetzung positiv definit auf $Z = \{s : a_j^T s = 0, j = 1, \ldots, p\}$ ist, folgt daher $v = 0$. Einsetzen dieses Ergebnisses in (3.54) ergibt dann

$$\sum_{j=1}^{p} w_j a_j = 0,$$

und die lineare Unabhängigkeit der Vektoren $\{a_j, j = 1, \ldots, p\}$ zeigt nun, daß $w = 0$ gelten muß. Somit ist gezeigt, daß $B_\infty\binom{v}{w} = 0$ stets $\binom{v}{w} = \binom{0}{0}$ impliziert. Folglich ist B_∞ nichtsingulär.

Lemma 3.3.2 lehrt uns nun, daß es ein $c > 0$ mit $\mathrm{lub}_\infty(B_k^{-1}) \leq c$ für alle $k \geq k_0$ gibt. Da die Matrix $B(u,k)$ stetig von $u \in \mathbb{R}^m$ abhängt, ist folgender Schluß möglich:

Es existiert $\delta > 0$, so daß für $\|u\|_\infty = k\|\lambda - \lambda^*\|_\infty \leq \delta$ und $\lambda_{(p)} \geq \varepsilon$ nicht nur $B(u,k)$ invertierbar, sondern darüberhinaus die Norm der Inversen für alle $k \geq k_0$ durch das Doppelte der Konstanten c abschätzbar, d.h. die Ungleichung

$$\mathrm{lub}_\infty(B(u,k)^{-1}) \leq 2 \cdot c. \tag{3.56}$$

richtig ist.

Als nächstes benötigen wir eine Abschätzung der Norm $\mathrm{lub}_\infty(N(u,k))$. Zu diesem Zweck berechnen wir zuerst $\frac{d}{du}[h(x(u,k),u,k)]$. Es gilt

$$\begin{aligned}
\tfrac{d}{du}[h(x(u,k),u,k)] &= \tfrac{d}{du}\left\{k\sum_{j=p+1}^{m}\tfrac{u_j}{k(a_j^Tx(u,k)-b_k)+1}\cdot a_j\right\}\\
= \;& k\,(a_{p+1},\dots,a_m)\cdot\left(0^{m-p,p} \quad ; \quad \Delta_{(m-p)}(x(u,k),k)^{-1}\right)\\
& -k^2\,(a_{p+1},\dots,a_m)\cdot U_{(m-p)}\cdot\Delta_{(m-p)}(x(u,k),k)^{-2}\cdot\begin{pmatrix} a_{p+1}^T\\ \vdots\\ a_m^T\end{pmatrix}\cdot D_ux(u,k).
\end{aligned}$$

Betrachten wir zunächst wieder $N(u,k)$ für $u=0$. Dann gilt wegen $x(0,k)=x^*$, $\Delta_{(p)}(x^*,k)=I^{p,p}$ und $U_{(m-p)}=0$ (da $u=0$):

$$N(0,k)=\begin{pmatrix}(a_{p+1},\dots,a_m)\cdot\left(0^{m-p,p} \quad ; \quad k\Delta_{(m-p)}(x^*,k)^{-1}\right)\\ \left(-I^{p,p} \quad ; \quad 0^{p,m-p}\right)\end{pmatrix}=\begin{pmatrix}N_1(k)\\ N_2(k)\end{pmatrix}.$$

Die lub–Norm zur Vektornorm $\|.\|_\infty$ ist die Maximumsspaltensummennorm (vgl. Seite 10). Daher gilt $\mathrm{lub}_\infty(N(0,k))\le\mathrm{lub}_\infty(N_1(k))+\mathrm{lub}_\infty(N_2(k))$. Wir schätzen die Normen der beiden Matrizen $N_1(k)$ und $N_2(k)$ ab. Dies ist für $N_2(k)$ trivial: $\mathrm{lub}_\infty(N_2(k))=1$ für alle $k>0$. Kaum aufwendiger ist die Abschätzung für $N_1(k)$. Wegen $a_j^Tx^*-b_j\ge\sigma$ für $j=p+1,\dots,m$ ist nämlich für diese j

$$\frac{k}{k(a_j^Tx^*-b_j)+1}\le\frac{k}{k\sigma+1}=\frac{1}{\sigma+1/k}<\sigma^{-1}.$$

Deswegen ist jeder Eintrag der Diagonalmatrix $k\Delta_{(m-p)}(x^*,k)^{-1}$ vom Betrag her durch σ^{-1} beschränkt. Konsequenterweise ist daher die Maximumsspaltensummennorm von $\left(0^{m-p,p} \quad ; \quad k\Delta_{(m-p)}(x^*,k)^{-1}\right)$ durch σ^{-1} beschränkt. Wegen der Submultiplikativität der lub–Norm folgt nun

$$\mathrm{lub}_\infty(N_2(k))\le\sigma^{-1}\mathrm{lub}_\infty((a_{p+1},\dots,a_m))$$

und schließlich

$$\mathrm{lub}_\infty(N(0,k))\le\mathrm{lub}_\infty(N_1(k))+\mathrm{lub}_\infty(N_2(k))\le 1+\sigma^{-1}\mathrm{lub}_\infty((a_{p+1},\dots,a_m)). \quad (3.57)$$

Wiederum hängt $N(u,k)$ stetig von u ab. Folglich impliziert daher $\|u\|_\infty\le\delta$ für genügend kleines $\delta>0$, daß sich die Norm von $N(u,k)$ durch das Zweifache des Wertes der rechten Seite in (3.57) beschränken läßt, wobei diese Abschätzung für alle $k\ge k_0$

ihre Richtigkeit behält. Aus (3.53) folgt nun mit den letzten Abschätzungen

$$\begin{aligned}
\max\left\{\mathrm{lub}_\infty(D_u x(u,k)), \mathrm{lub}_\infty(D_u\lambda_{(p)}(u,k))\right\} &\leq \mathrm{lub}_\infty\begin{pmatrix} D_u x(u,k) \\ D_u\lambda_{(p)}(u,k)\end{pmatrix} \\
&\leq \mathrm{lub}_\infty(B(u,k)^{-1})\cdot \mathrm{lub}_\infty(N(u,k)) \\
&\leq 4c\cdot(1+\sigma^{-1}\mathrm{lub}_\infty((a_{p+1},\ldots,a_m))) =: \omega
\end{aligned}$$

für alle $\|u\|_\infty \leq \delta$ und alle $k \geq k_0$.

Die Aussage in (iii) folgt nun unmittelbar aus (ii) und (3.35f).

Damit ist die Abschätzung (3.48) bewiesen und der Beweis beendet. □

3.4 Barrierefunktionen mit Shift

In diesem Abschnitt betrachten wir einen Spezialfall der Modifizierten Barrierefunktion $\mathcal{F}(x,\lambda,k)$ für (QP), nämlich

$$\begin{aligned}
\mathcal{M}(x,k) &= \mathcal{F}(x,e,k) \\
&= \begin{cases} \frac{1}{2}x^TQx + c^Tx - \frac{1}{k}\sum\limits_{j=1}^{m}\log(k(a_j^Tx-b_j)+1) & \text{, falls } x\in \mathrm{int}\ S_k \\ +\infty & \text{, sonst.}\end{cases} \qquad (3.58)
\end{aligned}$$

Die Funktion $\mathcal{M}(.,k)$ werden wir als Barrierefunktion mit Shift bezeichnen.

Für das lineare Programm (LP) zeigt Polyak in [28, Assertion 5] den folgenden

Satz 3.4.1 *Sei für das Paar* (LP), (LPD) *von dualen linearen Programmen die Voraussetzung 1 erfüllt und die Menge S der zulässigen Lösungen von* (LP) *beschränkt. Dann gilt*

(i) *Für jedes $k > 0$ existiert ein eindeutiges $x(k) = \arg\ \min\{\mathcal{M}(x,k) : x \in \mathbb{R}^n\}$ und es gilt*

$$\nabla_x\mathcal{M}(x(k),k)) = 0.$$

(ii) *Es existiert ein $k_0 > 0$, so daß für das Paar $x(k)$ und*

$$\lambda(k) = \Delta(x(k),k)^{-1}e = \mathrm{diag}(k(a_1^Tx(k)-b_1)+1,\ldots,k(a_m^Tx(k)-b_m)+1)^{-1}e$$

die Abschätzung

$$\max\{\|x(k)-x^*\|_\infty, \|\lambda(k)-\lambda^*\|_\infty\} \leq \frac{c_3}{k}$$

gilt, wobei c_3 eine Konstante ist, die nicht von $k \geq k_0$ abhängt.

Wiederum möchten wir ein Analogon für den quadratischen Fall beweisen. Wir notieren im folgenden Satz zunächst die Hauptresultate über die Barrierefunktionen mit Shift für (QP) und gehen dann auf die Bedeutung dieser Funktionen für die Modifizierte Barrieremethode ein.

Satz 3.4.2 *Falls S nichtleer und beschränkt, sowie die Voraussetzung 2 erfüllt ist, dann gelten folgende Aussagen für die Barrierefunktion mit Shift $\mathcal{M}(.,k)$ aus (3.58):*

(i) *$\mathcal{M}(.,k)$ ist strikt konvex in S_k.*

(ii) *Für alle $k > 0$ existiert ein eindeutiges $x(k) \in \operatorname{int} S_k$ mit*

$$x(k) = \arg\ \min\{\mathcal{M}(x,k) : x \in S_k\}.$$

Es gilt $\nabla_x \mathcal{M}(x(k),k) = 0$. Ferner hängt $x(k)$ stetig von k ab.

(iii) *Für $k > 0$ sei $\lambda(k) \in \mathrm{I\!R}^m$ der Vektor, der durch*

$$\lambda(k) = \Delta(x(k),k)^{-1}e = \operatorname{diag}(k(a_1^T x(k) - b_1) + 1, \ldots, k(a_m^T x(k) - b_m) + 1)^{-1}e$$

definiert ist. Dann gilt

$$\lim_{k\to\infty} x(k) = x^* \quad \text{und} \quad \lim_{k\to\infty} \lambda(k) = \lambda^*,$$

sowie

$$\lim_{k\to\infty} q(x(k)) = \lim_{k\to\infty} \mathcal{M}(x(k),k) = q(x^*).$$

(iv) *Für das Paar $x(k), \lambda(k)$ gilt*

$$\max\{\|x(k) - x^*\|_\infty, \|\lambda(k) - \lambda^*\|_\infty\} \le \frac{c_4}{k} \tag{3.59}$$

mit einer Konstanten c_4, die nicht von $k \ge k_0$ abhängt.

Bevor wir Satz 3.4.2 beweisen, möchten wir hier auf die Bedeutung der darin genannten Ergebnisse im Zusammenhang mit Satz 3.3.3 hinweisen. Für die Ergebnisse aus Satz 3.3.3 wird ein Vektor λ mit $(\lambda, k) \in D(\lambda^*, k_0, \delta, \varepsilon)$ benötigt, also eine genügend gute Schätzung für den optimalen Lagrangevektor λ^*. In der Regel ist eine solche Schätzung nicht vorhanden. Die Ergebnisse in Satz 3.4.2 ermöglichen es uns aber, einen Vektor λ mit den gewünschten Eigenschaften zu finden.

Wir beweisen nun zunächst ein nützliches Konvergenzlemma:

Lemma 3.4.3 *Sei $(v^\nu)_\nu$ eine beschränkte Folge im $\mathrm{I\!R}^d$. Es gebe ein $v^* \in \mathrm{I\!R}^d$, so daß $\lim_{\ell\to\infty} v^{\nu_\ell} = v^*$ für jede* konvergente *Teilfolge $(v^{\nu_\ell})_\ell$ von $(v^\nu)_\nu$. Dann gilt $\lim_{\nu\to\infty} v^\nu = v^*$.*

Beweis: Wir nehmen an, die Behauptung sei falsch. Dann gibt es ein $\varepsilon > 0$, so daß $\|v^* - v^\nu\| > \varepsilon$ für unendlich viele ν. Daher finden wir $\nu_1 < \nu_2 < \cdots$, so daß $\|v^* - v^{\nu_\ell}\|_2 > \varepsilon$ für $\ell = 1, 2, \ldots$.

Dann ist $(v^{\nu_\ell})_\ell$ eine Teilfolge von $(x^\nu)_\nu$. Da $(x^\nu)_\nu$ beschränkt ist, ist auch diese Teilfolge beschränkt und besitzt daher eine konvergente Teilfolge, etwa $\lim\limits_{i\to\infty} v^{\nu_{\ell_i}} = v'$. Die zuletzt gefundene Teilfolge ist selbstverständlich auch eine Teilfolge der ursprünglichen Folge $(v^\nu)_\nu$. Daher müßte $v^* = v' = \lim\limits_{i\to\infty} v^{\nu_{\ell_i}}$ gelten. Dies ist aber nach Konstruktion nicht möglich, da $\|v^* - v^{\nu_\ell}\|_2 > \varepsilon$ für alle ℓ. Widerspruch. $\rightarrow\leftarrow$ $\square$

Der hier geführte Beweis des Satzes 3.4.2 entspricht weitgehend demjenigen von [27, Theorem 2].

Beweis von Satz 3.4.2: Die Aussagen (i) und (ii) sind Spezialfälle der Aussagen (ii) bzw. (iii) von Satz 3.2.2. Da $\mathcal{M}(.,k)$ strikt konvex ist, folgt, daß $x(k)$ eindeutige Lösung von

$$\nabla_x \mathcal{M}(x,k) = 0$$

in int S_k ist. Durch Anwendung des impliziten Funktionentheorems erhält man, daß $x(k)$ (und somit auch $\lambda(k)$) insbesondere stetig von k abhängt.

Es verbleibt, die Behauptungen in (iii) und (iv) zu zeigen. Dazu müssen wir beweisen, daß für jede Folge $(\lambda^\nu)_\nu$ positiver Zahlen mit $\lim\limits_{\nu\to\infty} k^\nu = +\infty$ gilt:

$$\lim_{\nu\to\infty} x(k^\nu) = x^* \quad \text{und} \quad \lim_{\nu\to\infty} \lambda(k^\nu) = \lambda^*.$$

Zunächt werden wir zeigen, daß $\lim\limits_{\nu\to\infty} x(k^\nu) = x^*$ gilt. Zur Verkürzung der Notation sei $v^\nu = x(k^\nu)$. Nach Definition von $v^\nu = x(k^\nu)$ gilt

$$\mathcal{M}(v^\nu, k^\nu) = \min_{x\in S_{k^\nu}} \mathcal{M}(x, k^\nu) \leq \mathcal{M}(x^*, k^\nu). \tag{3.60}$$

Da S nach Voraussetzung beschränkt ist, stellt Satz 3.2.2 auch die Beschränktheit von S_{k^1} sicher. Wegen $\lim\limits_{\nu\to\infty} k^\nu = +\infty$ gilt für großes ν dann $k^\nu > k^1$, also $S_{k^\nu} \subset S_{k^1}$. Somit folgt wegen $v^\nu \in \text{int}\, S_{k^\nu}$ auch $v^\nu \in S_{k^1}$ für großes ν und somit die Beschränktheit der Folge $(v^\nu)_\nu$. Um nun $\lim\limits_{\nu\to\infty} v^\nu = x^*$ zu zeigen, genügt es nach Lemma 3.4.3 zu beweisen, daß jede konvergente Teilfolge von $(v^\nu)_\nu$ gegen x^* konvergiert. Sei $(v^{\nu_\ell})_\ell$ eine solche Folge, etwa $\lim\limits_{\ell\to\infty} v^{\nu_\ell} = \bar{x}$. Zu zeigen ist $\bar{x} = x^*$. Wegen $v^{\nu_\ell} \in \text{int}\, S_{k^{\nu_\ell}}$ gilt

$$a_j^T v^{\nu_\ell} - b_j \geq -\frac{1}{k^{\nu_\ell}}, \quad j = 1, \ldots, m. \tag{3.61}$$

Durch Grenzübergang $\ell \to \infty$ folgt aus (3.61) wegen $\lim\limits_{\ell\to\infty} k^{\nu_\ell} = +\infty$

$$a_j^T \bar{x} - b_j \geq 0, \quad j = 1, \ldots, m,$$

also $\bar{x} \in S$. Da die Menge der zulässigen Lösungen S beschränkt ist kann man $a_j^T x - b_j$ für alle $x \in S$ und alle j nach oben durch eine Konstante abschätzen. Mithin folgt

$$\frac{1}{k}\sum_{j=1}^{m} \log(k \cdot (a_j^T x - b_j) + 1) \leq \omega \frac{1}{k} \log k \quad \text{für alle } k > K \text{ und alle } x \in S \tag{3.62}$$

mit geeigneten Konstanten $\omega > 0$ und $K > 0$. Da die Folge $(v^\nu)_\nu$ beschränkt ist, kann man analog ein $\tilde{\omega} > 0$ finden, so daß

$$\frac{1}{k}\sum_{j=1}^{m} \log(k \cdot (a_j^T v^\nu - b_j) + 1) \leq \tilde{\omega} \frac{1}{k} \log k \quad \text{für alle } k > K \text{ und } \nu = 1, 2, \ldots \tag{3.63}$$

Die Ungleichung (3.63) besagt dann, daß für $k > K$ die Ungleichung

$$\mathcal{M}(v^{\nu_\ell}, k) = q(v^\nu) - \frac{1}{k}\sum_{j=1}^{m} \log(k \cdot (a_j^T v^{\nu_\ell} - b_j) + 1) \geq q(v^{\nu_\ell}) - \tilde{\omega} \frac{1}{k} \log k \tag{3.64}$$

richtig ist. Da $\bar{x} \in S$ gilt und $\mathcal{M}(., k)$ stetig auf int $S_k \supset S$ ist, gewinnen wir aus (3.64) durch Grenzübergang $\ell \to \infty$

$$q(\bar{x}) \leq \mathcal{M}(\bar{x}, k) + \tilde{\omega} \frac{1}{k} \log k \quad \text{für } k > K_0. \tag{3.65}$$

Da $\lim\limits_{k \to \infty} \frac{1}{k} \log k = 0$ und $\lim\limits_{\ell \to \infty} k^{\nu_\ell} = +\infty$, finden wir daher zu $\varepsilon > 0$ ein $N > 0$, so daß für $\ell > N$ gilt:

$$q(\bar{x}) \leq \mathcal{M}(\bar{x}, k^{\nu_\ell}) + \varepsilon \tag{3.66}$$

(dies ist wegen (3.65) möglich),

$$q(v^{\nu_\ell}) - \varepsilon \leq q(\bar{x}) \leq q(v^{\nu_\ell}) + \varepsilon \tag{3.67}$$

(wegen $\lim\limits_{\ell \to \infty} q(x^{\nu_\ell}) = q(\bar{x})$), sowie

$$\left| \frac{1}{k^{\nu_\ell}} \sum_{j=1}^{m} \log(k^{\nu_\ell} \cdot (a_j^T \bar{x} - b_j) + 1) - \frac{1}{k^{\nu_\ell}} \sum_{j=1}^{m} \log(k^{\nu_\ell} \cdot (a_j^T v^\nu - b_j) + 1) \right| < \varepsilon \tag{3.68}$$

(wegen (3.62), (3.63), $\lim\limits_{k \to \infty} \frac{1}{k} \log k = 0$ und $\bar{x} \in S$). Benutzen wir die letzten drei Ungleichungen, so ergibt sich für $\ell > N$:

$$\begin{aligned}
q(v^{\nu_\ell}) - \varepsilon &\overset{(3.67)}{\leq} q(\bar{x}) \\
&\overset{(3.66)}{\leq} \mathcal{M}(\bar{x}, k^{\nu_\ell}) + \varepsilon \\
&= \varepsilon + q(\bar{x}) - \frac{1}{k^{\nu_\ell}} \sum_{j=1}^{m} \log(k^{\nu_\ell} \cdot (a_j^T \bar{x} - b_j) + 1) \\
&\overset{(3.67)}{\leq} \varepsilon + \varepsilon + q(v^{\nu_\ell}) - \frac{1}{k^{\nu_\ell}} \sum_{j=1}^{m} \log(k^{\nu_\ell} \cdot (a_j^T \bar{x} - b_j) + 1) \\
&\overset{(3.68)}{\leq} \varepsilon + \varepsilon + \varepsilon + q(v^{\nu_\ell}) - \frac{1}{k^{\nu_\ell}} \sum_{j=1}^{m} \log(k^{\nu_\ell} \cdot (a_j^T v^{\nu_\ell} - b_j) + 1) \\
&= \mathcal{M}(v^{\nu_\ell}, k^{\nu_\ell}) + 3\varepsilon.
\end{aligned}$$

Somit gilt für großes $\ell > N$:

$$\begin{aligned} q(v^{\nu_\ell}) &\leq \mathcal{M}(v^{\nu_\ell}, k^{\nu_\ell}) + 4\varepsilon \\ &\overset{(3.60)}{\leq} \mathcal{M}(x^*, k^{\nu_\ell}) + 4\varepsilon \\ &= 4\varepsilon + q(x^*) - \frac{1}{k^{\nu_\ell}} \sum_{j=1}^m \log(\overbrace{k^{\nu_\ell} \cdot (a_j^T x^* - b_j) + 1}^{\geq 1}) \\ &\leq q(x^*) + 4\varepsilon. \end{aligned}$$

Aus der letzten Rechnung folgt dann durch Grenzübergang $\ell \to \infty$ auch $q(\bar{x}) \leq q(x^*) + 4\varepsilon$. Da $\varepsilon > 0$ beliebig war, muß daher $q(\bar{x}) \leq q(x^*)$ gelten. Nun war aber $\bar{x} \in S$ und dies zeigt, daß $\bar{x}$ eine Optimallösung von (QP) sein muß. Da x^* aber eindeutige Optimallösung von (QP) war, folgt $\bar{x} = x^*$. Wie oben bereits erwähnt, impliziert Lemma 3.4.3 nun sogar $\lim\limits_{\nu\to\infty} v^\nu = \lim\limits_{\nu\to\infty} x(k^\nu) = x^*$. Daß auch $\lim\limits_{\nu\to\infty} \lambda(k^\nu) = \lambda^*$ gilt, wird letztendlich aus der Behauptung (iv) folgen.

Aus der Stetigkeit von q folgt wegen $\lim\limits_{\nu\to\infty} x(k^\nu) = x^*$ auch sofort $\lim\limits_{\nu\to\infty} q(x(k^\nu)) = q(x^*)$. Um zu sehen, daß $\lim\limits_{\nu\to\infty} \mathcal{M}(x(k^\nu), k^\nu) = q(x^*)$ gilt, bemerken wir, daß wegen der Ungleichung (3.63) und $\lim\limits_{\nu\to\infty} k^\nu = +\infty$ für alle großen ν die Abschätzung

$$|q(x(k^\nu)) - \mathcal{M}(x(k^\nu), k^\nu)| \leq \tilde{\omega} \frac{1}{k^\nu} \log k^\nu \tag{3.69}$$

gilt. Für $\nu \to \infty$ strebt die rechte Seite der Ungleichung (3.69) gegen 0. Da $q(x(k^\nu)) \to q(x^*)$ folgt dann auch $\mathcal{M}(x(k^\nu), k^\nu) \to q(x^*)$. Damit sind alle Behauptungen in (iii) bewiesen.

Zuletzt zeigen wir die Aussagen in (iv), d.h. wir schätzen die Normen von $\Delta x := x(k) - x^*$ und $\Delta\lambda := \lambda(k) - \lambda^*$ ab. Wie im Beweis von Satz 3.3.3 partitionieren wir den Vektor $\lambda(k)$ als $\lambda(k) = \binom{\lambda_{(p)}(k)}{\lambda_{(m-p)}(k)}$. Da $x(k) \in \operatorname{int} S_k$ folgt aus der Definition von $\lambda(k)$, daß $\lambda(k) > 0$ für alle ν.

Wegen $\lim\limits_{k\to\infty} x(k) = x^*$ gilt für alle großen $k > 0$

$$a_j^T x(k) - b_j \geq \frac{\sigma}{2}, \quad j = p+1, \ldots, m. \tag{3.70}$$

Dabei ist σ das in (3.23) definierte Maß an Nichtdegeneriertheit von (QP). Aus der letzten Gleichung folgt dann zusammen mit der Nichtnegativität von $\lambda(k)$

$$0 < \lambda_j(k) \leq \frac{1}{k\frac{\sigma}{2} + 1} \leq \frac{2}{\sigma} \cdot \frac{1}{k}, \quad j = p+1, \ldots, m. \tag{3.71}$$

Wegen $\lambda^*_{(m-p)} = 0$ zeigt dies, es ein K_0 gibt, so daß für $k > K_0$ die Abschätzung

$$\|\Delta\lambda_{(m-p)}\| = \|\lambda_{(m-p)}(k) - \lambda^*_{(m-p)}\|_\infty \leq \frac{2}{\sigma} \cdot \frac{1}{k} \tag{3.72}$$

richtig ist.

Wir definieren die Funktion $\hat{\lambda}_{(p)} : S_k \times \mathbb{R}_+ \to \mathbb{R}_+^p$ durch

$$\hat{\lambda}_j(x,k) := \frac{1}{k \cdot (a_j^T x - b_j) + 1}, \quad j = 1, \dots, p.$$

Taylorentwicklung von $\hat{\lambda}$ um x^* liefert dann unter Ausnutzung von $\hat{\lambda}_{(p)}(x(k),k) = \lambda_{(p)}(k)$:

$$\begin{aligned}\lambda_{(p)}(k) &= \hat{\lambda}_{(p)}(x^*,k) + D_x\hat{\lambda}_{(p)}(x^*,k)(\underbrace{x(k)-x^*}_{=\Delta x}) + H(\Delta x)\\ &= \hat{\lambda}_{(p)}(x^*,k) + D_x\hat{\lambda}_{(p)}(x^*,k)\Delta x + H(\Delta x) \qquad (3.73)\end{aligned}$$

wobei $H(0) = 0$ und $\|H(\Delta x)\|_\infty \leq R(\Delta x)\|\Delta x\|_\infty$ mit $R(\Delta x) \to 0$ für $\Delta x \to 0$.

Wir berechnen $\hat{\lambda}_{(p)}(x^*,k)$ und $D_x\hat{\lambda}_{(p)}(x^*,k)$. Für $j = 1, \dots, p$ gilt wegen $a_j^T x^* - b_j = 0$

$$\hat{\lambda}_j(x^*,k) = \frac{1}{k \cdot 0 + 1} = 1.$$

Dieses Ergebnis schreibt sich auch als

$$\hat{\lambda}_{(p)}(x^*,k) = e_{(p)}. \qquad (3.74)$$

Ferner ist

$$\nabla_x\hat{\lambda}_j(x,k) = -k\frac{a_j}{(k \cdot (a_j^T x - b_j) + 1)^2}, \quad j = 1, \dots, p$$

und daher

$$D_x\hat{\lambda}_{(p)}(x^*,k) = -k\begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} \qquad (3.75)$$

Benutzt man (3.74) und (3.75) in (3.73), so folgt

$$\lambda_{(p)}(k) = e_{(r)} - k\begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix}\Delta x + H(\Delta x).$$

Subtrahiert man auf beiden Seiten der letzten Identität noch $\lambda_{(p)}^*$, so resultiert dies in

$$\Delta\lambda_{(r)} = e_{(p)} - \lambda_{(p)}^* - k\begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix}\Delta x + H(\Delta x). \qquad (3.76)$$

Weiterhin gilt

$$\begin{aligned}
0 &= \nabla_x \mathcal{M}(x(k), k) \\
&= Qx(k) + c - \sum_{j=1}^{m} \frac{a_j}{k \cdot (a_j^T x(k) - b_j) + 1} \\
&= Qx(k) + c - \sum_{j=1}^{m} \lambda_j(k) a_j \\
&= Q(x(k) - x^*) + Qx^* + c - \sum_{j=1}^{m} \lambda_j(k) a_j \\
&= Q\Delta x + Qx^* + c - \sum_{j=1}^{m} (\Delta\lambda_j + \lambda_j^*) a_j \\
&= Q\Delta x - \sum_{j=1}^{m} \Delta\lambda_j a_j + \underbrace{Qx^* + c - \sum_{j=1}^{m} \lambda_j^* a_j}_{=0} \\
&= Q\Delta x - \sum_{j=1}^{p} \Delta\lambda_j a_j - \sum_{j=p+1}^{m} (\lambda_j(k) - \underbrace{\lambda^*}_{=0}) a_j \\
&= Q\Delta x - \sum_{j=1}^{p} \Delta\lambda_j a_j - \sum_{j=p+1}^{m} \lambda_j(k) a_j,
\end{aligned}$$

also

$$Q\Delta x - \begin{pmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_p^T \end{pmatrix} \Delta\lambda_{(p)} = \sum_{j=p+1}^{m} \lambda_j(k). \tag{3.77}$$

Aus den Gleichungen (3.76) und (3.77) folgt, daß Δx und $\Delta\lambda_{(p)}$ Lösung des folgenden linearen Gleichungssystems sind:

$$\left(\begin{array}{c|cccc} Q & -a_1 & -a_2 & \cdots & -a_p \\ \hline \begin{matrix} -a_1^T \\ -a_2^T \\ \vdots \\ -a_p^T \end{matrix} & \multicolumn{4}{c}{-\dfrac{1}{k} I^{p,p}} \end{array}\right) \cdot \begin{pmatrix} \Delta x \\ \Delta\lambda_{(p)} \end{pmatrix} = \begin{pmatrix} \sum\limits_{j=p+1}^{m} \lambda_j(k) a_j \\ -\frac{1}{k}(e_{(p)} - \lambda_{(p)}^*) - \frac{1}{k} H(\Delta x) \end{pmatrix}.$$

Dieses Gleichungssystem schreibt sich auch als

$$G_k \cdot \begin{pmatrix} \Delta x \\ \Delta\lambda_{(p)} \end{pmatrix} = \begin{pmatrix} \sum\limits_{j=p+1}^{m} \lambda_j(k) a_j \\ -\frac{1}{k}(e_{(p)} - \lambda_{(p)}^*) \end{pmatrix} + \begin{pmatrix} 0 \\ -\frac{1}{k} H(\Delta x) \end{pmatrix}. \tag{3.78}$$

Mit G_k betrachtet man noch die Matrix

$$G_\infty = \lim_{k\to\infty} G_k = \left(\begin{array}{c|cccc} Q & -a_1 & -a_2 & \cdots & -a_p \\ \hline -a_1^T & & & & \\ -a_2^T & & 0 & & \\ \vdots & & & & \\ -a_p^T & & & & \end{array}\right).$$

Vollkommen analog zum Beweis der Nichtsingularität von B_k bzw. B_∞ in Satz 3.3.3 beweist man, daß G_k und G_∞ nichtsingulär sind. Mit Lemma 3.3.2 können wir daher erneut schließen, daß $\mathrm{lub}_\infty((G_k)^{-1}) \leq c$ für eine Konstante $c > 0$ und alle ν gilt.

Wir schätzen die Normen der Vektoren auf der rechten Seite von (3.78) ab. Wegen (3.71) gilt

$$\|\lambda_j(k)a_j\|_\infty \leq \frac{2\|a_j\|_\infty}{\sigma} \cdot \frac{1}{k}, \quad j = p+1, \ldots, m,$$

so daß die Abschätzung

$$\left\| \begin{pmatrix} \sum_{j=p+1}^{m} \lambda_j(k)a_j \\ -\frac{1}{k}(e_{(p)} - \lambda^*_{(p)}) \end{pmatrix} \right\|_\infty \leq \omega \frac{1}{k} \tag{3.79}$$

mit

$$\omega = \max\left\{ \|e_{(p)} - \lambda^*_{(p)}\|_\infty, \frac{2(m-p)}{\sigma} \max\{\|a_j\|_\infty : j = p+1, \ldots, m\} \right\}.$$

für alle großen k richtig ist. Wegen $\|H(\Delta x)\|_\infty \leq R(\Delta x)\|\Delta x\|_\infty$ mit $R(\Delta x) \to 0$ für $\Delta x \to 0$ gilt

$$\left\| \begin{pmatrix} 0 \\ -\frac{1}{k}H(\Delta x) \end{pmatrix} \right\|_\infty \leq \omega' \frac{1}{k} \tag{3.80}$$

für eine Konstante $\omega' > 0$. Somit erhält man aus (3.78) zusammen mit $\mathrm{lub}_\infty((B_k)^{-1}) \leq c$ für alle großen k die Abschätzung

$$\begin{aligned} \left\| \begin{pmatrix} \Delta x \\ \Delta\lambda_{(r)} \end{pmatrix} \right\|_\infty &\leq \mathrm{lub}_\infty((B_k)^{-1}) \left(\left\| \begin{pmatrix} \sum_{j=p+1}^{m} \lambda_j(k)a_j \\ -\frac{1}{k}(e_{(p)} - \lambda^*_{(p)}) \end{pmatrix} \right\|_\infty + \left\| \begin{pmatrix} 0 \\ -\frac{1}{k}H(\Delta x) \end{pmatrix} \right\|_\infty \right) \\ &\leq c(\omega + \omega') \cdot \frac{1}{k} \quad . \end{aligned} \tag{3.81}$$

Nimmt man nun (3.72) und (3.81) zusammen, so folgt

$$\max\{\|x(k) - x^*\|_\infty, \|\lambda(k) - \lambda^*\|_\infty\} \leq \max\{c(\omega + \omega'), \frac{2}{\sigma}\} \frac{1}{k}$$

für alle $k > K'$, wobei $K' \geq k_0$ eine geeignete Konstante ist. Setzen wir nun

$$\begin{aligned} \omega_x &= \max_{k_0 \leq k \leq K'} k\|x(k) - x^*\|_\infty \\ \omega_\lambda &= \max_{k_0 \leq k \leq K'} k\|\lambda(k) - \lambda^*\|_\infty \end{aligned}$$

so folgt die Behauptung des Satzes mit $c_4 := \max\{\omega_x, \omega_\lambda, c(\omega + \omega'), \frac{2}{\sigma}\}$. □

Kapitel 4

Die Modifizierte Barrieremethode

Wir betrachten wieder das quadratische Optimierungsproblem (QP). In diesem Kapitel wird zunächst die Modifizierte Barrieremethode bei der Anwendung auf (QP) vorgestellt und dann einige grundsätzliche Eigenschaften des Verfahrens hergeleitet.

4.1 Herleitung des Verfahrens

Sei $\lambda^0 \in \mathbb{R}^m$, $\lambda^0 > 0$. Nach Satz 3.2.5 existiert dann $x^0 = x(\lambda^0, k) \in \arg\ \min\{\mathcal{F}(x, \lambda^0, k) : x \in S_k\}$ mit $\nabla_x \mathcal{F}(x^0, \lambda^0, k) = 0$. Die Gleichung (3.8) für den Gradienten der Modifizierten Barrierefunktion $\mathcal{F}$ liefert dann:

$$Qx^0 + c - \sum_{j=1}^{m} \frac{\lambda_j^0}{k \cdot (a_j^T x^0 - b_j) + 1} \cdot a_j = 0. \tag{4.1}$$

Sei (x^*, λ^*) ein Karush–Kuhn–Tucker Paar für (QP). Dann können wir die Kuhn–Tucker Bedingung (1.2a) auch in folgender Form schreiben:

$$Qx^* + c - \sum_{j=1}^{m} \lambda_j^* a_j = 0. \tag{4.2}$$

Vergleicht man die beiden Identitäten (4.1) und (4.2), so motiviert dies die folgende Update–Formel, um aus λ^0 einen neuen Vektor λ^1 zu gewinnen:

$$\lambda_j^1 = \frac{\lambda_j^0}{k \cdot (a_j^T x^0 - b_j) + 1} \qquad \text{für } j = 1, \ldots, m.$$

Da x^0 im Inneren int S_k von S_k liegt (vgl. Satz 3.2.2 und Satz 3.2.5), gilt dann wegen $\lambda^0 > 0$ und der Update–Formel auch $\lambda^1 > 0$.

Die zuletzt erhaltene Update–Formel und die Ergebnisse von Satz 3.2.2 bzw. Satz 3.2.5 führen nun direkt dazu, den folgenden Algorithmus zur Lösung von (QP) zu betrachten, der auf der Modifizierten Barrierefunktion $\mathcal{F}(x, \lambda, k)$ beruht:

Algorithmus 4.1.1 (Modifizierte Barrieremethode)

(1) *Wähle* $\kappa > 0$ *und eine Folge* $(k^\nu)_\nu$ *mit* $0 < \kappa \leq k^\nu$ *für* $\nu = 0, 1, 2, \ldots$.

(2) *Wähle* $\lambda^0 \in \mathrm{IR}^m$ *mit* $\lambda^0 > 0$.

(3) *Setze* $\nu := 0$.

(4) *Zu gegebenem* $\lambda^\nu > 0$ *und* $k^\nu > 0$ *bestimme* $x^\nu \in \arg\ \min\{\mathcal{F}(x, \lambda^\nu, k^\nu) : x \in \mathrm{IR}^n\}$.

(5) *Setze* $\lambda_j^{\nu+1} := \lambda_j^\nu / [k^\nu \cdot (a_j^T x^\nu - b_j) + 1]$ *für* $j = 1, \ldots, m$.

(6) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (4).

Man beachte, daß wir hier außer $k^\nu > \kappa$ a priori keine weiteren Forderungen an die Folge $(k^\nu)_\nu$ der Barriere–Parameter stellen. Besonders interessant sind selbstverständlich unter anderem die beiden Fälle

(1) $k^\nu = k$ für $\nu = 0, 1, 2, \ldots$.

(2) $\lim\limits_{\nu \to \infty} k^\nu = +\infty$

Eine einfache Induktion zeigt, daß x^ν für $\nu = 0, 1, 2, \ldots$ definiert ist, in int S_{k^ν} liegt und invariant $\lambda^\nu > 0$ gilt. Für $\nu = 0$ folgt die Aussage sofort aus Satz 3.2.5 wegen $\lambda^0 > 0$. Ist S beschränkt, so weiß man aufgrund von Satz 3.2.2 sogar, daß x^ν eindeutig bestimmt ist. Sei die Aussage für $\nu \geq 0$ bereits bewiesen. Dann folgt wegen $x^\nu \in \text{int } S_{k^\nu}$ aus der Update–Formel in Schritt (5), daß $\lambda^{\nu+1} > 0$ gilt. Daher stellen dann Satz 3.2.2 oder Satz 3.2.5 wiederum sicher, daß $x^{\nu+1} \in \text{int } S_{k^{\nu+1}}$ definiert ist. Wir fassen diese Ergebnisse im folgenden Satz zusammen:

Satz 4.1.2 *Unter der Voraussetzung, daß die Menge* S^* *der Optimallösungen von* (QP) *nichtleer beschränkt ist, ist die Folge* $(x^\nu, \lambda^\nu)_\nu$*, die der Algorithmus 4.1.1 erzeugt, definiert und es gilt* $\lambda^\nu > 0$ *für* $\nu = 0, 1, 2, \ldots$.

Wir werden die Folge $(x^\nu)_\nu$ als die primäre und $(\lambda^\nu)_\nu$ als die duale Folge bezeichnen. Im Abschnitt 5.4 wird deutlich werden, warum wir diese Bezeichnungen verwenden.

Für die weitere Analyse werden wir voraussetzen, daß die Menge S^* der Optimallösungen von (QP) eine nichtleere kompakte Menge ist. Für einige Resultate werden wir die stärkere Voraussetzung benutzen, daß nicht nur S^* sondern sogar die Menge S der zulässigen Lösungen von (QP) beschränkt ist. Sofern die erarbeiteten Ergebnisse für beide Fälle gelten, werden wir die Voraussetzungen nicht jedes Mal erneut aufführen.

Falls ein Resultat nur für den Fall gilt, daß S beschränkt ist, wird dies jeweils entsprechend gekennzeichnet.

Wir bezeichnen für den Rest der Darstellung die Menge aller zulässigen Optimallösungen von (QP) weiterhin mit S^*, den Optimalwert von q auf S mit q^*.

Bemerkung 4.1.3 *Aus Satz 3.2.2 bzw. Satz 3.2.5 und der Update–Formel in Schritt (5) ergibt sich die folgende wichtige Gleichung:*

$$0 = \nabla_x \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) = Qx^\nu + c - \sum_{j=1}^m \frac{\lambda^\nu}{k^\nu \cdot (a_j^T x^\nu - b_j) + 1} a_j = Qx^\nu + c - \sum_{j=1}^m \lambda_j^{\nu+1} a_j.$$

Bemerkung 4.1.4 *Aus der Update–Formel in Schritt (5) ersieht man ferner, daß*

$$b_j - a_j^T x^\nu = \frac{1}{k^\nu}\left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right), \qquad j = 1, \ldots, m.$$

4.2 Grundlegende Eigenschaften des Algorithmus

In diesem Abschnitt sollen einige grundlegende Eigenschaften der Folgen $(x^\nu, \lambda^\nu)_\nu$, die durch den Algorithmus 4.1.1 erzeugt wird, gezeigt werden. Dabei übertragen wir einige Aussagen aus [29] vom linearen auf den quadratischen Fall.

Das erste Lemma ist eine Verallgemeinerung von [29, Lemma 2.2]:

Lemma 4.2.1 *Sei x^* eine Optimallösung von* (QP). *Dann gilt für $\nu = 0, 1, 2, \ldots$*

$$\mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \le \frac{1}{2}(x^*)^T Q x^* + c^T x^* = q^*. \tag{4.3}$$

Beweis: Sei $x^* \in S^*$ eine beliebige Optimallösung von (QP). Nach Definition von x^ν gilt dann

$$\begin{aligned}
\mathcal{F}(x^\nu, \lambda^\nu, k^\nu) &\le \mathcal{F}(x^*, \lambda^\nu, k^\nu) \\
&= \frac{1}{2}(x^*)^T Q x^* + c^T x^* - \frac{1}{k^\nu} \sum_{j=1}^m \lambda^\nu \log(k^\nu \cdot (a_j^T x^* - b_j) + 1) \\
&\le \tfrac{1}{2}(x^*)^T Q x^* + c^T x^* = q(x^*),
\end{aligned}$$

Hierbei folgt die letzte Ungleichung, da wegen der Zulässigkeit von x^* die Ungleichung $a_j^T x^* - b_j \ge 0$, also $\lambda^\nu \log(k^\nu \cdot (a_j^T x^* - b_j) + 1) \ge 0$, für $j = 1, \ldots, m$ gilt.. □

Das folgende Lemma verallgemeinert die Aussagen von [29, Lemma 2.3] und zeigt die Monotonie der Funktionswerte $\mathcal{F}(x^\nu, \lambda^\nu, k^\nu)$.

Lemma 4.2.2 *Ist S beschränkt, so gilt für die Folge $(x^\nu, \lambda^\nu)_\nu$, die vom Algorithmus 4.1.1 erzeugt wird,*

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k^{\nu+1}) - \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \geq \frac{k^\nu}{2(c_\kappa k^\nu + 1)} \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j)^2 \geq 0, \quad \nu = 0, 1, 2, \ldots,$$

wobei

$$c_\kappa = \max\{\|Ax - b\|_\infty : x \in S_k\} < +\infty.$$

Insbesondere ist die Folge $(\mathcal{F}(x^\nu, \lambda^\nu, k^\nu))_\nu$ monoton steigend. Ist $(k^\nu)_\nu$ beschränkt, so existiert eine Konstante $\hat{c}_\kappa$, so daß

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k) - \mathcal{F}(x^\nu, \lambda^\nu, k) \geq \hat{c}_\kappa k^\nu \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j)^2, \quad \nu = 0, 1, 2, \ldots.$$

Beweis: Da q konvex ist, gilt

$$q(x^{\nu+1}) - q(x^\nu) \geq \nabla q(x^\nu)^T (x^{\nu+1} - x^\nu).$$

Wegen $\nabla q(x^\nu) = Qx^\nu + c$ und Bemerkung (4.1.3) haben wir dann

$$q(x^{\nu+1}) - q(x^\nu) \geq \sum_{j=1}^m \lambda_j^{\nu+1} a_j^T (x^{\nu+1} - x^\nu). \tag{4.4}$$

Wir benutzen nun noch, daß für $t > -1$ die Ungleichung $\log(t+1) \leq t$ gilt, und erhalten:

$$\begin{aligned}
&\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k) - \mathcal{F}(x^\nu, \lambda^\nu, k) \\
&= q(x^{\nu+1}) - q(x^\nu) + \\
&\quad \tfrac{1}{k^\nu} \textstyle\sum_{j=1}^m \lambda_j^\nu \log[k^\nu(a_j^T x^\nu - b_j) + 1] - \tfrac{1}{k^{\nu+1}} \sum_{j=1}^m \lambda_j^{\nu+1} \log[k^{\nu+1}(a_j^T x^{\nu+1} - b_j) + 1] \\
&\overset{(4.4)}{\geq} \textstyle\sum_{j=1}^m \lambda_j^{\nu+1} a_j^T (x^{\nu+1} - x^\nu) + \\
&\quad \tfrac{1}{k^\nu} \textstyle\sum_{j=1}^m \lambda_j^\nu \log[k^\nu(a_j^T x^\nu - b_j) + 1] - \tfrac{1}{k^{\nu+1}} \sum_{j=1}^m \lambda_j^{\nu+1} \log[k^{\nu+1}(a_j^T x^{\nu+1} - b_j) + 1] \\
&\geq \textstyle\sum_{j=1}^m \lambda_j^{\nu+1} a_j^T (x^{\nu+1} - x^\nu) + \tfrac{1}{k^\nu} \sum_{j=1}^m \lambda_j^\nu \log[k^\nu(a_j^T x^\nu - b_j) + 1] - \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^{\nu+1} - b_j) \\
&= \tfrac{1}{k^\nu} \textstyle\sum_{j=1}^m \lambda_j^\nu \log[k^\nu(a_j^T x^\nu - b_j) + 1] - \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j) \\
&= \tfrac{1}{k^\nu} \textstyle\sum_{j=1}^m \lambda_j^{\nu+1} [k^\nu(a_j^T x^\nu - b_j) + 1] \log[k^\nu(a_j^T x^\nu - b_j) + 1] - \tfrac{1}{k^\nu} \sum_{j=1}^m k^\nu \lambda_j^{\nu+1}(a_j^T x^\nu - b_j) \\
&= \tfrac{1}{k^\nu} \textstyle\sum_{j=1}^m \lambda_j^{\nu+1} \Big\{[k^\nu(a_j^T x^\nu - b_j) + 1] \log[k^\nu(a_j^T x^\nu - b_j) + 1] - k^\nu(a_j^T x^\nu - b_j)\Big\}.
\end{aligned}$$

Wenn man die Funktion $\varphi(t) = (t+1)\log(t+1) - t$ einführt, so schreibt sich dieses Ergebnis auch als

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k^{\nu+1}) - \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \geq \frac{1}{k^\nu} \sum_{j=1}^m \lambda_j^{\nu+1} \varphi(k^\nu \cdot (a_j^T x^\nu - b_j)). \tag{4.5}$$

Da $k^\nu \geq \kappa$, folgt $S_{k^\nu} \subseteq S_\kappa$ für $\nu = 0, 1, 2, \ldots$. Da S_κ nach Satz 3.2.2 beschränkt (und wegen der Abgeschlossenheit somit auch kompakt) ist, gilt $c_\kappa = \max\{\|Ax - b\|_\infty : x \in S_\kappa\} < +\infty$. Aus der Definition von c_κ folgt wegen $x^\nu \in \operatorname{int} S_{k^\nu} \subset S_\kappa$, daß $c_\kappa \geq a_j^T x^\nu - b_j$ für $j = 1, \ldots, m$ und *alle* $\nu \geq 0$. Wir betrachten die Funktion $\varphi(t) = (t+1)\log(t+1) - t$ für $-1 < t \leq c_\kappa k^\nu$. Durch Taylorentwicklung von φ um $t_0 = 0$ erhalten wir für $-1 < t \leq c_\kappa k^\nu$

$$\begin{aligned} \varphi(t) &= \varphi(0) + \varphi'(0)t + \tfrac{1}{2}t^2\varphi''(\zeta) \\ &= 0 + t\log(0+1) + \tfrac{1}{2(\zeta+1)}t^2 = \tfrac{1}{2(\zeta+1)}t^2, \end{aligned} \tag{4.6}$$

wobei ζ ein Punkt im Intervall $(\min\{0, t\}, \max\{0, t\})$ ist. In jedem Fall ist $\zeta \leq c_\kappa k^\nu$. Somit ersieht man aus (4.6), daß

$$\varphi(t) \geq \frac{1}{2(c_\kappa k^\nu + 1)}t^2.$$

Die Verwendung dieser Ungleichung in (4.5) zeigt dann, daß

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k^{\nu+1}) - \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \geq \frac{1}{k^\nu}\frac{1}{2(c_\kappa k^\nu + 1)}\sum_{j=1}^m \lambda_j^{\nu+1}\left(k^\nu\left(a_j^T x^\nu - b_j\right)\right)^2,$$

also

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k^{\nu+1}) - \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \geq \frac{k^\nu}{2(c_\kappa k^\nu + 1)}\sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j)^2 \geq 0, \tag{4.7}$$

gilt. Dies zeigt die erste Behauptung des Lemmas. Ist die Folge $(k^\nu)_\nu$ beschränkt, so ist $k^\nu < K$ für eine Konstante K. Dann gilt

$$\frac{1}{2(c_\kappa k^\nu + 1)} \geq \frac{1}{2(c_\kappa K + 1)}. \tag{4.8}$$

Setzen wir $\hat{c}_\kappa := \frac{1}{2(c_\kappa K+1)}$, so folgt nun die zweite Behauptung des Lemmas sofort aus der Definition von $\hat{c}_\kappa$ mit den Ungleichungen (4.8) und (4.7). $\square$

Aus Lemma 4.2.2 ergibt sich nun das folgende

Korollar 4.2.3 *Ist S beschränkt, so existiert $\lim\limits_{\nu\to\infty} \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) =: \mathcal{F}^\infty$ und es gilt $\mathcal{F}^\infty \leq q^*$.*

Beweis: Nach Lemma 4.2.2 ist die Folge $(\mathcal{F}(x^\nu, \lambda^\nu, k^\nu))_\nu$ monoton wachsend. Wegen Lemma 4.2.1 ist die Folge von oben durch q^* beschränkt und es folgt, daß die Folge gegen eine Zahl F^∞ konvergiert. Für alle $\nu = 0, 1, 2, \ldots$ gilt nach Lemma 4.2.1 $\mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \leq q^*$ und somit muß auch der Grenzwert $\mathcal{F}^*$ die Ungleichung $\mathcal{F}^* \leq q^*$

erfüllen. □

Aus Korollar 4.2.3 kann man nun (unter der Voraussetzung, daß S beschränkt ist) direkt weitere Folgerungen ableiten, wie z.B. Powell in [29] (für den linearen Fall). Sei dazu $0 < \varepsilon < \frac{c_\kappa}{2}$ so klein gewählt, daß

$$0 < \frac{2\varepsilon}{1 - 2\varepsilon c_\kappa} < \kappa.$$

Dann gilt

$$k^\nu \geq \kappa > \frac{2\varepsilon}{1 - 2\varepsilon c_\kappa}, \quad \nu = 0, 1, 2, \ldots .$$

und daraus ergibt sich dann sofort durch elementare Umformungen

$$\frac{k^\nu}{2(c_\kappa k^\nu + 1)} \geq \varepsilon, \quad \nu = 0, 1, 2, \ldots . \tag{4.9}$$

Nach Lemma 4.2.2 gilt dann zusammen mit (4.9)

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k) - \mathcal{F}(x^\nu, \lambda^\nu, k) \geq \varepsilon \sum_{j=1}^{m} \lambda_j^{\nu+1} (a_j^T x^\nu - b_j)^2 \geq 0, \quad \nu = 0, 1, 2, \ldots . \tag{4.10}$$

Da der Grenzwert $\mathcal{F}^\infty = \lim_{\nu \to \infty} \mathcal{F}(x^\nu, \lambda^\nu, k^\nu)$ existiert, ist die Folge $(\mathcal{F}(x^\nu, \lambda^\nu, k^\nu))_\nu$ insbesondere eine Cauchy–Folge. Daher konvergiert die Differenz auf der linken Seite von (4.10) für $\nu \to \infty$ gegen 0. Somit muß auch

$$\lim_{\nu \to \infty} \sum_{j=1}^{m} \lambda_j^{\nu+1} (a_j^T x^\nu - b_j)^2 = 0$$

gelten. Da jeder Term in der obigen Summe nichtnegativ ist, folgt

$$\lim_{\nu \to \infty} \lambda_j^{\nu+1} (a_j^T x^\nu - b_j)^2 = 0, \quad j = 1, \ldots, m. \tag{4.11}$$

Vergleicht man dies mit der Komplementaritätsbedingung (1.2c), so wird dadurch die Hoffnung geweckt, daß, falls die primäre und die duale Folge konvergieren, die Grenzwerte ein Karush–Kuhn–Tucker Paar für (QP) sind.

Die zuletzt gewonnenen Ergebnisse setzten voraus, daß die Menge S der zulässigen Lösungen beschränkt ist. Wir zeigen als nächstes eine weitere Eigenschaft der Folge $(x^\nu, \lambda^\nu)_\nu$, die auch dann gilt, wenn man nur die Beschränktheit der Menge S^* der Optimallösungen von (QP) voraussetzt.

Lemma 4.2.4 *Konvergiert die Folge $(\lambda^\nu)_\nu$, so folgt*

$$\lim_{\nu \to \infty} \lambda_j^{\nu+1} (a_j^T x^\nu - b_j) = 0.$$

Beweis: Aus Bemerkung 4.1.4 folgt

$$\lambda_j^{\nu+1}(a_j^T x^\nu - b_j) = \lambda_j^{\nu+1} \frac{1}{k^\nu} \left(\frac{\lambda_j^\nu}{\lambda_j^{\nu+1}} - 1 \right) = \frac{1}{k^\nu} \left(\lambda_j^\nu - \lambda_j^{\nu+1} \right). \tag{4.12}$$

Da nach Voraussetzung die duale Folge $(\lambda^\nu)_\nu$ konvergiert, ist sie insbesondere eine Cauchy–Folge. Da $k^\nu \geq \kappa$ und somit $\frac{1}{k^\nu}$ für $\nu \to \infty$ beschränkt bleibt, konvergiert daher der am weitesten rechts stehende Ausdruck in (4.12) gegen 0, wenn ν gegen Unendlich strebt. $\square$

Das Ergebnis aus Lemma 4.2.4 hat eine interessante Konsequenz, die in folgendem Korollar notiert ist.

Korollar 4.2.5 *Gilt* $\lim\limits_{\nu\to\infty} x^\nu = \hat{x}$ *und* $\lim\limits_{\nu\to\infty} \lambda^\nu = \hat{\lambda}$, *so erfüllen* $\hat{x}$ *und* $\hat{\lambda}$ *die Karush–Kuhn–Tucker Bedingungen (1.2a) bis (1.2c) für* (QP) , *d.h.* $\hat{x}$ *ist Optimallösung für* (QP) *und* $\hat{\lambda}$ *ein zugehöriger Lagrangemultiplikator.*

Beweis: Wir verifizieren die Karush–Kuhn–Tucker Bedingungen für $\hat{x}, \hat{\lambda}$. Nach Bemerkung 4.1.3 gilt für $\nu = 0, 2, \ldots$, daß $Qx^\nu + c = \sum\limits_{j=1}^m \lambda_j^{\nu+1} a_j$. Somit ist

$$Q\hat{x} + c = \lim_{\nu\to\infty} Qx^\nu + c = \lim_{\nu\to\infty} \sum_{j=1}^m \lambda_j^{\nu+1} a_j = \lim_{\nu\to\infty} \sum_{j=1}^m \lambda_j^\nu a_j = \sum_{j=1}^m \hat{\lambda}_j a_j. \tag{4.13}$$

Aus $\lambda^\nu > 0$ für alle ν folgt auch $\hat{\lambda} = \lim\limits_{\nu\to\infty} \lambda^\nu \geq 0$.

Als nächstes zeigen wir, daß $\hat{x}$ zulässig für (QP) ist. Aus der Update–Formel in Schritt (5) des Algorithmus 4.1.1 ergibt sich durch eine einfache Induktion:

$$\lambda_j^\nu = \frac{\lambda^0}{\prod_{\ell=1}^{\nu-1} [k^\ell \cdot (a_j^T x^\ell - b_j) + 1]} \qquad \text{für } j = 1, \ldots, m. \tag{4.14}$$

Wäre nun $a_{j_0}^T \hat{x} - b_{j_0} < 0$ für ein j_0, so folgte wegen der Konvergenz der x^ν gegen $\hat{x}$, daß für alle großen ν auch $a_{j_0}^T x^\nu - b_{j_0} \leq \tau < 0$ für eine Konstante $\tau < 0$, also

$$k^\nu \cdot (a_{j_0}^T x^\nu - b_{j_0}) + 1 \leq k^\nu \tau + 1 \leq \kappa\tau + 1 < 1,$$

gelten müßte. Damit würde aber für dieses j_0 die Folge $\lambda_{j_0}^\nu$ divergieren, wie man sofort aus (4.14) ersieht. Dies steht im Widerspruch zur Konvergenz der Folge $(\lambda^\nu)_\nu$ gegen $\hat{\lambda}$. Somit muß

$$a_j^T \hat{x} - b_j \geq 0, \text{ für } j = 1, \ldots, m \tag{4.15}$$

gelten. Nach Lemma 4.2.4 gilt für $j = 1, \ldots, m$:

$$0 = \lim_{\nu\to\infty} \lambda_j^{\nu+1}(a_j^T x^\nu - b_j) = \lim_{\nu\to\infty} \lambda_j^{\nu+1} \lim_{\nu\to\infty} (a_j^T x^\nu - b_j) = \hat{\lambda}_j (a_j^T \hat{x} - b_j),$$

also
$$\hat{\lambda}_j(a_j^T\hat{x} - b_j) = 0 \text{ für } j = 1, \ldots, m. \tag{4.16}$$
Die Gleichungen (4.13), (4.15) und (4.16) sind genau die Karush–Kuhn–Tucker Bedingungen (1.2a) bis (1.2c) für (QP). □

Kapitel 5

Die Konvergenz des Verfahrens

Dieses Kapitel beschäftigt sich mit der Konvergenzanalyse der Modifizierten Barrieremethode bei Anwendung auf das quadratische konvexe Programm (QP) bzw. bei Anwendung auf das lineare Programm (LP). Dabei werden verschiedene Ansätze dargestellt.

Der erste benutzt die Ergebnisse aus den Abschnitten 3.3 und 3.4. Er entspricht den Ergebnissen von Polyak in [28] für lineare Programme bzw. in [27] für konvexe nichtlineare Probleme. Dieser Ansatz ist ist im Abschnitt 5.1 dargestellt. Er verwendet recht starke Voraussetzungen an die Regularität der Optimallösungen von (QP) bzw (LP).

Im Abschnitt 5.2 zitieren wir dann die Resultate von Powell aus [29] für lineare Programme, für die als einzige Voraussetzung die Beschränktheit der zulässigen Punktmenge nötig ist.

Um die Voraussetzungen für den dritten Ansatz zu schaffen, geben wir im Abschnitt 5.3 zunächst eine kurze Einführung in die benötigten Definitionen und Ergebnisse aus der konvexen Analysis. Dabei wird unter anderem ein duales Problem zu (QP) eingeführt.

Für den dritten Ansatz wählen wir einen anderen Zugang zum Verfahren. Dabei wird die Modifizierte Barrieremethode zunächst unter der Voraussetzung betrachtet, daß die Menge S^* der Optimallösungen von (QP) beschränkt und nichtleer ist. Diese Voraussetzung ist insbesondere dann erfüllt, wenn die Menge $S = \{x : Ax \geq b\}$ der zulässigen Lösungen von (QP) beschränkt und nichtleer ist. Die Beweise orientieren sich zum Teil an den Resultaten von Tseng und Bertsekas in [37], wo die Konvergenz eines verwandten Verfahrens, des Exponential–Multiplikatorverfahrens (vgl. Abschnitt 2.2), untersucht wird, sind ansonsten aber neue Ergebnisse. Diese sind in den Abschnitten 5.4, 5.5 und 5.6 dargestellt.

Der Abschnitt 5.4 beschäftigt sich mit der Konvergenz der dualen Folge $(\lambda^\nu)_\nu$. Es wird ein Zusammenhang zwischen der vom Algorithmus erzeugten dualen Folge und dem dualen Problem hergestellt. Dabei stellt sich heraus, daß die duale Folge gegen eine Optimallösung dieses Problems konvergiert.

Im darauffolgenden Abschnitt wird dann die primäre Folge $(x^\nu)_\nu$ auf Konvergenz untersucht. Dabei werden hinreichende Kriterien für die Konvergenz bewiesen.

Der letzte Abschnitt des Kapitels beschäftigt sich mit der Konvergenzrate der dualen Folge, wenn die Modifizierte Barrieremethode auf das lineare Programm (LP) angewendet wird. Auch die Resultate, die in diesem Abschnitt bewiesen werden, sind eigene Ergebnisse.

5.1 Polyaks Konvergenzresultate

In diesem Abschnitt beweisen wir, aufbauend auf den Ergebnissen der Abschnitte 3.3 und 3.4, die Konvergenzresulate von Polyak aus [27] (für den Spezialfall des konvexen quadratischen Programms (QP)) und [28] (für das lineare Programm (LP)).

Es gilt der folgende

Satz 5.1.1 *Es sei entweder $Q = 0$ und es gelte die Voraussetzung 1 oder es sei $Q \neq 0$ und die Voraussetzung 2 erfüllt. Weiterhin sei die Menge S der zulässigen Lösungen von* (QP) *beschränkt. Man wähle im Algorithmus 4.1.1*

$$\begin{array}{rcl} \lambda^0 & := & e \\ k^\nu & := & \kappa \quad \text{für } \nu = 0, 1, 2, \ldots . \end{array}$$

Zu jedem $0 < \gamma < 1$ existiert dann ein $\bar{\kappa} > 0$, so daß für alle $\kappa > \bar{\kappa}$ folgende Aussagen richtig sind:

(i) *Die primäre Folge konvergiert gegen x^*: $\lim_{\nu \to \infty} x^\nu = x^*$.*

(ii) *Die duale Folge konvergiert gegen λ^*: $\lim_{\nu \to \infty} \lambda^\nu = \lambda^*$.*

(iii) *Für alle ν gilt die Abschätzung*

$$\max \left\{ \|x^\nu - x^*\|_\infty, \|\lambda^{\nu+1} - \lambda^*\|_\infty \right\} \leq \gamma \|\lambda^\nu - \lambda^*\|_\infty .$$

Beweis: Der Beweis des Satzes ergibt sich für $Q = 0$ durch Anwendung von Satz 3.4.1 und 3.3.1 und für $Q \neq 0$ aus Satz 3.4.2 und 3.3.3, falls $Q \neq 0$:

Wir zeigen die Behauptung (iii) des Satzes durch Induktion nach ν. Aus technischen Gründen nehmen wir noch die Behauptung

(iv) $(\lambda^{\nu+1}, \kappa) \in D(\lambda^*, k_0, \delta, \varepsilon)$

hinzu. Dabei ist $D(\lambda^*, k_0, \delta, \varepsilon)$ die in Abschnitt 3.3 auf Seite 34 definierte Menge und k_0, δ, ε die Konstanten aus den Sätzen 3.3.1 und 3.3.3.

Nach Satz 3.4.1 bzw. 3.4.2 können wir ein $\bar{\kappa} > k_0 > 0$ finden, so daß folgende Aussagen richtig sind:

(P1) Für alle $\kappa > \bar{\kappa}$ existiert ein eindeutiges

$$\begin{aligned} x(\kappa) &= \arg\ \min\{\mathcal{M}(x,\kappa) : x \in \mathbb{R}^n\} \\ &= \arg\ \min\{\mathcal{F}(x,e,\kappa) : x \in \mathbb{R}^n\} \\ &= \arg\ \min\{\mathcal{F}(x,\lambda^0,\kappa) : x \in \mathbb{R}^n\} \end{aligned}$$

und es gilt $\nabla_x \mathcal{F}(x, \lambda^0, \kappa) = 0$.

(P2) Es gilt für $x(\kappa)$ und $\lambda(\kappa) = \Delta(x(\kappa), \kappa) \cdot e$ die Abschätzung

$$\max\{\|x(\kappa) - x^*\|_\infty, \|\lambda(\kappa) - \lambda^*\|_\infty\} \le \min\left\{\kappa\delta, \frac{\omega}{\kappa}\right\} \|e - \lambda^*\|_\infty.$$

(P3) Aus $a_j^T x^* = b_j$ folgt: $\lambda_j(k) > \varepsilon$ (denn für diese Indizes gilt: $\lambda_j^* \ge \sigma > \varepsilon$).

Dabei ist $\omega = \max\{c_0, c_3\}$ bzw. $\omega = \max\{c_1, c_4\}$. Wählt man $\bar{\kappa} > 0$ groß genug, so gilt $\frac{\omega}{\kappa} < \frac{1}{2}$ und $\frac{\omega}{\kappa} < \kappa\delta$ für alle $\kappa > \bar{\kappa}$, so daß die Ungleichung

$$0 < \min\left\{\kappa\delta, \frac{\omega}{\kappa}\right\} \le \gamma \quad \text{für alle } \kappa > \bar{\kappa} \tag{5.1}$$

richtig ist. Dies zeigt, daß für $\nu = 0$ und $x^\nu = x(\kappa)$ und $\lambda^{\nu+1} = \Delta(x(\kappa), \kappa) \cdot e$ gilt:

$$\max\left\{\|x^0 - x^*\|_\infty, \|\lambda^1 - \lambda^*\|_\infty\right\} \le \gamma\|e - \lambda^*\|_\infty = \gamma\|\lambda^0 - \lambda^*\|_\infty.$$

Aus (P2) und (P3) folgt $(\lambda^1, \kappa) \in D(\lambda^*, k_0, \delta, \varepsilon)$. Dies verifiziert die Behauptung (iii) für $\nu = 0$.

Wir nehmen nun an, daß die Behauptung (iii) und (iv) für ein $\nu \ge 0$ bereits bewiesen sind. Nach (iv) gilt dann: $\lambda^{\nu+1} \in D(\lambda^*, k_0, \delta, \varepsilon)$. Daher besagen dann Satz 3.3.1 und Satz 3.3.3, daß für $x^{\nu+1} = \arg\ \min\{\mathcal{F}(x, \lambda^{\nu+1}, \kappa : x \in \mathbb{R}^n\}$ und $\lambda^{\nu+2} = \Delta(x^{\nu+1}, \kappa)^{-1}\lambda^{\nu+1}$ die Abschätzung

$$\max\left\{\|x^{\nu+1} - x^*\|_\infty, \|\lambda^{\nu+2} - \lambda^*\|_\infty\right\} \le \gamma\|\lambda^{\nu+1} - \lambda^*\|_\infty$$

richtig ist und

$$(\lambda^{\nu+2}, \kappa) \in D(\lambda^*, k_0, \delta, \varepsilon)$$

gilt. Daher gelten alle Aussagen auch für $\nu + 1$ und der Induktionsbeweis ist damit vollständig. Die Aussagen (i) und (ii) sind nun eine sofortige Konsequenz aus (iii). □

Der letzte Satz betrachtete den Fall, daß die Folge $(k^\nu)_\nu$ der Barriereparameter konstant ist. Divergieren die Barriereparameter hingegen gegen $+\infty$, so erhält man statt der linearen Konvergenz sogar superlineare Konvergenz:

Satz 5.1.2 *Unter den Voraussetzungen von Satz 5.1.1 wähle man im Algorithmus 4.1.1*

$$\lambda^0 := e$$

und eine Folge $(k^\nu)_\nu$ *von Barriere–Parametern mit* $0 < \kappa \leq k^0 < k^1 < k^2 < \cdots$ *sowie*

$$\lim_{\nu\to\infty} k^\nu = +\infty.$$

Dann gibt es ein $\bar{\kappa} > 0$*, so daß für alle* $\kappa > \bar{\kappa}$ *folgende Aussagen richtig sind:*

(i) *Die primäre Folge konvergiert gegen* x^**:* $\lim_{\nu\to\infty} x^\nu = x^*$*.*

(ii) *Die duale Folge konvergiert gegen* λ^**:* $\lim_{\nu\to\infty} \lambda^\nu = \lambda^*$*.*

(iii) *Für alle* ν *gilt die Abschätzung*

$$\max\left\{\|x^\nu - x^*\|_\infty, \|\lambda^{\nu+1} - \lambda^*\|_\infty\right\} \leq \gamma_1 \cdot \gamma_2 \cdots \gamma_\nu \cdot \|\lambda^\nu - \lambda^*\|_\infty$$

mit $\lim_{\nu\to\infty} \gamma_\nu = 0$*.*

Beweis: Vollkommen analog zum Beweis von Satz 5.1.1. □

5.2 Powells Ergebnisse für lineare Programme

Wie bereits weiter oben angekündigt, werden in diesem Abschnitt die Ergebnisse von Powell aus [29] zitiert. Diese Resultate beziehen sich auf die Anwendung der Modifizierten Barrieremethode mit *konstantem Barriereparameter*, d.h. $k^\nu = \kappa$ für alle ν, auf das lineare Programm

$$\text{(LP)} \quad \begin{array}{ll} \min & c^T x \\ x \in \mathbb{R}^n : & Ax \geq b \end{array} \quad .$$

Powells Konvergenzresultate benötigen deutlich schwächere Voraussetzungen als die entsprechenden Ergebnisse von Polyak für lineare Programme. Zunächst zeigt Powell, daß folgender Satz richtig ist:

Satz 5.2.1 *Sei die Menge* S *der zulässigen Lösungen des linearen Programms* (LP) *nichtleer und beschränkt. Im Algorithmus 4.1.1 sei* $\lambda^0 > 0$ *beliebig und* $k^\nu = \kappa$ *für alle* ν*, wobei* $\kappa > 0$ *beliebig sei. Dann konvergiert die primäre Folge* $(x^\nu)_\nu$ *gegen eine Optimallösung* $x^* \in S^*$ *von* (LP)*.*

Beweis: [29, Theorem 5.1] □

Powell betrachtet auch die vom Algorithmus gelieferte duale Folge $(\lambda^\nu)_\nu$ und zeigt ebenfalls deren Konvergenz. Sei dazu $\mathcal{J}^* = \{j : a_j^T x = b_j \text{ für alle } x \in S^*\}$ die Menge der Indizes derjenigen Nebenbedingungen, die bei jeder Optimallösung aktiv sind. Es gilt dann der

Satz 5.2.2 *Unter den Voraussetzungen von Satz 5.2.1 konvergiert die Folge $(\lambda_j^\nu)_\nu$ für $j \in \mathcal{J}^*$ gegen einen positiven Grenzwert. Für $j \notin \mathcal{J}^*$ konvergiert $(\lambda_j^\nu)_\nu$ gegen 0 mit der (Q–)linearen Konvergenzrate:*

$$\lim_{\nu\to\infty} \frac{\lambda_j^{\nu+1}}{\lambda_j^\nu} = \Big(\kappa \cdot (a_j^T x^* - b_j) + 1\Big)^{-1}.$$

Dabei sei x^* hier diejenige Optimallösung von (LP), *gegen die die primäre Folge nach Satz 5.2.1 konvergiert.*

Beweis: [29, Theorem 5.2] □

Letztendlich leitet Powell auch gewisse Aussagen über die Konvergenzrate des Verfahrens her. Dabei benutzt er, daß für $\nu = 0, 1, \ldots,$ gilt:

$$c^T x^\nu - c^T x^* = \sum_{j=1}^m \lambda_j^{\nu+1} \left(a_j^T x^\nu - a_j^T x^*\right).$$

Die letzte Gleichung ergibt sich z.B. als Spezialfall der Bemerkung 4.1.3 dieser Arbeit (mit $Q = 0$). Wegen der Zulässigkeit von x^* gilt dann

$$c^T x^\nu - c^T x^* \leq \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j). \tag{5.2}$$

Powell beweist, daß die rechte Seite in (5.2) linear gegen Null konvergiert. Letztendlich zeigen seine Ergebnisse, daß auch die maximale Verletzung der Nebenbedingungen

$$\max_{1\le j\le m} \max\left\{0, b_j - a_j^T x^\nu\right\}$$

linear gegen Null strebt.

5.3 Einige Resultate aus der konvexen Analysis

Wegen der positiven Semidefinitheit von Q ist (QP) ein konvexes Optimierungsproblem. Wir schreiben es noch einmal in der üblichen Standardform für konvexe Probleme als

$$\begin{array}{lll} \text{(QP)} & \min & q(x) \equiv \frac{1}{2}x^T Q x + c^T x \\ & x \in \mathbb{R}^n : & b_j - a_j^T x \leq 0, \qquad j = 1, \ldots, m. \end{array} \tag{5.3}$$

Wie bereits in der Einleitung des Kapitels erwähnt, werden wir bei der Konvergenzanalyse voraussetzen, daß die Menge S^* der Optimallösungen von (QP) nichtleer und beschränkt ist. Diese Voraussetzung ist notwendig, um die Definiertheit des Verfahrens zu sichern. Wie wir in Abschnitt 3.2 gesehen haben, kann es sonst vorkommen, daß die Modifizierte Barrierefunktion kein endliches Minimum besitzt.

Setzen wir voraus, daß S^* nichtleer und beschränkt ist, dann existiert insbesondere eine Optimallösung von (QP) und der Satz von Kuhn und Tucker für konvexe Programme (siehe etwa [36, Theorem 6.7.8]) ist anwendbar. Dieser besagt, daß die Lagrangefunktion L aus (1.1) einen Sattelpunkt besitzt.

Um die Konvergenzanalyse durchzuführen, benötigen wir einige Definitionen und Ergebnisse aus der konvexen Analysis. Es erweist sich dabei als hilfreich, den Begriff einer konvexen bzw. konkaven Funktion dahingehend auszudehnen, daß auch die uneigentlichen Funktionswerte $+\infty$ und $-\infty$ zugelassen sind.

Definition 5.3.1 ([36, S.141]) *Eine Funktion $f : \mathbb{R}^m \to [-\infty, +\infty]$ heißt <u>konvex</u> (auch: <u>konvex im erweiterten Sinne</u>), falls die Ungleichung*

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y), \tag{5.4}$$

für alle $x, y \in \mathbb{R}^m$ und $0 \leq t \leq 1$ erfüllt ist, für welche die rechte Seite in (5.4) keinen unbestimmten Ausdruck der Form $+\infty + (-\infty)$ darstellt.

Eine Funktion $g : \mathbb{R}^m \to [-\infty, +\infty]$ heißt <u>konkav</u> (auch: <u>konkav im erweiterten Sinne</u>), falls $-f$ konvex ist.

Die letzte Definition zeigt auch zugleich einen kritischen Punkt bei der Erweiterung auf Funktionen mit den Funktionswerten $+\infty$ und $-\infty$: Es können unbestimmte Ausdrücke der Form $\infty - \infty$ auftreten. Bei einer bestimmte Teilmenge der konvexen (bzw. konkaven) Funktionen, den *eigentlichen* Funktionen, tritt diese Problematik nicht auf.

Definition 5.3.2 ([31, S.24]) *Eine konvexe Funktion $f : \mathbb{R}^m \to [-\infty, +\infty]$ heißt <u>eigentlich</u>, falls $f(y) < +\infty$ für mindestens ein $y \in \mathbb{R}^m$ und $f(y) > -\infty$ für alle $y \in \mathbb{R}^m$ gilt. Eine konkave Funktion $g : \mathbb{R}^m \to [-\infty, +\infty]$ wird <u>eigentlich</u> genannt, falls $-g$ eigentlich ist.*

Definition 5.3.3 ([31, S.52]) *Eine Funktion $f : \mathbb{R}^m \to [-\infty, +\infty]$ heißt <u>unterhalbstetig</u> in $y \in \mathbb{R}^m$, falls*

$$f(y) = \liminf_{v \to y} f(v) = \lim_{\varepsilon \to 0+} \left(\inf\{f(v) : \|y - v\| \leq \varepsilon\} \right)$$

gilt. Dies ist genau dann der Fall, falls für jede Folge $(y^\nu)_\nu$ *mit* $\lim_{\nu\to\infty} y^\nu = y$*, so daß* $\lim_{\nu\to\infty} f(y^\nu)$ *existiert* ($-\infty$ *und* $+\infty$ *eingeschlossen), gilt:*

$$f(y) \leq \lim_{\nu\to\infty} f(y^\nu).$$

Man nennt f *oberhalbstetig an* $y \in \mathbb{R}^m$*, wenn die Beziehung*

$$f(y) = \limsup_{v\to y} f(v) = \lim_{\varepsilon\to 0+} (\sup\{f(v) : \|y - v\| \leq \varepsilon\})$$

erfüllt ist. Offenbar ist f *genau dann oberhalbstetig an* y*, wenn* $-f$ *unterhalbstetig an* y *ist.*

Definition 5.3.4 ([31, S. 23]) *Sei* $f : \mathbb{R}^m \to [-\infty, +\infty]$ *eine (nicht notwendigerweise konvexe oder konkave) Funktion. Die Menge*

$$\mathrm{epi}(f) := \left\{ \begin{pmatrix} y \\ t \end{pmatrix} \in \mathbb{R}^{m+1} : y \in \mathbb{R}^m, t \geq f(y) \right\}.$$

nennt man den Epigraphen von f*.*

Eine Funktion $f : \mathbb{R}^m \to [-\infty, +\infty]$ ist genau dann konvex auf $\mathbb{R}^m$, falls $\mathrm{epi}(f)$ eine konvexe Menge des $\mathbb{R}^{m+1}$ ist ([31, S. 23-25]). Zu einer beliebigen Funktion $f : \mathbb{R}^m \to [-\infty, +\infty]$ existiert eine größte unterhalbstetige Funktion, die von f punktweise majorisiert wird, nämlich die Funktion, deren Epigraph der Abschluß des Epigraphen von f im $\mathbb{R}^{m+1}$ ist (siehe [31, S. 52]). Diese Funktion nennt man die unterhalbstetige Hülle von f.

Definition 5.3.5 ([31, S. 52]) *Der Abschluß* $\mathrm{cl}\, f$ *einer konvexen Funktion* $f : \mathbb{R}^m \to [-\infty, +\infty]$ *ist definiert als die unterhalbstetige Hülle von* f*, falls* f *nirgends den Wert* $-\infty$ *annimmt. Ist* $f(y) = -\infty$ *für ein* $y \in \mathbb{R}^m$*, dann definiert man den Abschluß als die Funktion, die konstant* $-\infty$ *ist. Der Abschluß von* f *wird mit* $\mathrm{cl}\, f$ *bezeichnet.*

Eine konvexe Funktion f *heißt abgeschlossen, falls* $\mathrm{cl}\, f = f$*.*

Der Abschluß $\mathrm{cl} f$ einer konvexen Funktion f ist wieder eine konvexe Funktion (siehe [31, S. 52]). Wie in [31, S. 52] bemerkt wird, sind für eine eigentliche konvexe Funktion die Begriffe der Abgeschlossenheit und der Unterhalbstetigkeit äquivalent. Ferner sind die einzigen abgeschlossenen uneigentlichen konvexen Funktionen die konstanten Funktionen $+\infty$ und $-\infty$. Es gilt nun der folgende

Satz 5.3.6 *Sei* $\{f_i\}_{i\in I}$ *eine Familie konvexer eigentlicher Funktionen und sei*

$$f(y) := \sup_{i\in I} f_i(y)$$

das punktweise Supremum der f_i*. Dann gilt:*

(i) *f ist eine konvexe Funktion.*

(ii) *Ist f für mindestens ein y endlich und jedes f_i abgeschlossen, so ist f abgeschlossen und eigentlich.*

Beweis:

(i) [31, Theorem 5.5]

(ii) [31, Theorem 9.4] □

Das sogenannte duale Funktional d zu (QP) ist durch

$$d(\lambda) = \inf_{x \in \mathbb{R}^n} L(x, \lambda) \tag{5.5}$$

gegeben. Dabei ist L wieder die gewöhnliche Lagrangefunktion (1.1) zum konvexen Optimierungsproblem (QP). Dann lautet das duale Problem zu (QP)

$$\text{(D)} \qquad \max_{\lambda \geq 0} \quad d(\lambda) \tag{5.6}$$

Oft findet man in der Literatur auch die Bezeichnungen Lagrange–duales Funktional für d und Lagrange–duales Problem für (D).

Ich möchte an dieser Stelle zunächst wieder explizit auf den linearen Fall, d.h. $Q = 0$ zu sprechen kommen. Hier besitzen sowohl das duale Funktional als auch das duale Problem eine besonders einfache Form. Nach Definition von d und der Lagrange–Funktion L in (1.1) folgt sofort, daß $d(\lambda) = -\infty$, falls $\lambda \not\geq 0$. Betrachten wir d also im folgenden für $\lambda \geq 0$. Zunächst ist dann

$$d(\lambda) = \inf_{x \in \mathbb{R}^n} L(x, \lambda) = \inf_{x \in \mathbb{R}^n} c^T x - \lambda^T (Ax - b).$$

Ist daher $A^T \lambda = c$, so gilt für *jedes* $x \in \mathbb{R}^n$, daß

$$L(x, \lambda) = c^T x - x^T \underbrace{(A^T \lambda)}_{=c} + b^T \lambda = c^T x - c^T c + b^T \lambda = b^T \lambda.$$

Folglich besitzt für diese λ das duale Funktional den Wert $d(\lambda) = b^T \lambda$. Man betrachte nun den Fall, daß $A^T \lambda \neq c$. Offenbar ist $L(x, \lambda)$ zu festem λ eine *lineare* Funktion von x. Der Gradient ist konstant und lautet $\nabla_x L(x, \lambda) = c - A^T \lambda \neq 0$. Daher ist $L(x, \lambda)$ als Funktion von x linear und nicht konstant, d.h. *unbeschränkt* auf $\mathbb{R}^n$. Damit folgt für diese λ, daß gilt: $d(\lambda) = -\infty$. Zusammenfassend gilt dann also

$$d(\lambda) = \begin{cases} b^T \lambda & \text{falls } A^T \lambda = c \text{ und } \lambda \geq 0 \\ -\infty & \text{sonst} \end{cases}$$

und das duale Problem (D) reduziert sich auf das klassische zu (LP) duale lineare Programm (LPD).

Auch für $Q \neq 0$ existiert ein Zusammenhang zwischen dem dualen Problem (D) und dem dualen quadratischen Programm (QPD) aus dem Satz 1.2.1 von Dorn. Sei dazu $\lambda \in \mathbb{R}^m$ und wir betrachten $d(\lambda)$. Gilt $\lambda \not\geq 0$, so folgt wieder $d(\lambda) = -\infty$ nach Definition der Lagrangefunktion L in (1.1). Sei daher im folgenden $\lambda \geq 0$. Die Funktion $L(.,\lambda)$ ist konvex. Existiert ein $\bar{x}$ mit

$$0 = \nabla_x L(\bar{x}, \lambda) = Q\bar{x} + c - A^T\lambda, \tag{5.7}$$

so folgt deshalb dann

$$d(\lambda) = \inf_{x\in\mathbb{R}^n} L(x,\lambda) = L(\bar{x},\lambda) = -\frac{1}{2}\bar{x}^T Q\bar{x} + b^T\lambda. \tag{5.8}$$

Ist $\hat{x}$ eine weitere Nullstelle des Gradienten $\nabla_x L(.,\lambda)$, so folgt aus (5.7) und

$$Q\hat{x} + c - A^T\lambda = 0$$

dann $Q\bar{x} = Q\hat{x}$, also

$$\bar{x}^T Q\bar{x} = \bar{x}^T Q\hat{x} = \left(\bar{x}^T Q\hat{x}\right)^T = \hat{x}^T Q^T\bar{x} = \hat{x}^T Q\bar{x} = \hat{x}^T Q\hat{x}.$$

Dies zeigt, daß der Ausdruck auf der rechten Seite von (5.8) *unabhängig* von der Wahl der Nullstelle $\bar{x}$ des Gradienten ist.

Es bleibt nun noch der Fall zu betrachten, daß $Qx + c - A^T\lambda \neq 0$ für alle $x \in \mathbb{R}^n$ gilt. In diesem Fall kann Q nicht invertierbar sein, denn wäre Q invertierbar, so wäre insbesondere $\bar{x} := Q^{-1}(A^T\lambda - c)$ Nullstelle des Gradienten von $L(.,\lambda)$. Ferner kann nicht $A^T\lambda = c$ gelten, denn sonst hätte der Gradient in $\bar{x} := 0$ eine Nullstelle.

Da Q symmetrisch ist, besagen elementare Resulate der Linearen Algebra, daß Kern $Q = \{x : Qx = 0\}$ und Bild $Q = \{Qx : x \in \mathbb{R}^n\}$ orthogonale Komplemente im $\mathbb{R}^n$ sind. Daher besitzt jedes $y \in \mathbb{R}^n$ eine eindeutige Darstellung als $y = y' + y''$ mit $y' \in$ Kern Q, $y'' \in$ Bild Q und $(y')^T y'' = 0$. Wie oben bereits festgestellt wurde, ist Q nicht invertierbar, besitzt also insbesondere einen nichttrivialen Kern. Sei $y \in \mathbb{R}^n$ beliebig und $y = y' + y''$ mit $y' \in$ Kern Q und $y'' \in$ Bild Q die eindeutige Orthogonalzerlegung von y. Gilt $y^T x = 0$ für alle $x \in$ Kern Q, so folgt dann $y' = 0$ und daher $y = y'' \in$ Bild Q.

Wir behaupten, daß es ein $\hat{x} \neq 0$ mit $Qx = 0$ und $\hat{x}^T(c - A^T\lambda) \neq 0$ gibt. Gäbe es kein solches $\hat{x}$, so wäre $x^T(c - A^T\lambda) = 0$ für alle $x \in$ Kern Q, und somit nach dem letzten Paragraphen $c - A^T\lambda \in$ Bild Q. Dies steht aber im Widerspruch zu $Qx + c - A^T\lambda \neq 0$ für alle $x \in \mathbb{R}^n$.

Sei also $\hat{x} \in$ Kern Q mit O.B.d.A. $\hat{x}^T(c - A^T\lambda) > 0$. Sei $t \in \mathbb{R}$. Dann gilt

$$L(t\hat{x}, \lambda) = b^T\lambda + t\,\underbrace{\hat{x}^T(c - A^T\lambda)}_{>0}$$

und es gilt $\lim\limits_{t \to -\infty} L(t\hat{x}, \lambda) = -\infty$. Daher folgt $d(\lambda) = -\infty$.

Wir fassen unsere Ergebnisse über das duale Funktional zusammen. Es gilt:

$$d(\lambda) = \begin{cases} -\frac{1}{2}x^T Q x + b^T \lambda, & \text{falls } \lambda \geq 0 \text{ und es ein } x \text{ gibt mit } A^T\lambda - Qx = c \\ -\infty & \text{sonst.} \end{cases}$$

Damit kann also auch im Fall $Q \neq 0$ das duale Problem (D) mit dem dualen quadratischen Programm identifiziert werden. Allerdings ist hier der Zusammenhang nicht ganz so schön wie im linearen Fall, da das duale Funktional d nur ein Argument, die Zielfunktion η von (QPD) jedoch zwei Argumente besitzt.

In [14] werden die folgenden beiden Sätze gezeigt, deren Ergebnisse für die weitere Beweisführung benötigt werden:

Satz 5.3.7 (Schwacher Dualitätssatz) *Sei* (P) *das folgende Optimierungsproblem*

$$\begin{array}{lll} \text{(P)} & \min & f(x) \\ & x \in \mathbb{R}^n : & g_j(x) \leq 0, \quad j = 1, \ldots, m \end{array}$$

mit zugehöriger Lagrangefunktion $L_{(P)}(x, \lambda) = f(x) + \sum\limits_{j=1}^{m} \lambda_j g_j(x)$. *Sei das duale Funktional* $d_{(P)}$ *durch* $d_{(P)}(\lambda) = \inf\limits_{x \in \mathbb{R}^n} L(x, \lambda)$ *und das zu* (P) *duale Optimierungsproblem durch*

$$\text{(DP)} \qquad \max_{\lambda \geq 0} d_{(P)}(\lambda)$$

definiert. Ist x *ein zulässiger Punkt von* (P) *und* λ *ein zulässiger Punkt von* (DP) *(d.h.* $\lambda \geq 0$*), so gilt*

$$f(x) \geq d_{(P)}(\lambda).$$

Beweis: [14, Satz 3.12]. $\square$

Satz 5.3.8 (Starker Dualitätssatz) *Folgende Aussagen sind äquivalent:*

(i) *Die Lagrangefunktion* $L_{(P)}(x, \lambda)$ *zu* (P) *besitzt einen Sattelpunkt* $(\bar{x}, \bar{\lambda}) \in \mathbb{R}^n \times \mathbb{R}^m_+$.

(ii) $\bar{x}$ *ist Optimallösung von* (P) , $\bar{\lambda}$ *ist Optimallösung von* (DP) *und* $f(\bar{x}) = d_{(P)}(\bar{\lambda})$.

Beweis: [14, Satz 3.13]. $\square$

Da (QP) ein konvexes Optimierungsproblem ist und unsere Voraussetzungen besagen, daß (QP) mindestens eine Optimallösung besitzt, wissen wir nach dem Satz von Kuhn und Tucker, daß die Lagrangefunktion L zu (QP) einen Sattelpunkt besitzt. Daher folgt, daß die zweite Aussage in Satz 5.3.8 richtig ist. Insbesondere besitzt (D) eine Optimallösung $\bar{\lambda}$ und $d(\bar{\lambda}) = q^* > -\infty$ ist endlich. Es ergibt sich aus Satz 5.3.8 außerdem folgendes Korollar:

Korollar 5.3.9 *Sei $x^* \in S^*$ eine beliebige Optimallösung von* (QP) *und λ^* ein beliebiger zugehöriger optimaler Lagrangevektor. Dann ist λ^* eine Optimallösung von* (D).

Beweis: Da (x^*, λ^*) ein Karush–Kuhn–Tucker Paar für (QP) ist, folgt aus (1.2a) die Beziehung

$$\nabla_x L(x^*, \lambda^*) = Qx^* + c - A^T\lambda^* \stackrel{(1.2a)}{=} 0.$$

Wegen der Konvexität von $L(., \lambda^*)$ impliziert dies, daß

$$L(x^*, \lambda^*) = \inf_{x\in\mathbb{R}^n} L(x, \lambda^*) = d(\lambda^*).$$

Dann ist aber

$$d(\lambda^*) = L(x^*, \lambda^*) = q(x^*) - \sum_{j=1}^{m} \lambda_j^*(b_j - a_j^T x^*) \stackrel{(1.2c)}{=} q(x^*),$$

und somit λ^* wegen Satz 5.3.8 eine Optimallösung von (D). □

Wir werden nun die Ergebnisse aus Satz 5.3.6 auf das duale Funktional d anwenden. Dazu zeigen wir zunächst das folgende

Lemma 5.3.10 *Für festes $x \in \mathbb{R}^n$ ist $-L(x, \lambda)$ als Funktion von λ konvex, eigentlich und unterhalbstetig.*

Beweis: Die Konvexität von $-L(x, .)$ ist trivial. Es gilt ferner $-L(x, \lambda) > -\infty$ nach Definition für jedes $\lambda \in \mathbb{R}^m$. Für jedes $\lambda \geq 0$ ist außerdem $-L(x, \lambda) < +\infty$, d.h. $L(x, .)$ ist eine eigentliche Funktion. Sei $\lambda \in \mathbb{R}^m$ fest. Wir zeigen nun, daß $L(x, .)$ an λ unterhalbstetig ist. Dazu sei $(y^\ell)_\ell$ eine Folge, die gegen λ konvergiert. Wir unterscheiden zwei Fälle:

(i) $\lambda \not\geq 0$: Dann ist für genügend großes ℓ auch $y^\ell \not\geq 0$ und somit $-L(x, y^\ell) = +\infty = -L(x, \lambda)$ für alle genügend großen ℓ. Es folgt $-L(x, \lambda) = \lim_{\ell\to\infty} -L(x, y^\ell)$.

(ii) $\lambda \geq 0$: Entweder gilt $y^\ell \geq 0$ für alle großen ℓ oder $y^\ell \not\geq 0$ für unendlich viele ℓ. Im ersten Fall ersieht man sofort, daß $\lim_{\ell\to\infty} -L(x, y^\ell) = -L(x, \lambda)$ gilt. Im zweiten Fall gilt für jedes $N > 0$, daß $\inf\{-L(x, y^\ell) : \ell \geq N\} = -L(x, \lambda)$, da jede Umgebung um λ Folgenglieder y^ℓ mit $-L(x, y^\ell) = +\infty$ enthält. Somit ist $-L(x, \lambda) \leq \varliminf_{\ell \to \infty} -L(x, y^\ell)$. $\square$

Wie schon erwähnt, werden wir die Ergebnisse aus Satz 5.3.6 auf das duale Funktional d anwenden. Nach Lemma 5.3.10 ist $\{-L(x, .)\}_{x \in \mathbb{R}^n}$ eine Familie konvexer, eigentlicher und unterhalbstetiger Funktionen. Wegen Satz 5.3.8 ist $d(\bar{\lambda})$ endlich, wobei $(\bar{x}, \bar{\lambda})$ ein (existierender) Sattelpunkt der Langrangefunktion L ist. Somit sind beide Resultate aus Satz 5.3.6 auf die Funktion

$$\hat{f}(\lambda) = \sup_{x \in \mathbb{R}^n} -L(x, \lambda) = -\inf_{x \in \mathbb{R}^n} L(x, \lambda) = -d(\lambda)$$

anwendbar. Es folgt, daß $\hat{f}$ konvex, eigentlich und unterhalbstetig ist. Daher ist d selbst konkav und oberhalbstetig. Offenbar gilt $d(\lambda) < +\infty$ für alle λ. Wir fassen die Eigenschaften von d zum Zweck der Refenerenz im folgendem Lemma zusammen:

Lemma 5.3.11 *Die Funktion d ist eine konkave, eigentliche und oberhalbstetige Funktion.*

Zuletzt benötigen wir noch die Definition des Subdifferentials.

Definition 5.3.12 *Das <u>Subdifferential</u> $\partial f(y)$ einer konvexen Funktion f an einem Punkt y ist durch*

$$\partial f(y) = \{v : f(u) \geq f(y) + v^T(u - y) \quad \forall u \in \mathbb{R}^m\}$$

definiert. Jeder Vektor $v \in \partial f(y)$ heißt ein <u>Subgradient</u> von f an der Stelle y.

Ist g eine konkave Funktion, so erklärt man das Subdifferential $\partial g(y)$ an y durch $\partial g(y) = -\partial(-g)(y)$. Folglich ist

$$\partial g(y) = \{v : g(u) \leq g(y) + v^T(u - y) \quad \forall u \in \mathbb{R}^m\}$$

Man nennt f (bzw. g) <u>subdifferenzierbar</u> an y, falls $\partial f(y)$ (bzw. $\partial g(y)$) nichtleer ist.

Aus der Definition des Subdifferentials folgt, daß x dann das globale Minimum einer konvexen Funktion f auf $\mathbb{R}^m$ ist, falls $0 \in \partial f(x)$. Ist nämlich $0 \in \partial f(x)$, so gilt für beliebiges $u \in \mathbb{R}^m$, daß $f(u) \geq f(x) + 0^T(u - x) = f(x)$, also $f(x) \leq f(u)$.

Bemerkung 5.3.13 *Ist eine konkave oder konvexe Funktion h an der Stelle y differenzierbar, so gilt $\partial h(y) = \{\nabla h(y)\}$.*

Einen Beweis für die letzte Bemerkung findet man z.B. in [31, Theorem 25.1].

5.4 Konvergenz der dualen Folge

In diesem Abschnitt soll die Konvergenz der dualen Folge $(\lambda^\nu)_\nu$ gegen eine Optimallösung des dualen Problems (D) gezeigt werden. Der Konvergenzbeweis gliedert sich grob in drei Schritte: Zuerst wird die Konvergenz der dualen Funktionswerte $(d(\lambda^\nu))_\nu$ bewiesen. Daraus wird dann im nächsten Schritt die Konvergenz der dualen Folge geschlossen. Zuletzt schließlich zeigen wir, daß die duale Folge tatsächlich gegen eine Optimallösung des dualen Problems konvergiert.

Lemma 5.4.1 *Für die Folge (x^ν, λ^ν), die vom Algorithmus 4.1.1 erzeugt wird, gilt:*

(i) $d(\lambda^{\nu+1}) = q(x^\nu) + \sum_{j=1}^m \lambda_j^{\nu+1}(b_j - a_j^T x^\nu)$,

(ii) $b - Ax^\nu \in \partial d(\lambda^{\nu+1})$,

(iii) $b_j - a_j^T x^\nu = \dfrac{1}{k^\nu}\left(1 - \dfrac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)$ *für* $j = 1, \ldots, m$.

Beweis: Wie bereits in Bemerkung 4.1.3 hergeleitet, gilt

$$0 = \nabla q(x^\nu) - \sum_{j=1}^m \lambda^{\nu+1} a_j.$$

Dies und die Konvexität von $L(., \lambda^{\nu+1})$ impliziert nun, daß in

$$d(\lambda^{\nu+1}) = \inf_{x \in \mathbb{R}^n} L(x, \lambda^{\nu+1})$$

das Minimum zumindest für $x = x^\nu$ angenommen wird. Nach Definition des dualen Funktionals in (5.5) folgt daher:

$$d(\lambda^{\nu+1}) = q(x^\nu) + \sum_{j=1}^m \lambda_j^{\nu+1}(b_j - a_j^T x^\nu).$$

Dies zeigt die erste Aussage des Lemmas. Durch die Abschätzung

$$\begin{aligned}
d(\lambda) &= \inf_{x \in \mathbb{R}^n} L(x, \lambda) \leq \inf_{x \in \mathbb{R}^n} \left\{ q(x) + \sum_{j=1}^m \lambda_j \left(b_j - a_j^T x\right) \right\} \\
&\leq q(x^\nu) + \sum_{j=1}^m \lambda_j \left(b_j - a_j^T x^\nu\right) \\
&= q(x^\nu) + \sum_{j=1}^m \lambda_j^{\nu+1} \left(b_j - a_j^T x^\nu\right) + \sum_{j=1}^m \left(\lambda_j - \lambda_j^{\nu+1}\right)\left(b_j - a_j^T x^\nu\right) \\
&= d(\lambda^{\nu+1}) + \sum_{j=1}^m \left(\lambda_j - \lambda_j^{\nu+1}\right)\left(b_j - a_j^T x^\nu\right),
\end{aligned}$$

also

$$d(\lambda) \leq d(\lambda^{\nu+1}) + (b - Ax^\nu)^T(\lambda - \lambda^{\nu+1}), \tag{5.9}$$

die für alle $\lambda \in \mathbb{R}^m$ gilt, ergibt sich nach Definition des Subdifferentials $b - Ax^\nu \in \partial\, d(\lambda^{\nu+1})$, womit die zweite Behauptung gezeigt ist. Schließlich erhalten wir noch aus der Update–Formel in Schritt (5) für $\lambda^{\nu+1}$ durch einfaches Umformen die Identität in Behauptung (iii). □

Mit Hilfe von Lemma 5.4.1 kann man nun zeigen, daß die duale Folge $(\lambda^\nu)_\nu$ eine wichtige Eigenschaft besitzt:

Lemma 5.4.2 *Die duale Folge $(\lambda^\nu)_\nu$ erfüllt*

$$\lambda^{\nu+1} \in \arg\max_{\lambda\geq 0}\left[d(\lambda) - \frac{1}{k^\nu}\sum_{j=1}^{m}\lambda_j^\nu\beta\left(\frac{\lambda_j}{\lambda_j^\nu}\right)\right], \tag{5.10}$$

wobei $\beta(t) = -\log t + t - 1$.

Beweis: Wir definieren die Funktion h durch

$$h(\lambda) = \begin{cases} -\frac{1}{k^\nu}\sum\limits_{j=1}^{m}\lambda_j^\nu\beta\left(\frac{\lambda_j}{\lambda_j^\nu}\right) & \text{falls } \lambda \geq 0 \\ -\infty & \text{sonst ,} \end{cases}$$

wobei wir die Konvention

$$-\lambda_j^\nu\beta\left(\frac{0}{\lambda_j^\nu}\right) := -\lim_{t\to 0+}\lambda_j^\nu\beta\left(\frac{t}{\lambda_j^\nu}\right) = -\infty$$

verwenden. Da β eine konvexe Funktion (einer rellen Variablen) ist, folgt, daß h nach Konstruktion konkav ist mit $h(\lambda) = -\infty$, falls $\lambda_{j_0} = 0$ für ein j_0 gilt. Die zu zeigende Aussage ist äquivalent zu

$$\lambda^{\nu+1} \in \arg\max_{\lambda\geq 0}\left[d(\lambda) + h(\lambda)\right]. \tag{5.11}$$

Weiterhin ist h für alle $\lambda > 0$ differenzierbar. Aus Bemerkung 5.3.13 ersieht man daher, daß für $\lambda > 0$ die Beziehung $\partial d(\lambda) = \{\nabla_\lambda d(\lambda)\}$ richtig ist.

Nach Lemma 5.4.1 (ii) ist $b - Ax^\nu \in \partial d(\lambda^{\nu+1})$. Die Aussage (iii) in Lemma 5.4.1 zeigt, daß dies äquivalent zu

$$\frac{1}{k^\nu}\begin{pmatrix} 1 - \lambda_1^\nu/\lambda_1^{\nu+1} \\ 1 - \lambda_2^\nu/\lambda_2^{\nu+1} \\ \vdots \\ 1 - \lambda_m^\nu/\lambda_m^{\nu+1} \end{pmatrix} \in \partial d(\lambda^{\nu+1})$$

ist. Nach den Rechenregeln für den Subgradienten (siehe [31, Section 23]) gilt daher

$$0 \in \partial d(\lambda^{\nu+1}) - \frac{1}{k^\nu} \begin{pmatrix} 1 - \lambda_1^\nu/\lambda_1^{\nu+1} \\ 1 - \lambda_2^\nu/\lambda_2^{\nu+1} \\ \vdots \\ 1 - \lambda_m^\nu/\lambda_m^{\nu+1} \end{pmatrix}. \tag{5.12}$$

Weiterhin gilt $\beta'(t) = 1 - \frac{1}{t}$ und somit

$$\frac{\partial}{\partial \lambda_j} \beta\left(\frac{\lambda_j}{\lambda_j^\nu}\right) = \frac{1}{\lambda_j^\nu}\left(1 - \frac{\lambda_j^\nu}{\lambda_j}\right). \tag{5.13}$$

Aus (5.12) und (5.13) folgt

$$\begin{aligned} 0 \in \; & \partial d(\lambda^{\nu+1}) - \frac{1}{k^\nu} \begin{pmatrix} \frac{\partial}{\partial \lambda_1}\left(\lambda_1^\nu \cdot \beta(\lambda_1/\lambda_1^\nu)\right) \\ \\ \frac{\partial}{\partial \lambda_2}\left(\lambda_2^\nu \cdot \beta(\lambda_2/\lambda_2^\nu)\right) \\ \\ \vdots \\ \frac{\partial}{\partial \lambda_m}\left(\lambda_m^\nu \cdot \beta(\lambda_m/\lambda_n^\nu)\right) \end{pmatrix}_{\lambda = \lambda^{\nu+1}} \\ = \; & \partial d(\lambda^{\nu+1}) + \partial h(\lambda^{\nu+1}). \end{aligned} \tag{5.14}$$

Sei

$$\hat{f}(\lambda) = -(d(\lambda) + h(\lambda)),$$

dann ist $\hat{f}$ eine konvexe Funktion, da d und h konkav sind. Nach [31, Theorem 23.8] ergibt sich daher mit (5.14)

$$0 \in -\partial d(\lambda^{\nu+1}) - \partial h(\lambda^{\nu+1}) \subset \partial \hat{f}(\lambda^{\nu+1}), \tag{5.15}$$

also insbesondere $\lambda^{\nu+1} \in \arg \min_{\lambda \in \mathbb{R}^m} \hat{f}(\lambda)$ (vgl. Seite 73). Es folgt $\lambda^{\nu+1} \in \arg \max_{\lambda \in \mathbb{R}^m} [d(\lambda) + h(\lambda)]$ und wegen $\lambda^{\nu+1} \geq 0$ damit (5.11). Dies zeigt die Behauptung des Lemmas. □

Die Ergebnisse des letzten Lemmas zeigen, daß man die duale Folge $(\lambda^\nu)_\nu$ gewissermaßen auch alternativ durch die Bedingung (5.10) gegeben betrachten kann. Wir verwenden hier „gewissermaßen", da diese Aussage in voller Strenge nicht ganz richtig ist. Während die Update–Formel in Schritt (5) des Algorithmus 4.1.1 den Vektor $\lambda^{\nu+1}$ für alle ν eindeutig festlegt, könnte es durchaus möglich sein, daß (5.10) keine *eindeutige* Lösung besitzt.

Um die Konvergenz der Folge $(\lambda^\nu)_\nu$ herzuleiten, untersuchen wir die Folge $(d(\lambda^\nu))_\nu$ und zeigen im folgenden Lemma zunächst, daß diese Folge der dualen Funktionswerte monoton und konvergent ist.

Lemma 5.4.3 *Für die Folge $(\lambda^\nu)_\nu$, die vom Algorithmus 4.1.1 geliefert wird, sind folgende Aussagen richtig:*

(i) *$d(\lambda^\nu) \leq d(\lambda^{\nu+1})$, d.h. $(d(\lambda^\nu))_\nu$ ist eine monoton steigende Folge.*

(ii) *Es existiert $d^\infty = \lim\limits_{\nu\to\infty} d(\lambda^\nu)$ und es gilt $d^\infty \leq q^*$.*

Beweis: Es gilt

$$0 \leq \left(\lambda_j^{\nu+1} - \lambda_j^\nu\right)\left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right), \quad j = 1, \dots, m, \tag{5.16}$$

da entweder beide Faktoren auf der rechten Seite von (5.16) nichtnegativ oder beide negativ sind. Der Ausdruck in der zweiten Klammer in (5.16) ist gleich $k^\nu(b_j - a_j^T x^\nu)$, wie wir aus Lemma 5.4.1 (iii) ersehen. Daraus ergibt sich, wenn man in (5.16) über j von 1 bis m summiert, die Ungleichung

$$0 \leq k^\nu(b_j - Ax^\nu)^T(\lambda^{\nu+1} - \lambda^\nu) \leq k^\nu\left(d(\lambda^{\nu+1}) - d(\lambda^\nu)\right),$$

wobei die letzte Ungleichung aus (5.9) mit $\lambda = \lambda^\nu$ folgt. Dies zeigt wegen $k^\nu > 0$ die erste Aussage des Lemmas. Um die zweite Behauptung zu beweisen, bemerken wir, daß nach dem schwachen Dualitätssatz für alle $\lambda \geq 0$ gilt: $d(\lambda) \leq q^*$. Damit ist insbesondere die Folge $(d(\lambda^\nu))_\nu$ nach oben beschränkt und die Behauptung folgt nun aus dem elementaren Ergebnis der Analysis, daß eine monoton wachsende und von oben beschränkte Folge reeller Zahlen konvergent ist. $\square$

Sei $M^\infty = \{\lambda \geq 0 : d(\lambda) \geq d^\infty\}$. Zunächst bemerken wir, daß M^∞ nichtleer ist. In der Tat, da $d^\infty \leq q^*$ und (D) nach dem Dualitätssatz 5.3.8 eine Optimallösung λ^* mit $d(\lambda^*) = q^* \geq d^\infty$ besitzt, folgt zumindest $\lambda^* \in M^\infty$. Für jedes $\lambda \in M^\infty$ ist $d(\lambda)$ endlich, denn nach Definition von M^∞ gilt: $d(\lambda) \geq d^\infty > -\infty$.

Um die Konvergenz der dualen Folge zu beweisen, benötigen wir eine geeignete Distanzfunktion. Dazu notieren wir die folgenden beiden Hilfsaussagen aus [37], deren Beweise wir der Vollständigkeit halber hier noch einmal aufführen.

Lemma 5.4.4 *Sei $\hat\delta : [0,\infty) \times (0,\infty) \to \mathbb{R}$ definiert durch*

$$\hat\delta(s,t) = s \log\left(\frac{s}{t}\right) - s + t, \tag{5.17}$$

wobei wir $0 \log 0 := 0$ setzen. Dann gilt:

(i) *$\hat\delta(s,t) = \psi^*(s/t)t$, wobei $\psi^*(s) = s\log s - s + 1$.*

(ii) $\hat{\delta}(s,t) = \psi^*(s) - \psi^*(t) - (\psi^*(t))'(s-t)$ *für alle* $s \geq 0$ *und alle* $t > 0$.

(iii) $\hat{\delta}(s,t) \geq 0$ *für alle* $s > 0$ *und* $t \geq 0$. *Ferner ist* $\hat{\delta}(s,t) = 0$ *genau dann, wenn* $s = t$.

(iv) *Ist* $s^\nu > 0$ $(\nu = 0,1,2,\ldots)$ *und* $s \geq 0$, *dann gilt* $\hat{\delta}(s,s^\nu) \to 0$ *genau dann, wenn* $s^\nu \to s$.

(v) *Für alle* $s \geq 0$ *hat* $\hat{\delta}(s,.)$ *beschränkte Niveaumengen.*

Beweis: (vgl. [37, Lemma 2.1])

(i) $\psi^*(s/t)t = t\left(\frac{s}{t}\log\left(\frac{s}{t}\right) - \frac{s}{t} + 1\right) = s\log\left(\frac{s}{t}\right) - s + t = \hat{\delta}(s,t)$.

(ii) Es gilt nach Definition von $\hat{\delta}$ und ψ^*

$$\begin{aligned} & \psi^*(s) - \psi^*(t) - (\psi^*(t))'(s-t) \\ = \; & s\log s - s + 1 - t\log t + t - 1 - (s-t)\log t \\ = \; & s(\log s - \log t) - s + t \\ = \; & s\log\left(\tfrac{s}{t}\right) - s + t \\ = \; & \hat{\delta}(s,t). \end{aligned}$$

(iii) Wegen $\nabla^2\psi^*(t) = \frac{1}{t}$ ist ψ^* für $t > 0$ strikt konvex. Damit folgt die Behauptung aus (ii).

(iv) Wir unterscheiden zwei Fälle: $s = 0$ und $s > 0$. Ist $s = 0$, so gilt wegen $s^\nu > 0$ dann $\hat{\delta}(s,s^\nu) = s^\nu$. Folglich gilt in diesem Fall $\hat{\delta}(s,s^\nu) \to 0$ genau dann, wenn $s^\nu \to 0 = s$. Ist $s > 0$, so ist $\hat{\delta}(s,.)$ als Funktion des zweiten Argumentes stetig und somit folgt $\hat{\delta}(s,s^\nu) \to 0$, falls $s^\nu \to s$. Die Umkehrung folgt aus der strikten Konvexität von ψ^*, der Stetigkeit von $\hat{\delta}(s,.)$ und der Darstellung in (ii).

(v) Sei s fest. Ist $s = 0$, so gilt $\hat{\delta}(s,t) = t$ für alle $t > 0$ und die Aussage ist trivial. Sei daher $s > 0$. Es gilt dann $\hat{\delta}(s,t) = s\log s - s + t - s\log t$. Wir müssen zeigen, daß, falls $t \to +\infty$ auch $\hat{\delta}(s,t) \to +\infty$ gilt. Dazu betrachten wir die Funktion $\varphi(t) = \frac{\log(st)}{t}$ für $t > 0$. Nach der L'Hospitalschen Regel gilt:

$$\lim_{t\to\infty}\varphi(t) = \lim_{t\to\infty}\frac{\log(t)}{t} = \lim_{t\to\infty}\frac{1/t}{1} = 0.$$

Mithin ist $\frac{\log(t)}{t} < \frac{1}{2s}$, also

$$\log(t) < \frac{1}{2s}t,$$

für großes t. Für diese t ist dann aber auch

$$\begin{aligned}\hat{\delta}(s,t) &= s\log\left(\tfrac{s}{t}\right) - s + t \\ &= s\log s - s\log t - s + t \\ &> s\log s - s\tfrac{1}{2s}t - s + t \\ &= s\log s - s + \tfrac{t}{2}\end{aligned}$$

und es ist nun klar, daß für t gegen $+infty$ auch $\hat{\delta}(s,t)$ gegen $+\infty$ strebt. □

Die Funktion $\hat{\delta}$ wird in der Literatur oft als Entropie–Funktion oder Entropie–Distanz bezeichnet. Mit Hilfe von $\hat{\delta}$ definiert man nun die Bregman–Distanz $\mathcal{D} : \mathbb{R}^m_+ \times \mathbb{R}^m_{++} \to [0,\infty)$ durch

$$\mathcal{D}(\lambda,\mu) = \sum_{j=1}^{m} \hat{\delta}(\lambda_j,\mu_j). \tag{5.18}$$

Lemma 5.4.5 *Für die Funktion $\mathcal{D}$ aus (5.18), gilt:*

(i) *$\mathcal{D}$ ist nichtnegativ*

(ii) *Ist $\hat{\lambda} \geq 0$ und $\lambda^\nu > 0$, $\nu = 1,2,\ldots$, dann gilt $\mathcal{D}(\hat{\lambda},\lambda^\nu) \to 0$ genau dann, wenn $\lambda^\nu \to \hat{\lambda}$.*

(iii) *Für jedes feste $\hat{\lambda} \geq 0$ hat die Funktion $\mathcal{D}(\hat{\lambda},.)$ beschränkte Niveaumengen.*

Beweis: Folgt unmittelbar aus der Definition von $\mathcal{D}$ und Lemma 5.4.4. □

Lemma 5.4.5 zeigt, daß man $\mathcal{D}$ tatsächlich als eine Art Distanzfunktion betrachten kann. Vor allem die Aussage (ii) wird für den Konvergenzbeweis der Folge $(\lambda^\nu)_\nu$ von entscheidender Wichtigkeit sein.

Es besteht ein wichtiger Zusammenhang zwischen der Funktion β aus Lemma 5.4.2 und der Entropie–Funktion $\hat{\delta}$. Es gilt für $s,t > 0$ die Beziehung

$$s\beta\left(\frac{t}{s}\right) = s\left(-\log\left(\frac{t}{s}\right) + \frac{t}{s} - 1\right) = s\log\left(\frac{s}{t}\right) - s + t = \hat{\delta}(s,t). \tag{5.19}$$

Dies zeigt wegen $\lambda^\nu > 0$ $(\nu = 1,2,\ldots)$ zusammen mit Lemma 5.4.2, daß

$$\lambda^{\nu+1} \in \arg\max_{\lambda\geq 0}\left[d(\lambda) - \frac{1}{k^\nu}\sum_{j=1}^{m}\hat{\delta}(\lambda^\nu_j,\lambda_j)\right], \tag{5.20}$$

oder

$$\lambda^{\nu+1} \in \arg\max_{\lambda\geq 0}\left[d(\lambda) - \frac{1}{k^\nu}\mathcal{D}(\lambda^\nu,\lambda)\right]. \tag{5.21}$$

Im folgenden Lemma beweisen wir, daß für jedes $\hat{\lambda} \in M^\infty$ die Bregman–Distanz $\mathcal{D}(\hat{\lambda},\lambda^\nu)$ monoton abnimmt.

Lemma 5.4.6 *Sei $\hat{\lambda} \in M^\infty$ beliebig. Dann ist die Folge $(\mathcal{D}(\hat{\lambda}, \lambda^\nu))_\nu$ monoton fallend.*

Beweis: Wir wissen, daß $b - Ax^\nu \in \partial d(\lambda^{\nu+1})$. Aus der Definition des Subdifferentials folgt daher, daß

$$(b - Ax^\nu)^T(\lambda^{\nu+1} - \hat{\lambda}) \le d(\lambda^{\nu+1}) - d(\hat{\lambda}) \le d(\lambda^{\nu+1}) - d^\infty \le 0, \tag{5.22}$$

wobei sich die letzte Ungleichung aus der Monotonie der Folge $(d(\lambda^\nu))_\nu$ ergibt. Man beachte, daß sowohl $d(\lambda^{\nu+1})$ als auch $d(\hat{\lambda})$ endlich sind, die Differenz $d(\lambda^{\nu+1}) - d(\hat{\lambda})$ in (5.22) also wohldefiniert ist. Wir benutzen nun wieder, daß wegen Lemma 5.4.1 (iii) $k^\nu(b_j - a_j^T x^\nu) = 1 - \lambda_j^\nu/\lambda_j^{\nu+1}$ gilt, und erhalten zusammen mit (5.22)

$$0 \ge (b - Ax^\nu)^T(\lambda^{\nu+1} - \hat{\lambda}) = \sum_{j=1}^m (b_j - a_j^T x^\nu)\cdot(\lambda_j^{\nu+1} - \hat{\lambda}_j) = \frac{1}{k^\nu}\sum_{j=1}^m \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\left(\lambda_j^{\nu+1} - \hat{\lambda}_j\right),$$

also wegen $k^\nu > 0$

$$0 \le \sum_{j=1}^m \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\left(\hat{\lambda}_j - \lambda_j^{\nu+1}\right). \tag{5.23}$$

Nun verwenden wir die Definition von $\mathcal{D}$. Wir wollen zeigen, daß

$$\mathcal{D}(\hat{\lambda}, \lambda^{\nu+1}) - \mathcal{D}(\hat{\lambda}, \lambda^\nu) \le 0$$

gilt. Wir haben

$$\begin{aligned}
&\mathcal{D}(\hat{\lambda}, \lambda^{\nu+1}) - \mathcal{D}(\hat{\lambda}, \lambda^\nu) \\
&\quad = \sum_{j=1}^m \left[\hat{\delta}(\hat{\lambda}_j, \lambda_j^{\nu+1}) - \hat{\delta}(\hat{\lambda}_j, \lambda_j^\nu)\right] \\
&\quad = \sum_{j=1}^m \left[\hat{\lambda}_j \log(\hat{\lambda}_j/\lambda_j^{\nu+1}) - \hat{\lambda}_j + \lambda_j^{\nu+1} - \hat{\lambda}_j \log(\hat{\lambda}_j/\lambda_j^\nu) + \hat{\lambda}_j - \lambda_j^\nu\right] \\
&\quad = \sum_{j=1}^m \left[-\hat{\lambda}_j \log(\lambda_j^{\nu+1}/\lambda_j^\nu) + \left(1 - \tfrac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\lambda_j^{\nu+1}\right] \\
&\quad \overset{(5.23)}{\le} \sum_{j=1}^m \left[-\hat{\lambda}_j \log(\lambda_j^{\nu+1}/\lambda_j^\nu) + \left(1 - \tfrac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\lambda_j^{\nu+1} + \left(1 - \tfrac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\left(\hat{\lambda}_j - \lambda_j^{\nu+1}\right)\right] \\
&\quad = \sum_{j=1}^m \hat{\lambda}_j \left(1 - \tfrac{\lambda_j^\nu}{\lambda_j^{\nu+1}} - \log(\lambda_j^{\nu+1}/\lambda_j^\nu)\right).
\end{aligned}$$

Dieses Ergebnis schreiben wir als

$$\mathcal{D}(\hat{\lambda}, \lambda^{\nu+1}) - \mathcal{D}(\hat{\lambda}, \lambda^\nu) \le \sum_{j=1}^m \hat{\lambda}_j \varphi\left(\frac{\lambda_j^{\nu+1}}{\lambda_j^\nu}\right), \tag{5.24}$$

wobei φ die Funktion einer Variablen

$$\varphi(t) = 1 - \frac{1}{t} - \log t, \quad t > 0, \tag{5.25}$$

ist. Man beachte, daß wegen $\lambda^\nu, \lambda^{\nu+1} \in \mathbb{R}^m_{++}$ die Funktion φ an der Stelle $\lambda_j^{\nu+1}/\lambda_j^\nu$ wohldefiniert und endlich ist. Für ihre Ableitung φ' gilt:

$$\varphi'(t) = \frac{1}{t^2} - \frac{1}{t} = \frac{1-t}{t^2} \quad \begin{cases} > 0 & \text{für} \quad 0 < t < 1 \\ = 0 & \text{für} \quad t = 1 \\ < 0 & \text{für} \quad t > 1 \end{cases} . \tag{5.26}$$

Zusammen mit $\varphi(1) = 0$ folgt damit $\varphi(t) \leq 0$ für alle $t > 0$. Wegen $\lambda^\nu, \lambda^{\nu+1} > 0$ und $\hat{\lambda} \geq 0$ zeigt dies daher, daß die rechte Seite in (5.24) kleiner oder gleich 0 ist. Daraus ergibt sich nun sofort die Behauptung des Lemmas. □

Wir notieren noch das folgende elementare

Lemma 5.4.7 *Sei $(a^\nu)_\nu$ eine monoton fallende Folge nichtnegativer reeller Zahlen. Gilt dann $a^{\nu_\ell} \to 0$ für eine Teilfolge $(a^{\nu_\ell})_\ell$, so folgt $a_\nu \to 0$.*

Beweis: Trivial. □

Nun können wir die Beweistechniken aus [37, Lemma 3.1 (b)] verwenden, um die Konvergenz der dualen Folge zu zeigen.

Lemma 5.4.8 *Die Folge $(\lambda^\nu)_\nu$, die vom Algorithmus 4.1.1 geliefert wird, konvergiert.*

Beweis: Sei $\hat{\lambda} \in M^\infty$ beliebig. Nach Lemma 5.4.6 gilt dann $\mathcal{D}(\hat{\lambda}, \lambda^\nu) \leq \mathcal{D}(\hat{\lambda}, \lambda^0)$ für $\nu = 0, 1, 2, \ldots$. Da nach Lemma 5.4.5 die Funktion $\mathcal{D}(\hat{\lambda}, .)$ beschränkte Niveaumengen besitzt, ergibt sich daraus die Beschränktheit der Folge $(\lambda^\nu)_\nu$. Daher hat die Folge $(\lambda^\nu)_\nu$ eine konvergente Teilfolge $(\lambda^{\nu_\ell})_\ell$, etwa $\lim_{\ell\to\infty} \lambda^{\nu_\ell} = \lambda^\infty$. Selbstverständlich gilt dann $\lambda^\infty \geq 0$. Wegen der Oberhalbstetigkeit von d haben wir

$$d(\lambda^\infty) = \limsup_{\ell\to\infty} d(\lambda^{\nu_\ell}) \geq d^\infty.$$

Dabei haben wir benutzt, daß die Folge $(d(\lambda^\nu))_\nu$ der dualen Funktionswerte nach Lemma 5.4.3 gegen d^∞ konvergiert. Somit ist gezeigt, daß $\lambda^\infty \in M^\infty$, und wir können das Lemma 5.4.6 mit $\hat{\lambda} = \lambda^\infty$ anwenden. Wir erhalten, daß $(\mathcal{D}(\lambda^\infty, \lambda^\nu))_\nu$ monoton fallend ist. Sei $a^\nu = \mathcal{D}(\lambda^\infty, \lambda^\nu)$. Wegen $\lambda^{\nu_\ell} \to \lambda^\infty$ zeigt nun Lemma 5.4.5, daß $\lim_{\ell\to\infty} a^{\nu_\ell} = 0$ gilt. Mit Lemma 5.4.7 folgt, daß auch $(a^\nu)_\nu$ gegen 0 konvergiert und erneute Anwendung von Lemma 5.4.5 ergibt die Konvergenz von $(\lambda^\nu)_\nu$ gegen λ^∞. Damit ist das Lemma bewiesen. □

Korollar 5.4.9 *Für $\nu \to \infty$ konvergiert $\lambda_j^{\nu+1}(a_j^T x^\nu - b_j)$ gegen 0 $(j = 1, \ldots, m)$.*

Beweis: Unmittelbar aus Lemma 5.4.8 und Lemma 4.2.4. □

Wir verwenden weiterhin die Notation von Lemma 5.4.8, indem wir mit λ^∞ den Grenzwert der Folge $(\lambda^\nu)_\nu$ bezeichnen. Während in [37] die Konvergenz der dualen Folge, die das Exponential–Multiplikatorverfahren liefert, gegen einen Optimallösung von (D) über die ergodische Konvergenz der primären Folge bewiesen wird, werden wir hier einen anderen Weg wählen. Dazu benötigen wir zunächst das folgende

Lemma 5.4.10 *Sei λ^* eine beliebige Optimallösung von* (D)*. Gilt $\lambda_j^\infty = 0$, so folgt auch $\lambda_j^* = 0$.*

Beweis: Wir nehmen an, die Behauptung sei falsch, d.h. es gäbe ein $j_0 \in \{1, \ldots, m\}$ mit $\lambda_{j_0}^\infty = 0$ aber $\lambda_{j_0}^* > 0$.

Zunächst gilt $\lambda^* \in M^\infty$, da $d(\lambda^*) \geq d(\lambda^\infty) = d^\infty$ und $\lambda^* \geq 0$. Damit ist Lemma 5.4.6 mit $\hat{\lambda} = \lambda^*$ anwendbar. Es folgt somit, daß die Folge $(\mathcal{D}(\lambda^*, \lambda^\nu))_\nu$ eine monoton fallende Folge nichtnegativer Zahlen ist. Insbesondere ist $\mathcal{D}(\lambda^*, \lambda^\nu) \leq \omega$ für eine Konstante $\omega > 0$. Wir betrachten nun noch einmal genau die Definition von $\mathcal{D}$. Es gilt

$$
\begin{aligned}
\mathcal{D}(\lambda^*, \lambda^\nu) &= \hat{\delta}(\lambda_{j_0}^*, \lambda_{j_0}^\nu) + \sum_{\substack{1 \leq j \leq m \\ j \neq j_0}} \hat{\delta}(\lambda_j^*, \lambda_j^\nu) \\
&\geq \hat{\delta}(\lambda_{j_0}^*, \lambda_{j_0}^\nu) \\
&= \lambda_{j_0}^* \log\left(\frac{\lambda_{j_0}^*}{\lambda_{j_0}^\nu}\right) + \lambda_{j_0}^\nu - \lambda_{j_0}^* \\
&= \lambda_{j_0}^* \log \lambda_{j_0}^* + \lambda_{j_0}^* - \lambda_{j_0}^\nu + (-\lambda_{j_0}^* \log(\lambda_{j_0}^\nu)), \qquad (5.27)
\end{aligned}
$$

wobei sich die Ungleichung aus Lemma 5.4.4 (iii) ergibt. Für $\nu \to \infty$ bleiben die ersten beiden Terme in (5.27) konstant, der dritte wegen der Konvergenz der λ^ν beschränkt, während der letzte Term aufgrund von $\lambda_{j_0}^* > 0$ und $\lambda_{j_0}^\nu \to 0$ gegen $+\infty$ divergiert. Damit folgt aber wegen (5.27), daß auch $\lim\limits_{\nu\to\infty} \mathcal{D}(\lambda^*, \lambda^\nu) = +\infty$ divergiert im Widerspruch zu $\mathcal{D}(\lambda^*, \lambda^\nu) \leq \omega$. $\rightarrow\leftarrow$ □

Damit sind nun alle Voraussetzungen geschaffen, um den folgenden Satz zu zeigen.

Satz 5.4.11 *Die Folge $(\lambda^\nu)_\nu$, die vom Algorithmus 4.1.1 geliefert wird, konvergiert gegen eine Optimallösung von* (D)*.*

Beweis: Sei λ^* eine beliebige Optimallösung von (D). Mit Hilfe von (5.9) und Lemma 5.4.1 (iii) folgt dann

$$\begin{aligned}
d(\lambda^*) - d(\lambda^{\nu+1}) &\overset{(5.9)}{\leq} (b - Ax^\nu)^T(\lambda^* - \lambda^{\nu+1}) \\
&\overset{5.4.1(\mathrm{iii})}{=} \frac{1}{k^\nu}\sum_{j=1}^{m}\left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right)\left(\lambda_j^* - \lambda_j^{\nu+1}\right) \\
&= \frac{1}{k^\nu}\sum_{j=1}^{m}\left[\lambda_j^* \cdot \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right) - \left(\lambda_j^{\nu+1} - \lambda_j^\nu\right)\right]
\end{aligned}$$

$$= \frac{1}{k^\nu}\sum_{j=1}^{m}\left(\lambda_j^\nu - \lambda_j^{\nu+1}\right) + \frac{1}{k^\nu}\sum_{\substack{1\leq j\leq m\\ \lambda_j^\infty = 0}}\lambda_j^* \cdot \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right) + \frac{1}{k^\nu}\sum_{\substack{1\leq j\leq m\\ \lambda_j^\infty > 0}}\lambda_j^* \cdot \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right). \quad (5.28)$$

Wegen Lemma 5.4.10 verschwindet die zweite Summe in (5.28) und wir erhalten die Abschätzung

$$d(\lambda^*) - d(\lambda^{\nu+1}) \leq \frac{1}{k^\nu}\sum_{j=1}^{m}\left(\lambda_j^\nu - \lambda_j^{\nu+1}\right) + \frac{1}{k^\nu}\sum_{\substack{1\leq j\leq m\\ \lambda_j^\infty > 0}}\lambda_j^* \cdot \left(1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right). \quad (5.29)$$

Wir betrachten die Ungleichung (5.29) für $\nu \to \infty$. Da die Folge $(\lambda^\nu)_\nu$ konvergiert und k^ν von Null weg beschränkt bleibt, konvergiert die erste Summe auf der rechten Seite von (5.29) gegen Null. Für $\lambda_j^\infty > 0$ ist auch

$$\lim_{\nu\to\infty}\left[1 - \frac{\lambda_j^\nu}{\lambda_j^{\nu+1}}\right] = 1 - \frac{\lim\limits_{\nu\to\infty}\lambda_j^\nu}{\lim\limits_{\nu\to\infty}\lambda_j^{\nu+1}} = 1 - \frac{\lambda_j^\infty}{\lambda_j^\infty} = 1 - 1 = 0,$$

so daß die gesamte rechte Seite von (5.29) für $\nu \to \infty$ gegen Null konvergiert. Damit ergibt sich durch Grenzübergang $\nu \to \infty$ in (5.29)

$$d(\lambda^*) - d^\infty \leq 0. \quad (5.30)$$

Wegen der Oberhalbstetigkeit von d und der Konvergenz der Folge $(\lambda^\nu)_\nu$ gegen λ^∞ gilt wieder $d^\infty = \lim\limits_{\nu\to\infty} d(\lambda^\nu) \leq d(\lambda^\infty)$, so daß aus (5.30) nun

$$d(\lambda^*) - d(\lambda^\infty) \leq 0$$

folgt. Daher ist λ^∞ ebenfalls optimal für (D). □

5.5 Konvergenzverhalten der primären Folge

In diesem Abschnitt soll nun die primäre Folge $(x^\nu)_\nu$ und ihre Konvergenz untersucht werden. Falls q strikt konvex, d.h. Q strikt positiv definit ist, läßt sich die Konvergenz der Folge $(x^\nu)_\nu$ gegen eine Optimallösung von (QP) sofort aus den bisherigen Ergebnissen ableiten. Nach Bemerkung 4.1.3 gilt dann nämlich wegen der Nichtsingularität von Q

$$x^\nu = Q^{-1}\left(\sum_{j=1}^{m} \lambda_j^{\nu+1} a_j - c\right). \tag{5.31}$$

Da nach Lemma 5.4.8 die Folge $(\lambda^\nu)_\nu$ gegen λ^∞ konvergiert, konvergiert die gesamte rechte Seite in (5.31) für $\nu \to \infty$. Daher ergibt sich auch die Konvergenz der x^ν gegen ein x^∞. Das Korollar 4.2.5 besagt nun, daß x^∞, λ^∞ ein Karush–Kuhn–Tucker–Paar für (QP) sind, also x^∞ optimal für (QP) ist. Man beachte, daß für dieses Resultat das Ergebnis aus Satz 5.4.11 nicht benötigt wird. Somit ist folgender Satz gezeigt:

Satz 5.5.1 *Ist q strikt konvex, d.h. Q strikt positiv definit, so konvergiert die Folge $(x^\nu)_\nu$ gegen eine Optimallösung von* (QP) *und die Folge $(\lambda^\nu)_\nu$ gegen einen zugehörigen Lagrangevektor.*

Für den Fall, daß Q nicht strikt positiv definit ist, muß man etwas mehr Aufwand investieren, um Aussagen über das Verhalten der Folge $(x^\nu)_\nu$ zu erhalten. Zunächst beweisen wir Eigenschaften von x^ν und $\lambda^{\nu+1}$ im Hinblick auf das duale quadratische Programm (QPD).

Lemma 5.5.2 *Für die vom Algorithmus 4.1.1 gelieferten Vektoren x^ν und λ^ν gelten folgende Aussagen:*

(i) $(x^\nu, \lambda^{\nu+1})$ *ist zulässig für* (QPD),

(ii) $\eta(x^\nu, \lambda^{\nu+1}) = d(\lambda^{\nu+1})$,

(iii) $\lim\limits_{\nu\to\infty} \eta(x^\nu, \lambda^{\nu+1}) = q^*$.

(iv) *Ist $\hat{x}$ ein Häufungspunkt der primären Folge $(x^\nu)_\nu$, so ist $(\hat{x}, \lambda^\infty)$ eine Optimallösung von* (QPD).

Beweis: Der Beweis besteht im wesentlichen aus der Verwendung von bereits in den Abschnitten 4.1, 4.2 und 5.4 erarbeiteten Resultaten. Wir wissen nach Satz 3.2.2, daß $\lambda^{\nu+1} > 0$. Ferner liefert Bemerkung 4.1.3, wie schon öfters verwendet,

$$Qx^\nu + c - \sum_{j=1}^{m} \lambda^{\nu+1} a_j = Qx^\nu + c - A^T\lambda^{\nu+1}. \tag{5.32}$$

Damit ist die Zulässigkeit von $(x^\nu, \lambda^{\nu+1})$ für (QPD) und somit die erste Aussage des Lemmas bewiesen. Die zweite Behauptung erfordert etwas elementare Algebra und die Benutzung von (5.32). Sei der Einfachheit halber $x = x^\nu$ und $\lambda = \lambda^{\nu+1}$. Dann gilt

$$\begin{aligned}
\eta(x,\lambda) &= -\tfrac{1}{2}x^TQx + b^T\lambda \\
&\overset{(5.32)}{=} -\tfrac{1}{2}x^T(A^T\lambda - c) + b^T\lambda \\
&= -\tfrac{1}{2}\lambda^TAx + \tfrac{1}{2}c^Tx + \lambda^Tb \\
&= \tfrac{1}{2}x^T(A^T\lambda - c) + c^Tx + \lambda^T(b - Ax) \\
&\overset{(5.32)}{=} \tfrac{1}{2}x^TQx + c^Tx + \lambda^T(b - Ax) = d(\lambda),
\end{aligned}$$

wobei die letzte Gleichheit aus Lemma 5.4.1 (i) folgt. Die Behauptung $\eta(x^\nu, \lambda^{\nu+1}) \to q^*$ folgt sofort aus (ii) und Satz 5.4.11.

Um die letzte Aussage zu zeigen, sei $(x^{\nu_\ell})_\ell$ eine konvergente Teilfolge von $(x^\nu)_\nu$, etwa $\lim\limits_{\ell\to\infty} x^{\nu_\ell} = \hat{x}$. Da die duale Folge $(\lambda^\nu)_\nu$ gegen λ^∞ konvergiert, konvergiert auch $(\lambda^{\nu_\ell+1})_\ell$ gegen λ^∞. Wegen der Stetigkeit von η und Aussage (iii) gilt dann, daß $\eta(\hat{x}, \lambda^\infty) = \lim\limits_{\ell\to\infty} \eta(x^{\nu_\ell}, \lambda^{\nu_\ell+1}) = \lim\limits_{\nu\to\infty} \eta(x^\nu, \lambda^{\nu+1}) = q^*$. Da $(x^{\nu_\ell}, \lambda^{\nu_\ell+1})$ zulässig für (QPD) ist, folgt nun mit den üblichen Schlüssen, daß auch $(\hat{x}, \lambda^\infty)$ zulässig für (QPD) sein muß. Damit ist das Lemma bewiesen. $\square$

Wiederum ist für die Konvergenz von $\eta(x^\nu, \lambda^{\nu+1})$ gegen den Optimalwert q^* die Konvergenz der Folge $(x^\nu)_\nu$ selbst nicht erforderlich. Aus dem letzten Lemma ergibt sich für den linearen Fall unmittelbar das

Korollar 5.5.3 *Sei $Q = 0$, d.h. wir betrachten den linearen Fall, und sei die Menge der Optimallösungen von* (LP) *beschränkt und nichtleer. Dann ist die duale Folge $(\lambda^\nu)_{\nu\geq 1}$ eine Folge von strikt für* (LPD) *zulässigen Vektoren, die gegen eine Optimallösung von* (LPD) *konvergiert.*

Die Modifizierte Barrieremethode kann also in diesem Fall auch als ein Verfahren zur Lösung des dualen linearen Programmes (LPD) angesehen werden. Im Abschnitt 5.6 werden wir für diesen Fall die *Konvergenzrate* der zugehörigen Folge der Funktionswerte untersuchen.

Lemma 5.5.4 *Sei $\bar{x}$ ein beliebiger Häufungspunkt der primären Folge $(x^\nu)_\nu$. Dann gilt $q(\bar{x}) \leq q^*$.*

Beweis: Sei x^* eine beliebige Optimallösung von (QP) und sei $(x^{\nu_\ell})_\ell$ eine Teilfolge von $(x^\nu)_\nu$, die gegen ein $\bar{x}$ konvergiert. Dann gilt

$$\begin{aligned} q(x^{\nu_\ell}) - q(x^*) &\leq \nabla q(x^{\nu_\ell})^T (x^{\nu_\ell} - x^*) \\ &= \sum_{j=1}^{m} \lambda_j^{\nu_\ell+1} (a_j^T x^{\nu_\ell} - a_j^T x^*) \\ &\leq \sum_{j=1}^{m} \lambda_j^{\nu_\ell+1} (a_j^T x^{\nu_\ell} - b_j). \end{aligned} \tag{5.33}$$

Dabei folgt die erste Ungleichung aus der Konvexität von q, die anschließende Gleichheit aus Bemerkung 4.1.3 und die letzte Ungleichung aus der Zulässigkeit von x^* für (QP). Wir betrachten nun die Ungleichung in (5.33) für $\ell \to \infty$. Die rechte Seite in (5.33) strebt wegen Korollar 5.4.9 dann gegen 0, während die linke Seite gegen $q(\bar{x}) - q(x^*)$ konvergiert, so daß man

$$q(\bar{x}) - q(x^*) \leq 0 \tag{5.34}$$

erhält. Dies zeigt die Behauptung des Lemmas. □

Im Falle der Beschränktheit von S ist die Existenz von Häufungspunkten der Folge $(x^\nu)_\nu$ klar, da $x^\nu \in S_{k^\nu}$ und dann S_{k^ν} beschränkt ist (vgl. Lemma 3.2.2). Setzt man hingegen nur voraus, daß die Menge S^* der Optimallösungen von (QP) beschränkt ist, so ist zunächst nicht klar, ob die primäre Folge überhaupt Häufungspunkte besitzt. Die Folge könnte unbeschränkt ohne einen einzigen Häufungspunkt sein. Das dies nicht so ist, zeigt das folgende

Lemma 5.5.5 *Die Folge $(x^\nu)_\nu$ ist beschränkt und besitzt somit Häufungspunkte.*

Beweis: Sei x^* eine beliebige Optimallösung von (QP). Wegen der Konvexität von q und Bemerkung 4.1.3 gilt dann:

$$q(x^*) - q(x^\nu) \geq \nabla q(x^\nu)^T (x^* - x^\nu) = \sum_{j=1}^{m} \lambda_j^{\nu+1} a_j^T (x^* - x^\nu) \geq \sum_{j=1}^{m} \lambda_j^{\nu+1} (b_j - a_j^T x^\nu). \tag{5.35}$$

Die letzte Ungleichung folgt dabei aus der Zulässigkeit von x^*, d.h. $a_j^T x^* \geq b_j$ für $j = 1, \ldots, m$. Wegen Lemma 5.4.9 konvergiert $\lambda_j^{\nu+1}(b_j - a_j^T x^\nu)$ für $\nu \to +\infty$ gegen Null. Insbesondere ist für großes ν

$$\lambda_j^{\nu+1}(b_j - a_j^T x^\nu) \geq -\frac{1}{2}, \quad j = 1, \ldots, m. \tag{5.36}$$

Verwendung von (5.36) in (5.35) ergibt dann:

$$q(x^*) - q(x^\nu) \geq -\frac{m}{2},$$

für alle großen ν. Also ist für alle genügend großen ν

$$x^\nu \in \{x \in S_\kappa : q(x) \le q^* + \frac{m}{2}\}. \tag{5.37}$$

Dabei haben wir benutzt, daß wegen $k^\nu \ge \kappa$ gilt: $S_{k^\nu} \subseteq S_\kappa$. Die Menge auf der rechten Seite in (5.37) ist nach Satz 3.2.3 kompakt. Da für große ν alle Folgenglieder in einer kompakten und damit insbesondere beschränkten Menge enthalten sind, folgt damit die Behauptung des Lemmas. □

Korollar 5.5.6 *Ist S^* beschränkt, so gilt für die Folge (x^ν, λ^ν), die vom Algorithmus 4.1.1 erzeugt wird,*

$$\mathcal{F}(x^{\nu+1}, \lambda^{\nu+1}, k^{\nu+1}) - \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) \ge \frac{k^\nu}{2(c_\kappa k^\nu + 1)} \sum_{j=1}^m \lambda_j^{\nu+1}(a_j^T x^\nu - b_j)^2 \ge 0, \quad \nu = 0, 1, 2, \ldots,$$

wobei

$$\tilde{c}_\kappa = \sup\{\|Ax^\nu - b\|_\infty : \nu = 0, 1, 2, \ldots\} < +\infty.$$

Insbesondere ist die Folge $(\mathcal{F}(x^\nu, \lambda^\nu, k^\nu))_\nu$ monoton steigend.

Beweis: Man dupliziere den Beweis von Lemma 4.2.2 und benutze $\tilde{c}_\kappa$ anstelle von c_κ. □

Korollar 5.5.7 *Ist S^* beschränkt, so existiert $\lim_{\nu \to \infty} \mathcal{F}(x^\nu, \lambda^\nu, k^\nu) = \mathcal{F}^\infty$ und es gilt $\mathcal{F}^\infty \le q^*$.*

Beweis: Man kopiere den Beweis von Korollar 4.2.3 und benutze Korollar 5.5.6 anstatt Lemma 4.2.2. □

Nun beweisen wir eines der Hauptergebnisse dieser Arbeit über das Konvergenzverhalten der primären Folge $(x^\nu)_\nu$:

Satz 5.5.8 *Die Menge S^* der Optimallösungen von* (QP) *sei nichtleer und beschränkt (man beachte, daß dies natürlich sofort dann der Fall ist, wenn S beschränkt ist). Dann ist die Folge $(x^\nu)_\nu$, die der Algorithmus 4.1.1 liefert, beschränkt und besitzt somit Häufungspunkte. Jeder für* (QP) *zulässige Häufungspunkt $\bar{x}$ der Folge $(x^\nu)_\nu$ ist eine Optimallösung von* (QP) *und der Grenzwert λ^∞ der dualen Folge ist ein zu $\bar{x}$ passender optimaler Lagrangevektor.*

Beweis: Die Behauptung, daß die primäre Folge beschränkt ist, wurde bereits bewiesen. Sei daher $\bar{x}$ ein Häufungspunkt der Folge, der zulässig für (QP) ist. Sei x^* eine beliebige Optimallösung von (QP). Aus der Optimalität von x^* und der Zulässigkeit von $\bar{x}$ für (QP) folgt $q^* = q(x^*) \le q(\bar{x})$, also

$$0 \le q(\bar{x}) - q(x^*). \tag{5.38}$$

Aus (5.38) und (5.34) ergibt sich nun $q(\bar{x}) = q(x^*)$ und $\bar{x}$ muß somit Optimallösung von (QP) sein. Indem wir in der Gleichung $Qx^{\nu_\ell} + c - A^T\lambda^{\nu_\ell+1} = 0$, die sich aus Bemerkung 4.1.3 ergibt, ℓ gegen Unendlich streben lassen, sehen wir, daß $\bar{x}$ und λ^∞ die Gleichung (1.2a) erfüllen. Daß auch die Komplementaritätsbedingung (1.2c) für das Paar $(\bar{x}, \lambda^\infty)$ richtig ist, folgt letztendlich sofort aus Korollar 5.4.9. □

Aus dem letzten Satz folgen jetzt noch die nächsten zwei Korollare:

Korollar 5.5.9 *Die Menge S^* der Optimallösungen von* (QP) *sei nichtleer und beschränkt und die Folge $(k^\nu)_\nu$ der Barriereparameter erfülle: $\lim\limits_{\nu\to\infty} k^\nu = +\infty$. Dann ist die Folge $(x^\nu)_\nu$, die der Algorithmus 4.1.1 liefert, beschränkt und besitzt somit Häufungspunkte. Jeder Häufungspunkt $\bar{x}$ der Folge $(x^\nu)_\nu$ ist eine Optimallösung von* (QP) *und λ^∞ ist ein zu $\bar{x}$ passender optimaler Lagrangevektor.*

Beweis: Sei $\bar{x}$ ein Häufungspunkt der primären Folge $(x^\nu)_\nu$, etwa $\bar{x} = \lim\limits_{\ell\to\infty} x^{\nu_\ell}$. Dann gilt

$$a_j^T\bar{x} - b_j = \lim_{\ell\to\infty}\left[a_j^T x^{\nu_\ell} - b_j\right] \ge \lim_{\ell\to\infty}\left[-\frac{1}{k^{\nu_\ell}}\right] = 0 \tag{5.39}$$

für $j = 1, \ldots, m$. Dabei folgt die Ungleichung wegen $x^{\nu_\ell} \in \operatorname{int} S_{k^{\nu_\ell}}$ (vgl. Satz 3.2.5), die letzte Gleichheit aus $\lim\limits_{\nu\to\infty} k^\nu = +\infty$. Nun ist (5.39) aber äquivalent zur Zulässigkeit von $\bar{x}$ für (QP). □

Korollar 5.5.10 *Das quadratische Programm* (QP) *besitze eine eindeutige Optimallösung x^*. Wie im letzten Korollar gelte $\lim\limits_{\nu\to\infty} k^\nu = +\infty$. Dann konvergiert die primäre Folge $(x^\nu)_\nu$ gegen x^*.*

Beweis: Nach Korollar 5.5.9 ist die Folge $(x^\nu)_\nu$ beschränkt und jeder Häufungspunkt der primären Folge Optimallösung von (QP), also gleich x^*. Somit konvergiert insbesondere jede konvergente Teilfolge der primären Folge gegen x^*. Nach Lemma 3.4.3 ist dies nur dann möglich, wenn bereits die Folge $(x^\nu)_\nu$ gegen x^* konvergiert. □

Das letzte Korollar zeigt die Konvergenz der primären Folge $(x^\nu)_\nu$ gegen die Optimallösung von (QP), falls diese eindeutig ist und die Barriereparameter gegen $+\infty$

divergieren. Wir wollen nun weitere hinreichende Kriterien für die Konvergenz der primären Folge beweisen, die auch dann gelten, wenn die Folge der Barriereparameter beschränkt bleibt.

Dazu zitieren wir zunächst folgenden Satz über die Existenz von strikt komplementären Optimallösungen des Paars (LP), (LPD) von linearen Programmen:

Satz 5.5.11 *Besitzt das lineare Programm* (LP) *eine Optimallösung x^*, so existiert eine Optimallösung λ^* von* (LPD)*, so daß für $j = 1, \ldots, m$ genau eine der folgenden Bedingungen gilt:*

(i) $a_j^T x^* > b_j$,

(ii) $\lambda_j^* > 0$.

Beweis: [33, Seite 95–96]. □

Satz 5.5.11 zeigt somit, daß im Falle eines linearen Programms zu einer Optimallösung x^* von (LP) immer ein strikt komplementärer Lagrangemultiplikator λ^* existiert. Mit Hilfe dieses Ergebnisses beweisen wir nun im folgenden Satz weitere Ergebnisse über das Konvergenzverhalten der primären Folge:

Satz 5.5.12 *Das Optimierungsproblem* (QP) *besitze eine eindeutige Lösung x^*. Es gelte eine der folgenden Voraussetzungen:*

(i) $Q = 0$.

(ii) *Es existiert ein zu x^* gehöriger Lagrangevektor λ^* mit $\lambda_j^* > 0$, falls $a_j^T x^* = b_j$.*

Dann konvergiert die primäre Folge $(x^\nu)_\nu$ gegen x^* und der Grenzwert λ^∞ der dualen Folge $(\lambda^\nu)_\nu$ ist ein passender Lagrangevektor zu x^*.

Beweis: Ist $Q = 0$, so besagt Satz 5.5.11, daß zu x^* ein strikt komplementärer Lagrangevektor existiert. Somit impliziert Voraussetzung (i) die Richtigkeit von Voraussetzung (ii) und wir können uns für den Rest des Beweises auf den Fall beschränken, daß die Voraussetzung (ii) erfüllt ist.

Sei x^* die nach Voraussetzung eindeutige Optimallösung von (QP) und sei λ^* ein strikt komplementärer Lagrangevektor zu x^*.

Wir nehmen an, daß die Folge $(x^\nu)_\nu$ nicht gegen x^* konvergiert. Nach Lemma 5.5.5 ist die primäre Folge $(x^\nu)_\nu$ beschränkt, besitzt also einen Häufungspunkt $\bar{x} \neq x^*$. Nach Lemma 5.5.4 gilt:

$$q(\bar{x}) \leq q(x^*).$$

Sei $(x^{\nu_\ell})_\ell$ eine Teilfolge von $(x^\nu)_\nu$, die gegen $\bar{x}$ konvergiert.

Wir zeigen zunächst, daß $a_j^T x^* = b_j$ auch $a_j^T \bar{x} = b_j$ impliziert. Sei dazu $j_0 \in \{1, \ldots, m\}$ fest. Korollar 5.4.9 besagt, daß

$$\lim_{\nu \to \infty} \lambda_{j_0}^{\nu+1}(a_{j_0}^T x^\nu - b_{j_0}) = 0$$

gilt. Somit folgt auch

$$0 = \lim_{\ell \to \infty} \lambda_{j_0}^{\nu_\ell+1}(a_{j_0}^T x^{\nu_\ell} - b_{j_0}) = \lambda_{j_0}^\infty(a_{j_0}^T \bar{x} - b_{j_0}). \tag{5.40}$$

Betrachten wir den zu x^* gehörigen strikt komplementären Lagrangevektor λ^*. Da $a_{j_0}^T x^* = b_j$, gilt nach Voraussetzung: $\lambda_{j_0}^* > 0$. Wir behaupten, daß dann auch $\lambda_{j_0}^\infty > 0$ erfüllt sein muß. In der Tat ist wegen Lemma 5.3.9 der Vektor λ^* eine Optimallösung von (D), und Lemma 5.4.10 besagt daher, daß $\lambda_{j_0}^\infty = 0$ auch $\lambda_{j_0}^* = 0$ impliziert. Folglich muß wegen $\lambda_{j_0}^* > 0$ dann $\lambda_{j_0}^\infty > 0$ gelten. Dies wiederum ergibt zusammen mit (5.40)

$$a_{j_0}^T \bar{x} = b_{j_0}.$$

Wir betrachten für $0 < t < 1$ den Vektor $x(t) = (1-t)x^* + t\bar{x} = x^* + t(\bar{x} - x^*)$ und zeigen, daß $x(t)$ für kleines t zulässig für (QP) ist. Ist $a_j^T x^* = b_j$, so wissen wir, daß auch $a_j^T \bar{x} = b_j$ gilt. Für diese j ist dann

$$a_j^T x(t) = (1-t)a_j^T x^* + t a_j^T \bar{x} = (1-t)b_j + t b_j = b_j.$$

Ansonsten ist $a_j^T x^* > b_j$. Wegen

$$a_j^T x(t) - b_j = \underbrace{a_j^T x^* - b_j}_{>0} + t a_j^T(\bar{x} - x^*)$$

ist dann für kleines $t > 0$ auch $a_j^T x(t) - b_j > 0$. Insgesamt ist daher $x(t)$ für kleines t zulässig für (QP). Nach Annahme gilt $\bar{x} \neq x^*$ und daher folgt auch, daß $x(t) \neq x^*$ gilt. Wegen der Konvexität von q und $q(\bar{x}) \leq q(x^*)$ folgt:

$$q(x(t)) = q((1-t)x^* + t\bar{x}) \leq (1-t)q(x^*) + tq(\bar{x}) \leq (1-t)q(x^*) + tq(x^*) = q(x^*).$$

Folglich ist $x(t) \neq x^*$ ebenfalls eine Optimallösung von (QP) im Widerspruch zur Eindeutigkeit von x^*. Somit kann die Annahme nicht korrekt sein und die Folge $(x^\nu)_\nu$ muß gegen x^* konvergieren. Die Tatsache, daß λ^∞ ein passender Lagrangevektor zu x^* ist, folgt nun sofort aus $x^\nu \to x^*$, $\lambda^\nu \to \lambda^\infty$ und Korollar 4.2.5. $\square$

5.6 Konvergenzrate der dualen Folge für lineare Programme

Die Ergebnisse von Korollar 5.5.3 belegen, daß man die Modifizierte Barrieremethode, wenn sie auf das lineare Programm (LP) angewendet wird, auch als Lösungsverfahren des zu (LP) dualen linearen Programms (LPD) betrachten kann. In diesem Abschnitt sollen für diesen Fall einige Konvergenzraten untersucht werden. Die Analyse verwendet Techniken aus [37, Abschnitt 4], wo das Exponential–Multiplikatorverfahren untersucht wird, und adaptiert die Beweise für die Modifizierte Barrieremethode.

Tseng und Bertsekas zeigen in [37] das folgende

Lemma 5.6.1 *Sei $(z^\nu)_\nu$ eine Folge von dual zulässigen Vektoren (d.h. $d(z^\nu) > -\infty$), die gegen ein z^∞ konvergiert mit $d(z^\nu) \le d(z^\infty)$ für $\nu = 0, 1, 2, \ldots$. Für jedes ν sei $\bar{z}^\nu$ ein Element der Menge*

$$\tilde{M} = \{z \in \mathbb{R}^m : z \ge 0 \text{ und } d(z) = d(z^\infty)\},$$

so daß

$$\bar{z}^\nu \in \arg\ \min\Big\{\|z - z^\nu\|_1 : z \in \tilde{M}, |z_j - z_j^\nu| \le |z_j^\infty - z_j^\nu| \text{ für } j = 1, \ldots, m\Big\}.$$

Dabei sei $\|.\|_1$ die übliche ℓ_1–Norm $\|z\|_1 = \sum_{j=1}^m |z_j|$ (Man beachte, daß stets mindestens ein solcher Vektor $\bar{z}^\nu$ existiert). Dann existiert eine Konstante $c_5 > 0$ und ein $N > 0$, so daß

$$c_5\|z^\nu - \bar{z}^\nu\|_1 \le d(\bar{z}^\nu) - d(z^\nu) \text{ für alle } \nu \ge N$$

gilt.

Beweis: [37, Lemma 4.2] □

Damit können wir nun folgendes Resultat zeigen:

Satz 5.6.2 *Sei die Menge S^* der Optimallösungen von* (LP) *nichtleer und beschränkt. Sei $(\lambda^\nu)_\nu$ die duale Folge, die der Algorithmus 4.1.1 bei Anwendung auf* (LP) *erzeugt. Dann konvergiert $(\lambda^\nu)_\nu$ gegen eine Optimallösung von* (LPD). *Die Folge der dualen Funktionswerte $(d(\lambda^\nu))_\nu = (b^T\lambda^\nu)_\nu$ konvergiert mindestens linear gegen den optimalen Funktionswert q^*. Weiterhin konvergiert der Abstand $\mathrm{dist}_1(\lambda^\nu, D^*)$ von λ^ν zur Menge D^* der Optimallösungen von* (D) *linear gegen 0.*

Gilt $\lim_{\nu\to\infty} k^\nu = +\infty$, so konvergieren die Funktionswerte $(b^T\lambda^\nu)_\nu$ sogar superlinear gegen q^*. In diesem Fall konvergiert auch $\mathrm{dist}_1(\lambda^\nu, D^*)$ superlinear gegen 0.

Beweis: Die Konvergenz der Folge $(\lambda^\nu)_\nu$ gegen eine Optimallösung λ^∞ von (LPD) wurde bereits in Korollar 5.5.3 gezeigt. Wir wenden nun Lemma 5.6.1 auf die Folge $(\lambda^\nu)_\nu$ an. Offenbar sind mit $z^\nu := \lambda^\nu$, $z^\infty := \lambda^\infty$, $\tilde{M} = D^*$ alle Voraussetzungen des Lemmas erfüllt. Sei daher für $\nu = 0, 1, 2, \ldots$

$$\bar{\lambda}^\nu \in \arg\ \min \left\{ \|\lambda - \lambda^\nu\|_1 : \lambda \in D^*, |\lambda_j - \lambda_j^\nu| \leq |\lambda_j^\infty - \lambda_j^\nu| \text{ für } j = 1, \ldots, m \right\}.$$

Nach Lemma 5.6.1 existiert eine Konstante $c_5 > 0$ und ein $N > 0$, so daß

$$c_5 \|\lambda^\nu - \bar{\lambda}^\nu\|_1 \leq d(\bar{\lambda}^\nu) - d(\lambda^\nu) = q^* - d(\lambda^\nu) \tag{5.41}$$

für alle $\nu \geq N$ gilt. Die letzte Gleichheit ist dabei eine Konsequenz aus $\bar{\lambda}^\nu \in D^*$. Wegen $\lim\limits_{\nu \to \infty} \lambda^\nu = \lambda^\infty$ gibt es ein $N' \geq N$, so daß

$$|\lambda_j^\infty - \lambda_j^\nu| \leq \lambda_j^\nu \tag{5.42}$$

für alle j und alle $\nu \geq N'$ gilt. Wegen $\lambda^\nu > 0$ für alle ν (vgl. hierzu Satz 4.1.2) kann man die Ungleichung (5.42) durch λ_j^ν dividieren und man erhält

$$\frac{|\lambda_j^\infty - \lambda_j^\nu|}{\lambda_j^\nu} \leq 1 \quad \text{für } \nu \geq N' \text{ und } j = 1, \ldots, m. \tag{5.43}$$

Nach Definition von $\bar{\lambda}^\nu$ gilt $|\bar{\lambda}_j^\nu - \lambda^\nu| \leq |\lambda_j^\infty - \lambda_j^\nu|$ für alle j, so daß man aus der Ungleichung (5.43) die Abschätzung

$$\frac{|\bar{\lambda}_j^\nu - \lambda_j^\nu|}{\lambda_j^\nu} \leq 1 \quad \text{für } \nu \geq N' \text{ und } j = 1, \ldots, m \tag{5.44}$$

erhält. Sei $0 < \varepsilon < 1$ fest und $0 < t \leq 1 - \varepsilon$. Wegen der Nichtnegativität der Entropie–Funktion $\hat{\delta}(.,.)$ und (5.20) gilt dann

$$\begin{aligned}
d(\lambda^{\nu+1}) &\geq d(\lambda^{\nu+1}) - \frac{1}{k^\nu} \sum_{j=1}^m \hat{\delta}(\lambda_j^\nu, \lambda_j^{\nu+1}) \\
&= \max_{\lambda \geq 0} \left[d(\lambda) - \frac{1}{k^\nu} \sum_{j=1}^m \hat{\delta}(\lambda_j^\nu, \lambda_j) \right] \\
&\geq d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{k^\nu} \sum_{j=1}^m \hat{\delta}\left(\lambda_j^\nu, \lambda_j^\nu + t(\bar{\lambda}^\nu - \lambda_j^\nu)\right) \\
&= d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{k^\nu} \sum_{j=1}^m \lambda_j^\nu \beta \left(\frac{\lambda_j^\nu + t(\bar{\lambda}^\nu - \lambda_j^\nu)}{\lambda_j^\nu} \right) \\
&= d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{k^\nu} \sum_{j=1}^m \lambda_j^\nu \beta \left(1 + t \cdot \frac{\bar{\lambda}_j^\nu - \lambda_j^\nu}{\lambda_j^\nu} \right).
\end{aligned} \tag{5.45}$$

Um die Ungleichung in (5.45) fortführen zu können, schätzen wir die Terme

$$\beta\left(1 + t \cdot \frac{\bar{\lambda}_j^\nu - \lambda_j^\nu}{\lambda_j^\nu}\right) \tag{5.46}$$

nach oben ab. Zur Vereinfachung der Notation sei $\xi_j = \frac{\bar{\lambda}_j^\nu - \lambda_j^\nu}{\lambda_j^\nu}$. Dann besitzt (5.46) die Form $\beta(1 + t\xi_j)$. Wir entwickeln die Funktion β in eine Taylorreihe um 1. Unter Ausnutzung von $\beta(1) = \beta'(1) = 0$ erhält man

$$\beta(1 + t\xi_j) = \beta(1) + \beta'(1)t\xi_j + \beta''(\tau_j)t^2\xi_j^2 = \tau_j^{-2}t^2\xi_j^2 \tag{5.47}$$

wobei $\tau_j \in (\min\{1, 1 + t\xi_j\}, \max\{1, 1 + t\xi_j\})$. Aus Ungleichung (5.44) sehen wir, daß $|\xi_j| \le 1$, also wegen $0 < t \le 1 - \varepsilon$, die Beziehung

$$\varepsilon \le 1 + t\xi_j \le 2 - \varepsilon$$

richtig ist. Daher ist auf jeden Fall $\tau_j \ge \varepsilon$, und wegen (5.47) folgt damit

$$\beta(1 + t\xi_j) \le \varepsilon^{-2}t^2\xi_j^2 \le \varepsilon^{-2}t^2|\xi_j|,$$

wobei die letzte Ungleichung daraus folgt, daß $|\xi_j| \le 1$ und somit $\xi_j^2 \le |\xi_j|$ ist. Damit hat man gezeigt, daß

$$\beta\left(1 + t \cdot \frac{\bar{\lambda}_j^\nu - \lambda_j^\nu}{\lambda_j^\nu}\right) \le \varepsilon^{-2}t^2\frac{|\bar{\lambda}_j^\nu - \lambda_j^\nu|}{\lambda_j^\nu}$$

gilt. Setzt man dieses Ergebnis in (5.45) ein, so ergibt sich für $\nu \ge N'$:

$$\begin{aligned}
d(\lambda^{\nu+1}) \quad &\ge \quad d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{\varepsilon^2 k^\nu}t^2\sum_{j=1}^m \lambda_j^\nu \frac{|\bar{\lambda}_j^\nu - \lambda_j^\nu|}{\lambda_j^\nu} \\
&= \quad d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{\varepsilon^2 k^\nu}t^2\sum_{j=1}^m |\bar{\lambda}_j^\nu - \lambda_j^\nu| \\
&= \quad d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{\varepsilon^2 k^\nu}t^2\|\bar{\lambda}^\nu - \lambda^\nu\|_1 \\
&\overset{(5.41)}{\ge} \quad d\left(\lambda^\nu + t(\bar{\lambda}^\nu - \lambda^\nu)\right) - \frac{1}{c_5\varepsilon^2 k^\nu}t^2\left(q^* - d(\lambda^\nu)\right) \\
&\overset{d \text{ konkav}}{\ge} \quad (1-t)d(\lambda^\nu) + td(\bar{\lambda}^\nu) - \frac{1}{c_5\varepsilon^2 k^\nu}t^2\left(q^* - d(\lambda^\nu)\right) \\
&= \quad (1-t)d(\lambda^\nu) + tq^* - \frac{1}{c_5\varepsilon^2 k^\nu}t^2\left(q^* - d(\lambda^\nu)\right) \\
&= \quad \left(1 - t + \frac{1}{c_5\varepsilon^2 k^\nu}t^2\right)d(\lambda^\nu) - \left(-t + \frac{1}{c_5\varepsilon^2 k^\nu}t^2\right)q^*.
\end{aligned} \tag{5.48}$$

Wegen (5.48) folgt daher

$$d(\lambda^{\nu+1}) - q^* \ge \left(1 - t + \frac{1}{c_5\varepsilon^2 k^\nu}t^2\right)\left(d(\lambda^\nu) - q^*\right) \quad \text{für alle } \nu \ge N'. \tag{5.49}$$

Bisher hatten wir außer $0 < t \le 1-\varepsilon$ keine weiteren Forderungen an t gestellt. Es gilt für diese t auf jeden Fall

$$1 - t + \frac{1}{c_5\varepsilon^2 k^\nu} t^2 \ge 1 - t \ge \varepsilon > 0.$$

Ist darüberhinaus $0 < t \le \frac{c_5\varepsilon^2 k^\nu}{2}$, so gilt

$$1 - t + \frac{1}{c_5\varepsilon^2 k^\nu} t^2 = 1 - t\left(1 - \frac{1}{c_5\varepsilon^2 k^\nu} t\right) \le 1 - \frac{1}{2} t < 1. \tag{5.50}$$

Wegen (5.49) folgt daher für $t = \min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\}$

$$\begin{aligned} q^* - d(\lambda^{\nu+1}) &\le \left(1 - \min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\}\left(1 - \frac{1}{c_5\varepsilon^2 k^\nu}\min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\}\right)\right)(q^* - d(\lambda^\nu)) \\ &= \left(1 - \min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\}\left(1 - \min\left\{\frac{1-\varepsilon}{c_5\varepsilon^2 k^\nu}, \frac{1}{2}\right\}\right)\right)(q^* - d(\lambda^\nu)) \end{aligned}$$

für alle $\nu \ge N'$. Folglich ist für diese ν

$$0 \le \frac{q^* - d(\lambda^{\nu+1})}{q^* - d(\lambda^\nu)} \le 1 - \min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\}\left(1 - \min\left\{\frac{1-\varepsilon}{c_5\varepsilon^2 k^\nu}, \frac{1}{2}\right\}\right). \tag{5.51}$$

Da $k^\nu \ge \kappa > 0$ und $0 < \varepsilon < 1$ fest ist, bleibt $t = \min\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\}$ für *alle* ν von 0 weg beschränkt und wir erhalten aus (5.51)

$$0 \le \frac{q^* - d(\lambda^{\nu+1})}{q^* - d(\lambda^\nu)} \le \gamma < 1$$

für alle $\nu \ge \widetilde{R}$ und eine Konstante $0 < \gamma < 1$. Daher gilt

$$q^* - d(\lambda^\nu) \le \omega\gamma^\nu \text{ für } \nu = 0, 1, 2, \ldots, \tag{5.52}$$

und dies zeigt, daß $(d(\lambda^\nu)_\nu)$ mindestens linear gegen q^* konvergiert. Aus (5.52) folgt dann mit (5.41) auch sofort, daß

$$\mathrm{dist}_1(\lambda^\nu, D^*) \le \|\lambda^\nu - \bar{\lambda}^\nu\|_1 \overset{(5.41)}{\le} \frac{1}{c_5}(q^* - d(\lambda^\nu)) \le \frac{\omega}{c_5}\gamma^\nu.$$

Daher konvergiert $\mathrm{dist}_1(\lambda^\nu, D^*)$ mindestens linear gegen 0.

Betrachten wir nun den Fall, daß die Folge $(k^\nu)_\nu$ gegen $+\infty$ strebt, d.h. $\lim\limits_{\nu\to\infty} k^\nu = +\infty$. Dann gilt

$$\lim_{\nu\to\infty} \min\left\{1-\varepsilon, \frac{c_5\varepsilon^2 k^\nu}{2}\right\} = 1 - \varepsilon$$

und

$$\lim_{\nu\to\infty}\left(1 - \min\left\{\frac{1-\varepsilon}{c_5\varepsilon^2 k^\nu}, \frac{1}{2}\right\}\right) = 1,$$

so daß für $\nu \to \infty$ die rechte Seite in (5.51) insgesamt gegen ε konvergiert. Betrachtet man die Ungleichung (5.51) daher für $\nu \to \infty$, so erhält man:

$$0 \leq \lim_{\nu \to \infty} \frac{q^* - d(\lambda^{\nu+1})}{q^* - d(\lambda^\nu)} \leq \varepsilon.$$

Da $\varepsilon > 0$ beliebig gewählt war, folgt daher

$$\lim_{\nu \to \infty} \frac{q^* - d(\lambda^{\nu+1})}{q^* - d(\lambda^\nu)} = 0,$$

und somit die superlineare Kovergenz von $(d(\lambda^\nu))_\nu$ gegen q^*. Die Tatsache, daß auch $\mathrm{dist}_1(\lambda^\nu, D^*)$ superlinear gegen 0 konvergiert, ergibt sich nun zuletzt wieder aus (5.41).

□

Kapitel 6

Implementierung der Modifizierten Barrieremethode

Um neben der Theorie der Modifizierten Barrieremethode auch die praktische Seite untersuchen zu können, wurde das Verfahren im Rahmen dieser Diplomarbeit implementiert. In diesem Kapitel sollen nun die Implementierung und die gewonnenen numerischen Ergebnisse besprochen werden.

6.1 Vom Modellalgorithmus zum implementierbaren Algorithmus

Dieser Abschnitt beschäftigt sich mit den grundsätzlichen Überlegungen, die nötig sind, um vom Modellalgorithmus 4.1.1 zu einem praktisch implementierbaren Algorithmus zu gelangen. Dabei folgen wir im wesentlichen den Empfehlungen von Polyak aus [19], [27] und [28]. Für eine Implementierung sind vor allem die folgenden Punkte zu klären:

(i) Lösung der Minimierungsprobleme in Schritt (4) des Modellalgorithmus 4.1.1, d.h. numerische Minimierung der Modifizierten Barrierefunktionen $\mathcal{F}(., \lambda, k)$,

(ii) Wahl von geeigneten Startvektoren,

(iii) Abbruchkriterien,

(iv) Wahl der Folge $(k^\nu)_\nu$ der Barriereparameter.

Diese Aspekte sollen in den nächsten Abschnitten zur Sprache kommen.

6.1.1 Numerische Minimierung der Modifizierten Barrierefunktionen

In einem implementierbaren Algorithmus muß der Schritt (4):

(4) *Zu gegebenem* $\lambda^\nu > 0$ *und* $k^\nu > 0$ *bestimme* $x^\nu \in \operatorname{argmin}\{\mathcal{F}(x, \lambda^\nu, k^\nu) : x \in \mathbb{R}^n\}$

des Modellalgorithmus 4.1.1 durch eine endliche Prozedur ersetzt werden. Da die Modifizierte Barrierefunktion $\mathcal{F}(., \lambda^\nu, k^\nu)$ auf dem Inneren int S_{k^ν} von

$$S_{k^\nu} = \left\{ x : a_j^T x - b_j \leq -\frac{1}{k^\nu}, \quad j = 1, \ldots, m \right\}$$

konvex und beliebig oft stetig differenzierbar ist, bietet sich für diesen Zweck das Newton–Verfahren an. Polyak schlägt in [27] bzw. [28] das Newton–Verfahren mit Line–Search nach der Armijo–Goldstein–Regel vor. Als Startpunkt wählt man dabei $x^{\nu,0} := x_{\text{newton-start}} := x^{\nu-1}$ und erzeugt dann rekursiv eine Folge $(x^{\nu,j})_j$. Im folgenden ist der von Polyak empfohlene Algorithmus zur Lösung des Problems

$$x^\nu \in \arg \min_{x \in \operatorname{int} S_k} \mathcal{F}(x, \lambda^\nu, k^\nu) = \arg \min_{x \in \mathbb{R}^n} \mathcal{F}(x, \lambda^\nu, k^\nu).$$

dargestellt:

Algorithmus 6.1.1 (Newton–Verfahren mit Line–Search nach Goldstein)

(1) *Setze* $x^{\nu,0} := x_{\text{newton-start}}$.

(2) *Setze* $j := 0$.

(3) *Setze* $x := x^{\nu,j}$.

(4) *Finde die Newtonrichtung* $\zeta = \zeta(x, \lambda^\nu, k^\nu)$ *als Lösung des linearen Gleichungssystems*

$$\nabla_{xx}\mathcal{F}(x, \lambda^\nu, k^\nu)\zeta = -\nabla_x \mathcal{F}(x, \lambda^\nu, k^\nu). \tag{6.1}$$

(5) *Setze* $t := 1$.

(6) *Teste, ob* $x + t\zeta \in \operatorname{int} S_{k^\nu}$ *und*

$$\mathcal{F}(x + t\zeta, \lambda^\nu, k^\nu) - \mathcal{F}(x, \lambda^\nu, k^\nu) \leq \frac{1}{3} \cdot t \cdot \nabla_x \mathcal{F}(x, \lambda^\nu, k^\nu)^T \zeta. \tag{6.2}$$

(7) *Falls* $x + t\zeta \notin \operatorname{int} S_{k^\nu}$ *oder die Ungleichung (6.2) nicht erfüllt ist, dann setze* $t := \frac{1}{2} \cdot\ t$ *und gehe zu Schritt* (6).

(8) *Setze* $x^{\nu,j+1} := x + t\zeta$ *und* $j := j+1$.

(9) *Falls* $t < 1$, *dann gehe zu Schritt* (3).

(10) *Falls* $t = 1$ *und* $\|\zeta\|_2 \le \varepsilon_{\text{newton}}$, *dann setze* $x^\nu := x^{\nu,j}$ *und* STOP.

(11) *Ansonsten gehe zu Schritt* (3).

Durch Anwendung des Algorithmus 6.1.1 erhalten wir eine Approximation für das Minimum der Modifizierten Barrierefunktion $\mathcal{F}(., \lambda^\nu, k^\nu)$. Danach wird die Lagrangemultiplikatorschätzung λ^ν mit Hilfe dieser Approximation zu $\lambda^{\nu+1}$ aktualisiert (vgl. hierzu Schritt (5) des Algorithmus 4.1.1).

Um die Newton–Richtung $\zeta = \zeta(x, \lambda^\nu, k^\nu)$ in einem Punkt $x \in \text{int } S_{k^\nu}$ zur Multiplikator–Approximation $\lambda := \lambda^\nu \in \mathbb{R}^m_{++}$ und zum Barriereparameter $k := k^\nu > 0$ zu finden, müssen wir das lineare Gleichungssystem (6.1) lösen. Setzen wir explizit die Formeln für den Gradienten und die Hessematrix von $\mathcal{F}$ ein, so schreibt sich dieses lineare System als

$$\left(Q + kA^T\Delta(x,k)^{-2}\Lambda A\right)\zeta = -\left(Qx + c - A^T\Delta(x,k)^{-1}\lambda\right). \tag{6.3}$$

Wir erinnern daran, daß die Diagonalmatrix $\Delta(x,k)$ in (3.16) als

$$\Delta(x,k) = \text{diag}(k(a_1^Tx - b_1) + 1, \ldots, k(a_m^Tx - b_m) + 1)$$

definiert wurde. Für $x \in \text{int } S_k$ besitzt $\Delta(x,k)$ nur strikt positive Einträge. Wegen $\lambda \in \mathbb{R}^m_{++}$ folgt daher, daß die Matrix $Q + kA^T\Delta(x,k)^{-2}\Lambda A$ positiv definit ist, sofern A vollen Spaltenrang besitzt oder Q strikt positiv definit auf dem Nullraum von A ist. Aufgrund dessen kann man dann zur Lösung des linearen Gleichungssystems (6.3) entsprechende Verfahren einsetzen, die diese Eigenschaft ausnutzen. In der Implementierung der Modifizierten Barrieremethode haben wir das *Cholesky–Verfahren* (siehe z.B. [35, Abschnitt 4.3]) verwendet.

6.1.2 Abbruchkriterien

Seien $x \in \mathbb{R}^n$ und $\lambda \in \mathbb{R}^m_+$ beliebige Vektoren. Dann definieren wir die primäre Unzulässigkeit von x durch

$$\mathcal{U}_{\text{primär}}(x) := \max_{1 \le j \le m} \max\left\{0, b_j - a_j^Tx\right\}$$

und die duale Unzulässigkeit von (x, λ) durch

$$\begin{aligned}\mathcal{U}_{\text{dual}}(x,\lambda,k) &:= \|Qx + c - A^T\Delta(x,k)^{-1}\lambda\|_\infty \\ &= \|\nabla_x\mathcal{F}(x,\lambda,k)\|_\infty.\end{aligned}$$

Wir definieren die Komplementarität der beiden Vektoren x und λ durch

$$\mathcal{C}(x,\lambda) := \sum_{j=1}^{m} \lambda_j \left| b_j - a_j^T x \right| .$$

Letztendlich sei

$$\mathcal{T}(x,\lambda,k) := \max\left\{ \mathcal{U}_{\text{primär}}(x), \mathcal{U}_{\text{dual}}(x,\lambda,k), \mathcal{C}(x,\lambda) \right\} .$$

Seien $x \in \mathbb{R}^n$ und $\lambda \in \mathbb{R}_+^m$. Dann ist x genau dann eine Optimallösung des konvexen quadratischen Programms (QP) und (x,λ) eine Optimallösung des dualen quadratischen Programms (QPD), wenn $\mathcal{T}(x,\lambda) = 0$ gilt.

Um dies zu sehen, sei zunächst $\mathcal{T}(x,\lambda,k) = 0$. Dann folgt aus $\mathcal{U}_{\text{primär}}(x) = 0$ die Zulässigkeit von x für (QP). Ferner ist

$$0 = \mathcal{C}(x,\lambda) = \sum_{j=1}^{m} \lambda_j \left| b_j - a_j^T x \right| = \sum_{j=1}^{m} \lambda_j (a_j^T x - b_j).$$

Wegen der Zulässigkeit von x für (QP) und $\lambda \geq 0$ folgt dann aus der letzten Gleichung:

$$\lambda_j (a_j^T x - b_j) = 0 \quad \text{für } j = 1,\ldots,m. \tag{6.4}$$

Dies wiederum impliziert: $\Delta(x,k)^{-1}\lambda = \lambda$. Daher folgt dann aus

$$0 = \mathcal{U}_{\text{dual}}(x,\lambda,k) = \|Qx + c - A^T \Delta(x,k)^{-1}\lambda\|_\infty = \|Qx + c - A^T\lambda\|_\infty$$

die Zulässigkeit von (x,λ) für (QPD). Zusammen mit (6.4) können wir nun schließen, daß

$$\lambda^T b = \lambda^T A x = x^T Q x + c^T x \tag{6.5}$$

gilt. Nun folgt die Optimalität von x für (QP) und die Optimalität von (x,λ) für (QPD) sofort aus aus dem Satz von Dorn (Satz 1.2.1).

Sei umgekehrt x optimal für (QP) und (x,λ) optimal für (QPD). Dann gilt: $\mathcal{U}_{\text{primär}}(x) = 0$. Ferner gilt wegen des Satzes von Dorn 1.2.1, daß (6.5) erfüllt ist. Daraus folgt dann insbesondere die Gültigkeit von Gleichung (6.4). Daher gilt wiederum $\Delta(x,k)^{-1}\lambda = \lambda$ und es folgt $\mathcal{U}_{\text{dual}}(x,\lambda,k) = 0$. Ebenso impliziert (6.4) auch $\mathcal{C}(x,\lambda) = 0$. Insgesamt folgt also $\mathcal{T}(x,\lambda,k) = 0$.

Vergleichen wir die Ausdrücke für die primäre und duale Unzulässigkeit sowie für die Komplementarität mit den Karush–Kuhn–Tucker Bedingungen (1.2a) bis (1.2c), so sehen wir, daß $\mathcal{T}(x,\lambda) = 0$ ebenfalls genau dann erfüllt ist, wenn die Vektoren x und λ ein Karush–Kuhn–Tucker Paar für (QP) sind.

Wir verwenden daher in unserer Implementierung die primäre Unzulässigkeit $\mathcal{U}_{\text{primär}}$, die duale Unzulässigkeit $\mathcal{U}_{\text{dual}}$ und die Komplementarität $\mathcal{C}$, um Abbruchkriterien zu erhalten. Seien x und λ die aktuellen Iterationsvektoren. Dann brechen wir die Iteration ab, falls folgende drei Bedingungen erfüllt sind:

$$\begin{aligned} \mathcal{U}_{\text{primär}}(x) &\leq \delta_{\text{primär}}, \\ \mathcal{U}_{\text{dual}}(x, \lambda, k) &\leq \delta_{\text{dual}}, \\ \mathcal{C}(x, \lambda) &\leq \delta_{\text{kompl}}. \end{aligned} \tag{6.6}$$

Dabei sind $\delta_{\text{primär}}$, δ_{dual} und δ_{kompl} Konstanten. Diese sind vom Programm mit Voreinstellungen belegt. Der Benutzer kann die Voreinstellungen jedoch in einer Konfigurationsdatei ändern, ohne das Programm neu zu übersetzen (siehe dazu Abschnitt A.2, Seite 141).

6.1.3 Wahl der Startvektoren

Wir benötigen für die Initialisierung der Modifizierten Barrieremethode zwei Startvektoren: $\bar{x} \in \mathbb{R}^n$ und $\lambda^0 \in \mathbb{R}^m_{++}$. Der Vektor λ^0 wird als erste Schätzung für den optimalen Lagrangemultiplikator gebraucht. Wie Polyaks Ergebnisse in Abschnitt 5.1 über die Konvergenz des Verfahrens nahelegen, wählen wir $\lambda^0 := e$.

Der Vektor $\bar{x}$ dient als Startpunkt der ersten Newton–Iteration, d.h. der Newton–Iteration zur (approximativen) Lösung von

$$\min_{x \in S_{k^0}} \mathcal{F}(x, \lambda^0, k^0).$$

Da die logarithmischen Strafterme der Modifizierten Barrierefunktion

$$\mathcal{F}(x, \lambda^0, k^0) = \frac{1}{2} x^T Q x + c^T x - \frac{1}{k^0} \sum_{j=1}^{m} \lambda_j \log(k^0 \cdot (a_j^T x - b_j) + 1)$$

nur für

$$x \in \text{int } S_{k^0} = \left\{ x : a_j^T x - b_j > -\frac{1}{k^0}, \quad j = 1, \ldots, m \right\}$$

definiert sind, muß $\bar{x} \in \text{int } S_{k^0}$ gelten. Probleme dieser Art löst man üblicherweise durch Einführen einer künstlichen Variablen x_{n+1} und einer zusätzlichen Nebenbedingung (vgl. hierzu etwa [24, Abschnitt 11.4.4], [21, Abschnitt 4]). Man transformiert das Ausgangsproblem

$$\begin{array}{lll} \text{(QP)} & \min & \frac{1}{2} x^T Q x + c^T x \\ & x \in \mathbb{R}^n : & Ax \geq b \end{array}$$

damit in das Problem

$$\begin{array}{lll} \text{(QP)}^{(\text{M})} & \min & \frac{1}{2} x^T Q x + c^T x + M x_{n+1} \\ & & Ax + e x_{n+1} \geq b, \\ & & x_{n+1} \geq 0. \end{array}$$

Dabei ist $M > 0$ eine (große) Konstante. Setzt man voraus, daß $S = \{x : Ax \geq b\}$ nichtleer ist, so kann man zeigen, daß für genügend großes M ein Vektor x^* genau dann Optimallösung von (QP) ist, wenn $\binom{x^*}{0}$ Optimallösung von $(\mathrm{QP})^{(\mathrm{M})}$ ist und daß jede Optimallösung $\binom{x}{x_{n+1}}$ von $(\mathrm{QP})^{(\mathrm{M})}$ die Bedingung $x_{n+1} = 0$ erfüllt.

Für das Problem $(\mathrm{QP})^{(\mathrm{M})}$ kann man leicht einen zulässigen Vektor angeben. Sei dazu $\hat{x} \in \mathbb{R}^n$ beliebig. Wählt man

$$\hat{x}_{n+1} > \max\left\{\min_{1\leq j\leq m} \left|\left\{0, a_j^T\hat{x} - b_j\right\}\right|\right\},$$

so ist $\binom{\hat{x}}{\hat{x}_{n+1}}$ strikt zulässig für $(\mathrm{QP})^{(\mathrm{M})}$. Man benutzt nun die Modifizierte Barrieremethode, um das Problem $(\mathrm{QP})^{(\mathrm{M})}$ zu lösen.

Die soeben vorgestellte Transformation nutzt nicht aus, daß die Modifizierte Barrieremethode zum Start der ersten Newton–Iteration eigentlich nur einen Vektor $x \in \operatorname{int} S_{k^0}$ benötigt. Dieser Vektor muß nicht im Inneren $\operatorname{int} S$ der zulässigen Punktmenge S liegen. Dies ist aber zumindest eine theoretische Stärke der Modifizierten Barrieremethode. Besonders interessant ist diese Eigenschaft in folgendem Zusammenhang:

Es sei eine Optimallösung x^* des quadratischen Programms (QP) bekannt. Nun nehmen wir an, daß die Nebenbedingungen unwesentlich verändert werden. Dann ist x^* im allgemeinen nicht mehr zulässig für das neue quadratische Programm, liegt aber möglicherweise sehr dicht an der neuen Optimallösung. Wählt man $k^0 > 0$ ausreichend *klein*, so liegt x^* im Inneren $\operatorname{int} S_{k^0}$ von S_{k^0} und kann als Startvektor für die erste Newton–Iteration verwendet werden. Das folgende Beispiel verdeutlicht diesen Sachverhalt:

Beispiel 6.1 Wir betrachten das quadratische Programm

$$\begin{aligned} &\min x^2 \\ &x \geq 0. \end{aligned}$$

Die Optimallösung des quadratischen Programms ist $x^* = 0$. Ersetzen wir die Nebenbedingung $x \geq 0$ nun durch $x \geq \delta$ für ein $\delta > 0$, so ist $x^* = 0$ nicht mehr zulässig für das veränderte Optimierungsproblem. Die Optimallösung des neuen Problems ist $x^*_{\mathrm{neu}} = \delta$ und liegt (für kleines δ) sehr dicht an $x^* = 0$. Für $k^0 < \frac{1}{\delta}$ können wir x^* als Startpunkt für die erste Newton–Iteration verwenden. $\diamond$

Wir haben eben gesehen, daß durch *genügend kleine* Wahl des ersten Barriereparameters auch unzulässige Startpunkte verwendet werden können, um die Iteration der Modifizierten Barrieremethode zu beginnen. Andererseits suggerieren die Ergebnisse von Polyak aus Abschnitt 5.1, daß man die Barriereparameter *ausreichend groß* wählen muß, um eine zufriedenstellende Konvergenzgeschwindigkeit zu erhalten. Daher liegt die Idee nahe, bei einem unzulässigen Startvektor zunächst einen kleinen Barriereparameter zu wählen und diesen dann im Laufe der Iteration nach Möglichkeit wachsen zu lassen. An dieser Stelle möchten wir darauf hinweisen, daß Polyak die eben beschriebene Variante weder in [19], [27] noch in [28] erwähnt.

In der Implementierung `modbfm` wurden beide oben erwähnten Konzepte — die Transformation auf $(QP)^{(M)}$ sowie die Möglichkeit, mit einem unzulässigen Punkt zu starten und den Barriereparameter dann zu vergrößern — realisiert. Der Benutzer hat die Wahl, welche Variante er verwenden möchte, um ein bestimmtes Problem zu lösen. Eine genauere Beschreibung des Programms `modbfm` und seiner Bedienung befindet sich in Anhang A.

6.1.4 Wahl der Barriereparameter

Für die Implementierung der Modifizierten Barriermethode ist weiterhin zu klären, wie man die Folge $(k^\nu)_\nu$ der Barriereparameter in der Praxis wählen soll. Wir wollen hier drei Möglichkeiten vorstellen, die zu drei verschiedenen Varianten der Modifizierten Barrieremethode führen.

Zum einen schlägt Polyak in [27] für allgemeine nichtlineare Programme eine Variante vor, in der versucht wird, den Barriereparameter während der Iteration anzupassen. Dazu betrachtet er zum Barriereparameter $k > 0$ außer der Menge S_k auch die Menge

$$S_k^+ = \left\{ x : a_j^T x - b_j > \frac{1}{k}, \quad j = 1, \dots, m \right\}.$$

Im folgenden ist die Modifizierte Barrieremethode mit Parameteranpassung dargestellt. Wir formulieren den Algorithmus ohne die Abbruchkriterien, die in Abschnitt 6.1.2 eingeführt wurden.

Algorithmus AP–MBFM (Modifizierte Barrieremethode mit Parameter–anpassung)

(1) *Wähle eine Folge* $(\hat{k}^s)_s$ *mit* $0 < \hat{k}^0 < \hat{k}^1 < \cdots$ *und* $\lim_{s \to \infty} \hat{k}^s = +\infty$.

(2) *Wähle* $\bar{x} \in S_{\hat{k}^0}^+$ *und setze* $\lambda^0 := e$.

(3) *Wähle* $0 < \gamma \le \frac{1}{2}$.

(4) *Setze* $\nu := 0$, $s := 0$ *und* $\ell := 1$.

(5) *Setze* $x_{\text{newton-start}} := \bar{x}$.

(6) *Setze* $k^\nu := \hat{k}^s$.

(7) *Verwende den Algorithmus 6.1.1, um, ausgehend vom Startpunkt* $x_{\text{newton-start}}$, *einen Punkt* $\hat{x}$ *mit* $\|\zeta\|_2 = \|\zeta(\hat{x}, \lambda^\nu, k^\nu)\|_2 \leq \varepsilon_{\text{newton}}$ *zu finden.*

(8) *Setze* $\hat{\lambda} = \Delta(\hat{x}, k^\nu)\lambda^\nu$.

(9) *Wenn* $\mathcal{T}(\hat{x}, \hat{\lambda}, k^\nu) \leq \gamma^{\ell+1}$, *dann setze* $x^\nu := \hat{x}$,
$\lambda^{\nu+1} := \hat{\lambda}$,
$\ell := \ell + 1$,
$\varepsilon_{\text{newton}} := \varepsilon_{\text{newton}} \cdot \gamma$,
$x_{\text{newton-start}} := x^\nu$,
$\nu := \nu + 1$ *und gehe zu Schritt* (6).

(10) *Setze* $x' := \arg\ \min\{q(x) : x \in \{x^1, \ldots, x^\nu\} \cup \{\hat{x}\}\}$.

(11) *Bestimme* $t_{\max} = \max\{t : 0 \leq t \leq 1, \bar{x} + t(x' - \bar{x}) \in S^+_{k^\nu}\}$.

(12) *Setze* $x^\nu := \bar{x} + t_{\max}(x' - \bar{x})$.

(13) *Setze* $\lambda^{\nu+1} := e$.

(14) *Setze* $s := s + 1$.

(15) *Setze* $\ell := 1$.

(16) *Setze* $\varepsilon_{\text{newton}} := \varepsilon_{\text{newton}}/k^\nu$.

(17) *Setze* $x_{\text{newton-start}} := x^\nu$.

(18) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (6).

Die Folge $(\hat{k}^s)_s$ aus dem Algorithmus AP–MBFM ist in der Implementierung `modbfm` rekursiv wie folgt definiert:

$$\begin{aligned} \hat{k}^0 &:= k_{\text{start}}, \\ \hat{k}^{s+1} &:= \varrho_{\text{faktor}} \cdot \hat{k}^s + \varrho_{\text{summand}}. \end{aligned} \tag{6.7}$$

Dabei sind $k_{\text{start}} > 0$, ϱ_{faktor} und ϱ_{summand} Konstanten, die vom Benutzer in einer Konfigurationsdatei bei Bedarf geändert werden können. Dies ist möglich, ohne das Programm neu zu übersetzen (siehe dazu Anhang A.1, Seite 140).

Im Gegensatz zum Algorithmus AP–MBFM steht die Modifizierte Barrieremethode mit konstantem Barriereparameter. Diese Version wird von Polyak vor allem in [19] für lineare Programme empfohlen. Bei der Darstellung verzichten wir wiederum auf die Aufführung der Abbruchbedingungen.

Algorithmus KP–MBFM (Modifizierte Barrieremethode mit konstantem Barriereparameter)

(1) *Wähle* $k > 0$.

(2) *Wähle* $\bar{x} \in \text{int } S_k$ *und setze* $\lambda^0 := e$.

(3) *Wähle* $0 < \gamma \leq 1$.

(4) *Setze* $\nu := 0$.

(5) *Setze* $x_{\text{newton-start}} := \bar{x}$.

(6) *Verwende den Algorithmus 6.1.1, um, ausgehend vom Startpunkt* $x_{\text{newton-start}}$, *einen Punkt* $\hat{x}$ *mit* $\|\zeta\|_2 = \|\zeta(\hat{x}, \lambda^\nu, k)\|_2 \leq \varepsilon_{\text{newton}}$ *zu finden.*

(7) *Setze* $x^\nu := \hat{x}$.

(8) *Setze* $\lambda^{\nu+1} := \Delta(x^\nu, k)^{-1}\lambda^\nu$.

(9) *Setze* $\varepsilon_{\text{newton}} := \varepsilon_{\text{newton}} \cdot \gamma$.

(10) *Setze* $x_{\text{newton-start}} := x^\nu$.

(11) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (6).

Im folgenden wollen wir noch eine dritte Variante der Modifizierten Barrieremethode vorschlagen, die bereits auf Seite 101 erwähnt wurde. Dabei startet man mit einem unzulässigen Vektor $\bar{x}$ und wählt den ersten Barriereparameter k^0 so klein, daß $\bar{x} \in \text{int } S_{k^0}$ gilt. Ausgehend von $\bar{x}$ und $\lambda^0 := e$ findet man durch das Newton–Verfahren ein approximatives Minimum x^0 von $\mathcal{F}(., \lambda^0, k^0) = \mathcal{M}(., k^0)$.

Wie bereits weiter oben erwähnt wurde, legen die Ergebnisse von Polyak aus Abschnitt 5.1 nahe, daß man für eine schnelle Konvergenz die Barriereparameter genügend groß wählen muß. Daher werden wir bei der im folgenden beschriebenen Variante bei jedem Multiplikator–Update den Barriereparameter (eventuell bis zu einer bestimmen oberen Schranke) erhöhen. Damit gelangt man dann vom kleinen Startbarriereparameter k^0 zu Barrieren, die eine rasche Konvergenz sichern.

Will man den Vektor x^ν, der das Endergebnis der letzten Newton–Iteration ist, als Startvektor für das neue Minimierungsproblem

$$x^{\nu+1} \in \arg \min_{x \in S_{k^{\nu+1}}} \mathcal{F}(x, \lambda^{\nu+1}, k^{\nu+1})$$

benutzen, so kann man $k^{\nu+1}$ nicht beliebig groß wählen. Der Vektor x^ν liegt nämlich in $\text{int } S_{k^\nu}$, aber im allgemeinen nicht in S. Dies bedeutet, daß in der Regel x^ν kein

Element von int $S_{k^{\nu+1}}$ sein wird, wenn $k^{\nu+1}$ zu groß gewählt ist. Ist $x^\nu \in S$, so bestehen für $k^{\nu+1}$ keine Einschränkungen. Sei im folgenden daher $x^\nu \notin S$. Damit x^ν noch in int $S_{k^{\nu+1}}$ liegt, muß dann gelten:

$$a_j^T x^\nu - b_j > -\frac{1}{k^{\nu+1}}, \quad j = 1, \ldots, m,$$

also

$$k^{\nu+1} < \min\left\{\frac{1}{b_j - a_j^T x^\nu} : a_j^T x^\nu < b_j\right\}.$$

Die dritte Variante der Modifizierten Barrieremethode läßt sich dann wie folgt formulieren:

Algorithmus US–MBFM (Modifizierte Barrieremethode mit unzulässigem Startpunkt)

(1) *Zu vorgegebenem Startpunkt* $\bar{x}$ *wähle* $k^0 > 0$ *so klein, daß* $\bar{x} \in \operatorname{int} S_{k^0}$.

(2) *Setze* $\lambda^0 := e$.

(3) *Wähle* $0 < \gamma \leq 1$.

(4) *Setze* $\nu := 0$.

(5) *Setze* $x_{\text{newton-start}} := \bar{x}$.

(6) *Verwende den Algorithmus 6.1.1, um, ausgehend vom Startpunkt* $x_{\text{newton-start}}$, *einen Punkt* $\hat{x}$ *mit* $\|\zeta\|_2 = \|\zeta(\hat{x}, \lambda^\nu, k^\nu)\|_2 \leq \varepsilon_{\text{newton}}$ *zu finden.*

(7) *Setze* $x^\nu := \hat{x}$.

(8) *Setze* $\lambda^{\nu+1} := \Delta(x^\nu, k^\nu)^{-1}\lambda^\nu$.

(9) *Setze* $\varepsilon_{\text{newton}} := \varepsilon_{\text{newton}} \cdot \gamma$.

(10) *Setze* $x_{\text{newton-start}} := x^\nu$.

(11) *Wenn* $x^\nu \in S$, *dann wähle* $k^{\nu+1} \geq k^\nu$,
setze $\nu := \nu + 1$ *und gehe zu Schritt* (6).

(12) *Wähle* $k^{\nu+1}$, *so daß* $k^\nu \leq k^{\nu+1} < \min\left\{\frac{1}{b_j - a_j^T x^\nu} : a_j^T x^\nu < b_j\right\}$.

(13) *Setze* $\nu := \nu + 1$ *und gehe zu Schritt* (6).

Die zuletzt beschriebene Variante der Modifizierten Barrieremethode US–MBFM besitzt den Vorteil, daß sie mit einem unzulässigen Vektor gestartet werden kann. Sie benötigt weiterhin an keiner Stelle die Existenz von Punkten im Inneren int S der zulässigen Menge und ist daher auch besonders dann interessant, wenn das Innere von S leer ist. Wie oben bereits erwähnt, wird diese Variante von Polyak nicht in Betracht gezogen. Die numerischen Ergebnisse aus Abschnitt 6.2 werden jedoch zeigen, daß diese Version der Modifizierten Barrieremethode ein effizientes Lösungsverfahren sein kann.

In der Implementierung `modbfm` wurden die Barriere–Parameter bei der Variante US–MBFM des Verfahrens wie folgt gewählt: Ausgehend vom Startvektor $\bar{x}$ wählen wir

$$k^0 := \begin{cases} k_{\text{Start}}, & \text{falls } \bar{x} \text{ zulässig für (QP) ist} \\ \min\left\{\frac{1}{2\cdot(b_j - a_j^T\bar{x}) + \frac{1}{2}} : a_j^T\bar{x} < b_j\right\}, & \text{sonst.} \end{cases}$$

Die Barriereparameter–Erhöhungen in Schritt (11) bzw. (12) des Algorithmus US–MBFM folgendermaßen realisiert: Bei einer Erhöhung des Barriereparameters ohne Einschränkungen (Schritt (11)) setzen wir

$$k^{\nu+1} := \varrho_{\text{faktor}} \cdot k^\nu + \varrho_{\text{summand}}.$$

Dabei sind k_{start}, ϱ_{faktor} und ϱ_{summand} vom Benutzer festlegbare Konstanten (vgl. auch Seite 103). Betrachten wir nun eine Parametererhöhung mit Einschränkung (Schritt (12)). Sei

$$\omega = \frac{1}{2}\left(k^\nu + \min\left\{\frac{1}{b_j - a_j^T x^\nu} : a_j^T x^\nu < b_j\right\}\right).$$

Dann wählen wir in Schritt (12):

$$k^{\nu+1} := \min\left(\varrho_{\text{faktor}} \cdot k^\nu + \varrho_{\text{summand}}, \omega\right)$$

Die Minimumsbildung aus $k \cdot \varrho_{\text{faktor}} + \varrho_{\text{summand}}$ und ω verhindert dabei, daß die Barriereparameter zu schnell wachsen. Die Motivation zu dieser Beschränkung des Wachstums lieferten einige praktische Versuche, bei denen sich zeigte, daß bei einem zu großen Quotienten zwischen zwei aufeinanderfolgenden Barriereparametern der zuletzt erhaltene Iterationspunkt in der Regel ein sehr schlechter Startpunkt für das neue (unrestringierte) Minimierungsproblem ist.

Bei der Implementierung wurde ferner vorgesehen, die Folge der Barriereparameter bei Bedarf von oben durch eine Konstante $k_{\max}$ zu beschränken. Ist diese Option gewählt, so wird nach jeder Barriereparameter–Erhöhung der Parameter $k^{\nu+1}$ durch $k^{\nu+1} := \min(k^{\nu+1}, k_{\max})$ festgelegt. Bei den Statistiken in den folgenden Abschnitten werden als Barriereerhöhungen nur echte Erhöhungen des Barriereparameters gezählt.

6.2 Numerische Ergebnisse

Die Implementierung der Modifizierten Barrieremethode in Form des Programmes `modbfm` wurde benutzt, um einige kleinere Probleme (mit bis zu 128 Unbekannten) zu lösen. In diesem Abschnitt sollen die verwendeten Testprobleme und die numerischen Ergebnisse vorgestellt werden.

Die Ergebnisse für die einzelnen Testproblemklassen sind in den Abschnitten 6.2.1 bis 6.2.5 dargestellt. Um eine Übersicht zu ermöglichen, fassen wir im Abschnitt 6.2.6 die wesentlichen Erkenntnisse aus den einzelnen Abschnitten noch einmal kurz zusammen.

Es wurde das Verhalten der Modifizierten Barrieremethode für drei Klassen von linearen Programmen und zwei Typen von quadratischen Programmen untersucht. Die Testprobleme wurden durch das Programm `gen`[1] erzeugt.

Das Programm `modbfm` wurde auf einer Sun SparcStation ELC [2] des Lehrstuhls für Informatik I an der Universität Würzburg eingesetzt. Die Quelltexte für `modbfm` wurden mit Hilfe des GNU–C++–Compilers[3] übersetzt. Zur Kontrolle wurde ein Teil der Ergebnisse anschließend auf einem Rechner mit Intel 80486 CPU[4] überprüft. Wiederum wurde der GNU–C++–Compiler benutzt, um aus den Quelltexten ein ausführbares Programm zu erzeugen. Bis auf gelegentliche Unterschiede in den hinteren Nachkommastellen (im Bereich 10^{-10} bis 10^{-7}) waren die Ergebnisse auf beiden Computern identisch. Die Differenzen sind auf die unterschiedlichen Implementierungen der Fließkommaarithmetik der beiden Rechnersysteme zurückzuführen.

Bereits zu Beginn der Testphase zeigte sich, daß der von Polyak in [27] vorgeschlagene Algorithmus AP–MBFM zumindest in dieser Form für praktische Zwecke ungeeignet zu sein scheint. Bei den numerischen Versuchen strebte das Maß $\mathcal{T}(\hat{x}, \hat{\lambda}, k)$ aus Schritt (9) des Algorithmus AP–MBFM nicht schnell genug gegen Null, so daß in der Iteration der Barriereparameter erhöht und die laufende Lagrangemultiplikatorschätzung λ^ν wieder mit dem Startvektor $\lambda^0 = e$ initialisiert wurde. Die Vergrößerung des Barriereparameters und die gleichzeitige Erhöhung der geforderten Newton–Genauigkeit $\varepsilon_{\mathrm{newton}}$ um den Faktor γ bewirkten letztendlich bei fast allen Testproblemen den Zusammenbruch der Line–Search: Der Line–Search Parameter t schrumpfte auf die Maschinengenauigkeit und dann auf Null. Dies führte dazu, daß das Verfahren keine neuen Iterationspunkte mehr liefern konnte und steckenblieb. Dieses Phänomen ließ sich auch nicht durch Veränderungen an den Parametern, etwa der Wahl von $\varepsilon_{\mathrm{newton}}$, γ oder von $\varrho_{\mathrm{faktor}}$ und $\varrho_{\mathrm{summand}}$, durch welche die Folge $(\hat{k}^s)_s$ erzeugt wird, korrigieren.

---

[1] Das Programm `gen` zum Generieren von Testproblemen wurde ebenfalls im Rahmen dieser Arbeit erstellt und ist im Anhang A.3 kurz beschrieben.

[2] unter SunOS 4.1.1

[3] Version 2.3.3, Compileroption –O2

[4] unter dem freien UNIX–Derivat Linux V0.99.14e

Es sollte an dieser Stelle erwähnt werden, daß die anderen beiden Varianten der Modifizierten Barrieremethode, d.h. die Variante KP–MBFM mit festem Barriereparameter und die Variante US–MBFM, numerisch stabil liefen.

Dies legt die Vermutung nahe, daß die Barrieremethode mit Parameteranpassung AP–MBFM mehr als theoretischer Anstoß für weitere Entwicklungen zu bewerten ist, nicht jedoch als fertiger praktischer Algorithmus. Verstärkt wird dieser Eindruck durch die Tatsache, daß die numerischen Resultate von Polayk et. al. in [19] ausschließlich durch die Modifizierte Barrieremethode mit konstantem Barriereparameter gewonnen wurden.

Wir beschränken uns daher im folgenden auf die Diskussion der Ergebnisse aus Algorithmus KP–MBFM, d.h. der Modifizierten Barrieremethode mit konstantem Barriereparameter, und aus Algorithmus US–MBFM, der mit einem unzulässigen Startpunkt initialisiert werden kann und den Barriereparameter dann erhöht.

Die Mininimierung der Modifizierten Barrierefunktionen werden wir des öfteren als (unrestringierte) Minimierungsprobleme bezeichnen, um Verwechslungen mit der Lösung des jeweiligen linearen oder quadratischen Programmes zu vermeiden. Bei der Darstellung der Ergebnisse in Form von Tabellen verwenden wir aus Platzgründen folgende Abkürzungen:

n	:	Anzahl der Unbekannten des jeweiligen Problems
Newton	:	Anzahl der insgesamt benötigten Newton–Schritte
Up	:	Anzahl der Multiplikator–Updates. Wir erinnern hier daran, daß die Anzahl der Multiplikator–Updates im Fall des Modellalgorithmus 4.1.1 gerade die Anzahl der benötigten Iterationen angibt.
$\mathcal{U}_{\text{primär}}$	:	Primäre Unzulässigkeit bei Abbruch
$\mathcal{U}_{\text{dual}}$	:	Duale Unzulässigkeit bei Abbruch
q_{ende}	:	Primärer Funktionswert bei Abbruch
η_{ende}	:	Dualer Funktionswert bei Abbruch
Dualitätslücke	:	Relative Dualitätslücke $\frac{q_{\text{ende}}-\eta_{\text{ende}}}{\max(1,q_{\text{ende}})}$ bei Abbruch
k	:	Verwendeter (konstanter) Barriereparameter beim Algorithmus KP–MBFM
Newtonschritte gesamt	:	Gesamtanzahl von Newton–Schritten bis zum Abbruch. Diese Angabe entspricht *Newton*
Newtonschritte ohne 1. Problem	:	Anzahl der Newton–Schritte ab dem ersten Multiplikator–Update bis zum Abbruch
Newtonschritte im 1. Problem	:	Anzahl der Newton–Schritte, die nötig sind um beim Algorithmus KP–MBFM das Minimum von $\mathcal{F}(.,\lambda^0,k) = \mathcal{M}(.,k)$ zu finden

Newtonschritte im 2. Problem	:	Anzahl der Newton–Schritte zur approximiativen Lösung von $\min\limits_x \mathcal{F}(x, \lambda^1, k)$
Newtonschritte letztes Problem	:	analog für das letzte Minimierungsproblem
Ø	:	Durchschnittliche Anzahl von Newton–Schritten pro (unrestringiertem) Minimierungsproblem
Ø *ohne 1. Problem*	:	Durchschnittliche Anzahl von Newton–Schritten im zweiten bis zum letzten (unrestringrierten) Minimierungsproblem
θ	:	Mit θ bezeichnen wir den Faktor, mit dem die relative Dualitätslücke pro Newtonschritt vermindert wird. Sei dazu u die relative Dualitätslücke nach dem ersten Multiplikator–Update und u_{ende} die relative Dualitätslücke bei Abbruch. Sei N die Anzahl der Newtonschritte ab dem ersten Update bis zum Abbruch. Dann definieren wir θ durch $\theta := (\frac{u_{\text{ende}}}{u})^{\frac{1}{N}}$. Den Wert θ werden wir auch des öfteren die Qualität der Newton–Schritte nennen.
k_{start}	:	Barriereparameter bei Start des Algorithmus US–MBFM
k_{ende}	:	Barriereparameter beim Abbruch des Algorithmus US–MBFM
Barriere–Erhöhungen	:	Anzahl der Update–Schritte, bei denen der Barriereparameter in Algorithmus US–MBFM echt erhöht wird

Bei der Bewertung der Tabellen, die im folgenden vorgestellt werden, ist zu beachten, daß der rechnerische Aufwand für die Modifizierte Barrieremethode vor allem durch die Anzahl der Newton–Schritte bestimmt wird, die bis zum Abbruch benötigt werden. Sei dazu wie bisher n die Anzahl der Variablen und m die Anzahl der Nebenbedingungen des quadratischen Programmes (QP). Die Berechnung eines Newton–Schrittes durch Lösen des Gleichungssystems (6.3) mit Hilfe des Cholesky–Verfahrens erfordert dann $O(n^3)$ wesentliche Operationen. Sind die Matrizen Q und A dünn besetzt, so kann man den Aufwand durch geeignete Techniken, wie etwa symbolische Faktorisierung von $Q + kA^T\Delta(x,k)^{-2}\Lambda A$ und Verwendung von Update–Formeln für diese Matrix nach Änderung von λ und/oder k, verringern (siehe etwa [10]). Ein Multiplikator–Update benötigt $O(m)$ wesentliche Operationen. Im Regelfall ist nicht nur $m \ll n^3$ sondern sogar $m \ll n^2$, so daß auch unter Ausnutzung der speziellen Struktur von Q und A die Berechnung eines Newton–Schrittes mehr Aufwand erfordert als ein Multiplikator–Update.

Bemerkung 6.2.1 *Wir möchten an dieser Stelle darauf hinweisen, daß in der Implementierung* `modbfm` *eine eventuell vorhandene spezielle Struktur der Matrizen Q oder A <u>nicht</u> ausgenutzt wurde. Bei den verwendeten Testproblemen war darüberhinaus lediglich bei dem Zuordnungsproblemen (siehe Abschnitt 6.2.2) der Fall gegeben, daß A dünn besetzt war und eine spezielle Struktur aufwies. Bei allen anderen Testproblemen* *war A jeweils dicht besetzt.*

6.2.1 Klee–Minty Probleme

Diese Klasse von Problemen wurde ursprünglich von Klee und Minty in [22] vorgestellt. Zur Lösung dieser Probleme benötigt die Simplexmethode mit verschiedenen Pivotregeln eine exponentielle Anzahl von Simplex–Schritten. Die hier verwendete Form dieser Probleme kann man etwa in [15] oder auch in [18] finden.

$$
\begin{aligned}
(\mathrm{LP})_{\mathrm{Klee-Minty}} \quad & \min - \sum_{j=1}^{n} \varepsilon^{n-j} x_j \\
x: \quad & 2 \sum_{j=1}^{i-1} \varepsilon^{i-j} x_j + x_i \leq 1, \quad i = 1, 2, \ldots, n \\
& x \geq 0
\end{aligned}
$$

Dabei ist $0 < \varepsilon < \frac{1}{2}$ eine Konstante; wir haben hier $\varepsilon := 0.4$ wie in [18] gewählt. Die Optimallösung des linearen Programms $(\mathrm{LP})_{\mathrm{Klee-Minty}}$ ist $x^* = (0, \ldots, 0, 1)^T$, der optimale Funktionswert ist $q^* = -1$.

Es wurden bei allen Rechnungen $\varepsilon_{\mathrm{Newton}} := 10^{-3}$ und $\gamma := 0.99$ gesetzt. Die Abbruchschranken wurden durch $\delta_{\text{primär}} := \delta_{\mathrm{dual}} := \delta_{\mathrm{kompl}} := 10^{-7}$ festgelegt. Für den Algorithmus KP–MBFM wurde $M := 100$ bei der Transformation von (QP) auf $(\mathrm{QP})^{(\mathrm{M})}$ gewählt. Die Variante US–MBFM wurde mit dem unzulässigen Startvektor $e = (1, \ldots, 1)^T$ initialisiert und es wurden $\varrho_{\mathrm{faktor}} := 3$ und $\varrho_{\mathrm{summand}} := 0$ gesetzt.

Die Tabelle 6.1 zeigt die Ergebnisse für den Algorithmus KP–MBFM mit konstantem Barriereparameter $k = 100$. Es sind die Anzahl der Variablen des Problems, die Gesamtanzahl der Newton–Schritte, die primäre und duale Unzulässigkeit, sowie der primäre und duale Funktionswert bei Abbruch des Programms aufgeführt. Aus Tabelle 6.2 kann man die entsprechenden Werte für den Barriereparameter $k = 1000$ entnehmen.

Man sieht aus beiden Tabellen, daß die benötigte Anzahl von Newton–Schritten langsam mit der Problemgröße steigt. Das Wachstum ist sublinear und die Anzahl der Newton–Schritte für $n = 32, 64, 128$ im wesentlichen konstant.

Sieht man von den Resultaten für das Klee–Minty Problem mit 4 Variablen bei konstantem Barriereparameter $k = 1000$ ab, so werden in jedem Fall weniger als 10 Multiplikator–Updates benötigt. Da die Anzahl der Multiplikator–Updates der Anzahl der Iterationen des Modellalgorithmus 4.1.1 entspricht, ist dies besser, als es die theoretischen Konvergenzresultate von Polyak aus Abschnitt 5.1 erwarten lassen. Die Ergebnisse aus Abschnitt 5.1 legen nahe, daß für größere Barriereparameter weniger Updates benötigt werden. Dies bestätigt sich für die Klee–Minty Probleme in der Praxis, wie aus den Tabellen 6.1 und 6.2 zu ersehen ist. Die größere Anzahl von Updates

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	q_{ende}	η_{ende}
4	25	6	1.34372768375e-12	1.76323336487e-08	-0.999999999829	-1
8	31	6	4.34195841954e-12	9.61938084743e-09	-0.999999999507	-1.00000000001
16	50	7	4.56107376179e-13	6.63323262984e-09	-0.999999999933	-1
32	64	6	2.56515326177e-11	7.86204878978e-10	-1.000000000140	-1.00000000660
64	62	6	1.13074324769e-10	7.29778903974e-10	-1.000000000870	-1.00000002922
128	63	6	2.75876306917e-10	1.15527498679e-09	-1.000000002230	-1.00000007128

Tabelle 6.1: Klee–Minty Probleme mit festem Barriereparameter $k = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	q_{ende}	η_{ende}
4	30	12	3.79577414970e-20	7.10542735760e-13	-1	-1
8	47	8	2.22044604925e-16	3.01287883531e-11	-1	-1
16	51	7	8.51733763895e-20	4.20369499210e-08	-0.99999999994	-1
32	68	4	2.41541578590e-13	8.17657053176e-12	-1	-1.00000000062
64	69	4	1.17696682953e-12	6.24211793365e-12	-1.00000000001	-1.00000000297
128	68	4	3.04815206575e-12	6.24034157681e-12	-1.00000000002	-1.00000000766

Tabelle 6.2: Klee–Minty Probleme mit festen Barriereparameter $k = 1000$

k	*Newtonschritte*							*Up*	θ
	gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	Ø	*Ø ohne 1. Problem*		
10	67	59	8	4	2	4.7857	4.5385	13	0.740899
20	65	56	9	10	3	5.9090	5.6000	10	0.715179
30	50	41	9	11	3	5.5555	5.1250	8	0.660967
40	65	54	11	11	4	7.2222	6.7500	8	0.703349
50	67	50	17	11	4	8.3750	7.1429	7	0.703960
100	64	50	14	11	6	9.1426	8.3333	6	0.694750
500	61	39	22	7	9	12.2000	9.7500	4	0.664285
1000	68	39	29	8	9	13.6000	9.7500	4	0.625471
2000	67	32	35	9	10	16.7500	10.6667	3	0.654903
5000	77	37	40	9	13	19.2500	12.3333	3	0.656084

Tabelle 6.3: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{LP})_{\text{Klee-Minty}}$

für 4 bzw. 8 Variablen in Tabelle 6.2 rührt daher, daß zur Lösung der letzten (unrestringierten) Minimierungaufgaben jeweils nur ein bis zwei Newtonschritte erforderlich waren, bis das Kriterium zur Beendigung des Newton–Verfahrens erfüllt war.

Um genauer zu untersuchen, welchen Einfluß die Größe des Barriereparameters auf das Verhalten der Modifizierten Barrieremethode KP–MBFM besitzt, wurde das Klee–Minty Problem mit 32 Variablen ausgewählt und für eine Zahl von verschiedenen Barriereparametern gelöst. Tabelle 6.3 zeigt die numerischen Resultate. Aufgeführt sind die Gesamtanzahl der Newton–Schritte, die Anzahl der Newton–Schritte ohne das erste (unrestringierte) Problem, die Anzahl im ersten, zweiten und letzten Problem, sowie die durchschnittliche Anzahl der Newton–Schritte, die nötig war, um das Abbruchkriterium bei der Lösung der (unrestringierten) Minimierungsaufgaben zu erfüllen. Die letzten beiden Spalten zeigen die Anzahl der Multiplikator–Updates sowie den Faktor θ, mit dem die relative Dualitätslücke pro Newton–Schritt verringert wurde (vgl. Seite 109).

Die Ergebnisse aus Tabelle 6.3 bestätigen sowohl die theoretischen Konvergenzresultate aus Abschnitt 5.1 als auch die ersten Beobachtungen, daß die Anzahl der Multiplikator–Updates mit steigendem Barriereparameter sinkt. Zu beobachten ist auch, daß die Gesamtzahl der Newton–Schritte und der Durchschnitt mit wachsendem Barriereparameter schwach ansteigt. Dies deckt sich mit der Erwartung, daß die Minimierung der Modifizierten Barrierefunktionen für größere Barriereparameter (wie bei den klassischen Barrierefunktionen) einen höheren Aufwand erfordert. Zu beachten ist, daß vor vor allem der Aufwand zur (approximativen) Lösung des ersten (unrestringierten) Minimierungsproblems mit dem Barriereparameter ansteigt. Der Aufwand zur Lösung des zweiten Problems ist dann bereits deutlich geringer, wie man Tabelle 6.3 entnehmen kann.

Interessant bei der Bewertung ist, daß sich die Qualität der Newton–Schritte, die wir hier an dem Faktor messen, mit dem die relative Dualitätslücke vermindert wird, mit steigendem Barriereparameter verbessert.

Ein weiterer Aspekt sollte hier zur Sprache kommen: die numerische Stabilität des linearen Systems (6.3) zur Berechnung des Newton–Schrittes. Sei $\tilde{\zeta}$ die numerische Lösung von

$$LL^T\zeta = -\nabla_x\mathcal{F}(x,\lambda^\nu,k).$$

Dabei sei $LL^T = \nabla^2_{xx}\mathcal{F}(x,\lambda^\nu,k)$ die Cholesky–Zerlegung von $\nabla^2_{xx}\mathcal{F}(x,\lambda^\nu,k)$. Dann definieren wir den absoluten Fehler f_{abs} bei der Berechnung von $\tilde{\zeta}$ als

$$f_{\text{abs}} := \|LL^T\tilde{\zeta} + \nabla_x\mathcal{F}(x,\lambda^\nu,k)\|_2$$

und den relativen Fehler f_{rel} als

$$f_{\text{rel}} := \frac{\|LL^T\tilde{\zeta} + \nabla_x\mathcal{F}(x,\lambda^\nu,k)\|_2}{\|\tilde{\zeta}\|_2} = \frac{f_{\text{abs}}}{\|\tilde{\zeta}\|_2}.$$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	q_{ende}	η_{ende}
4	33	7	0	6.50788159473e-10	-0.999999999986	-1.00000000001
8	39	11	3.72387220621e-11	3.14585308974e-10	-0.999999999998	-1
16	54	10	0	6.67739079887e-08	-0.999999996286	-1.00000000171
32	69	10	0	1.08821719157e-09	-0.999999999392	-1.00000006293
64	72	11	0	4.41022232621e-09	-0.999999999322	-1.00000000697
128	60	11	0	5.29069245490e-08	-0.999999997056	-1.00000001819

Tabelle 6.4: Klee–Minty Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
4	0.333778	100.00	7
8	0.316226	100.00	7
16	0.315790	100.00	7
32	0.315789	100.00	7
64	0.315789	100.00	7
128	0.315789	100.00	7

Tabelle 6.5: Klee–Minty Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

Bei der Lösung der Klee–Minty Probleme durch den Algorithmus KP–MBFM lag der relative Fehler f_{rel} für kleinen Barriereparameter $k \leq 500$ im Bereich 10^{-17} bis 10^{-11}. Für $k = 5000$ traten relative Fehler zwischen 10^{-16} und 10^{-8} auf. Dies bestätigt die Beobachtung von Polyak aus [28], daß die numerische Stabilität des Verfahrens mit wachsendem Barriereparameter sinkt.

Die Klee–Minty Probleme wurden ebenfalls mit Hilfe von Algorithmus US–MBFM gelöst. Wie oben bereits erwähnt, wurde als unzulässiger Startvektor $e = (1, \ldots, 1)^T$ gewählt. Dies führte je nach Problemgröße zu einem Start–Barriereparameter zwischen 0.31 und 0.34. Zunächst wurden die Barriereparameter von oben durch eine Konstante $k_{\max}$ beschränkt. Die Ergebnisse für $k_{\max} = 100$ sind in den Tabellen 6.4 und 6.5, diejenigen für $k_{\max} = 1000$ in den Tabellen 6.6 und 6.7 aufgeführt. Anschließend wurde die Beschränkung der Barriereparameter aufgehoben und die Klee–Minty Probleme erneut durch den Algorithmus US–MBFM gelöst. Die Tabellen 6.8 und 6.9 zeigen die numerischen Ergebnisse für diesen Fall.

Wie bei der Modifizierten Barrieremethode mit konstantem Barriereparameter KP–MBFM steigt die Anzahl der benötigten Newton–Schritte beim Algorithmus US–MBFM sublinear mit der Problemgröße. Im Vergleich zum Algorithmus KP–MBFM ist der Aufwand (gemessen in Newton–Schritten) dabei aber geringfügig höher. Auffällig ist auch, daß in der Iteration von US–MBFM mehr Multiplikator–Updates erfolgen als bei festem Barriereparameter in KP–MBFM. Zu beobachten ist, daß die Anzahl der Multiplikator–Updates offenbar nicht so stark davon abhängt, ob die Folge der Barriereparameter von oben durch ein $k_{\max}$ beschränkt ist. Man kann jedoch auch bei US–

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	q_{ende}	η_{ende}
4	33	10	1.24937560300e-10	1.19591989645e-08	-1.000000000010	-1
8	40	12	0	4.74989645532e-09	-0.999999999994	-1
16	56	13	7.11904346656e-10	1.78950073522e-08	-0.999999999986	-1
32	71	10	0	1.30670463586e-10	-0.999999999986	-1.00000000623
64	71	10	0	1.30670463774e-10	-0.999999999986	-1.00000002666
128	66	10	0	1.30617427367e-10	-0.999999999985	-1.00000006750

Tabelle 6.6: Klee–Minty Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 1000$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
4	0.333778	1000.00	10
8	0.316226	1000.00	9
16	0.315790	1000.00	10
32	0.315789	1000.00	10
64	0.315789	1000.00	10
128	0.315789	1000.00	10

Tabelle 6.7: Klee–Minty Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 1000$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	q_{ende}	η_{ende}
4	35	11	2.61351083493e-10	8.97705570102e-08	-1.00000000002	-1
8	43	12	1.18658088984e-11	3.55891710329e-08	-1	-0.999999999999
16	58	15	1.56957199858e-09	1.26209192953e-08	-1	-0.999999999947
32	71	9	0	5.10117507061e-11	-0.999999999998	-1.000000004100
64	71	9	0	5.10117507165e-11	-0.999999999998	-1.000000017380
128	71	9	0	5.10117340954e-11	-0.999999999998	-1.000000043920

Tabelle 6.8: Klee–Minty Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
4	0.333778	39246.30	11
8	0.316226	111337.00	12
16	0.315790	300174.00	15
32	0.315789	4117.61	9
64	0.315789	4117.61	9
128	0.315789	4117.61	9

Tabelle 6.9: Klee–Minty Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
4	29.1	6.4	4.44865349173e-12	6.35853325548e-09	2.18297602217e-11	0.241901
9	30.7	6.9	7.65831842386e-12	6.74435796100e-09	3.76146520470e-11	0.281705
16	29.9	6.6	2.35618080455e-13	2.27945218256e-10	1.75197346700e-12	0.242580
25	36.1	6.7	1.16740747793e-11	3.47921513821e-09	1.15520732296e-11	0.294066
36	34.8	6.3	7.18346808207e-13	9.13630060495e-10	1.23412618097e-10	0.268792
49	34.5	6.6	4.40086062281e-11	7.03946057001e-09	1.84576646619e-11	0.314150
64	35.3	5.8	1.60305720877e-10	1.14496970127e-08	8.29399252818e-11	0.393203
81	36.2	6.1	8.26509257352e-12	1.47835521602e-09	1.77345553790e-11	0.698202

Tabelle 6.10: Zuordnungs–Probleme bei konstantem Barriereparameter $k = 100$

MBFM zu sehen, daß für größere Barriereparameter letztendlich weniger Multiplikator–Updates erfolgen.

Interessant bei der Praxis des Algorithmus US–MBFM ist, wie schnell man den Barriereparameter wachsen lassen kann. Bei den Klee–Minty Problemen konnte fast bei jedem Update der Barriereparameter verzehnfacht werden. Da wir $\varrho_{\text{faktor}} = 3$ gewählt hatten, bedeutete dies eine Verdreifachung bei jedem Update–Schritt. Aus der Tabelle 6.9 ersieht man, daß die Barriereparameter für die Klee–Minty Probleme mit $n \geq 32$ Variablen bis zum Abbruch nicht so groß wurden wie für die kleineren Probleme.

Vergleicht man die Tabellen für US–MBFM mit denjenigen für KP–MBFM, so kann man beobachten, daß die primäre Unzulässigkeit bei Abbruch von US–MBFM in der Regel niedriger lag als bei konstantem Barriereparameter in KP–MBFM.

6.2.2 Zuordnungs–Probleme

Die Zuordnungs–Probleme (Assignment Problems) wurden als Beispiele von Problemen gewählt, bei denen die Matrix der Nebenbedingungen dünn besetzt ist, aber zugleich eine gewisse Struktur aufweist. Zudem sind die Zuordnungs–Probleme stark degeneriert.

$$
\begin{array}{rl}
(\mathrm{LP})_{\text{Zuord}} & \min - \sum_{i=1}^{r} \sum_{j=1}^{r} c_{ij} x_{ij} \\
x: & \sum_{j=1}^{r} x_{ij} \leq 1, \quad i = 1, 2, \ldots, r \\
 & \sum_{i=1}^{r} x_{ij} \leq 1, \quad j = 1, 2, \ldots, r \\
 & x \geq 0
\end{array}
$$

Die c_{ij} sind zufällige rationale Zahlen aus dem Intervall $(0, 1)$. Die Anzahl n der Unbekannten eines solchen linearen Zuordnungsproblems ist $n = r^2$.

Für $n \in \{r^2 : r = 2, \ldots, 9\}$ wurden jeweils zehn zufällige Testprobleme erzeugt und die

k	*Newtonschritte* gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	Ø	*Ø ohne 1. Problem*	*Up*	θ
10	45	33	12	10	1	3.7500	3.0000	11	0.439444
20	37	24	13	11	1	4.1111	3.0000	8	0.371585
30	34	21	13	11	1	4.2500	3.0000	7	0.339547
40	35	21	14	12	1	4.3750	3.0000	7	0.305376
50	34	20	14	12	1	4.8571	3.3333	6	0.375925
100	34	19	15	12	1	4.8571	3.1667	6	0.268790
500	39	23	16	6	1	4.8750	3.2857	7	0.465630
1000	41	24	17	6	1	4.5556	3.0000	8	0.464403
2000	50	34	17	5	1	4.2500	3.0909	11	0.555787
5000	51	33	18	5	1	4.2500	3.0000	11	0.533383

Tabelle 6.11: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{LP})_{\mathrm{Zuord}}$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
4	26.2	9.6	4.30124824646e-11	7.54684345849e-08	1.52307977341e-09	0.362567
9	34.3	10.4	5.61839463842e-12	8.74145422713e-10	1.56653244366e-10	0.441612
16	31.7	10.9	1.23167254173e-10	1.78551975982e-08	5.44252152668e-10	0.454263
25	39.3	10.6	9.05084658203e-10	4.61122318107e-08	2.70397265198e-10	0.507873
36	36.2	10.7	7.88869544648e-10	9.18549087903e-08	4.05622416258e-09	0.552744
49	41.5	13.6	1.78734278920e-12	1.95735649910e-11	3.11996896299e-14	0.424182
64	45.3	14.2	4.70432017387e-12	2.11668460537e-11	6.77437963044e-15	0.438033
81	44.8	14.5	5.67427480328e-11	5.23409982023e-10	1.63489740016e-13	0.463468

Tabelle 6.12: Zuordnungs–Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
4	0.40000	100.00	6.3
9	0.22222	100.00	6.1
16	0.15385	100.00	7.5
25	0.11766	100.00	7.2
36	0.09524	100.00	7.9
49	0.08000	100.00	8.6
64	0.06897	100.00	14.1
81	0.06061	100.00	14.2

Tabelle 6.13: Zuordnungs–Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
4	26.2	9.9	1.29964927709e-11	1.43952901163e-06	1.60216736544e-08	0.405559
9	37.3	12.4	6.70621156497e-13	3.04519204297e-09	1.14788241843e-11	0.439388
16	31.6	10.1	1.23856787286e-09	1.43569191713e-08	1.07743352182e-08	0.556721
25	39.9	12.2	8.70539136274e-13	2.24394289816e-08	9.44200330842e-13	0.436736
36	38.9	13.8	6.94329559496e-13	2.71514774552e-09	7.58958549136e-11	0.506546
49	40.6	11.1	9.88328980428e-10	4.39473083036e-08	3.66160633361e-11	0.509187
64	48.3	12.7	1.09400157294e-11	1.03116506889e-08	1.09824648597e-11	0.554036
81	44.1	12.5	6.57029625942e-11	7.30880584388e-08	5.13236624000e-10	0.572851

Tabelle 6.14: Zuordnungs–Probleme bei Start mit unzulässigem Vektor ohne Barriere-beschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
4	0.40000	7873.20	9.9
9	0.22222	84216.60	12.4
16	0.15000	6970.12	10.1
25	0.11766	45015.80	12.2
36	0.09524	111024.00	13.8
49	0.08000	8858.94	11.1
64	0.06897	22566.10	12.7
81	0.06061	19306.60	12.5

Tabelle 6.15: Zuordnungs–Probleme bei Start mit unzulässigem Vektor ohne Barriere-beschränkung

Algorithmen KP–MBFM und US–MBFM für diese Probleme getestet. Die Parameter der Verfahren wurden dabei wie bei den Klee–Minty Problemen gesetzt. Die hier vorgestellten Resultate ergaben sich dann als Mittel aus den individuellen Ergebnissen.

In Tabelle 6.10 sind die Ergebnisse für den Algorithmus KP–MBFM mit konstantem Barriereparameter $k = 100$ aufgeführt. Die Tabelle enthält die Anzahl der Variablen der entsprechenden Probleme, die Gesamtzahl von Newton–Schritten, die Anzahl der Multiplikator–Updates, die primäre und duale Unzulässigkeit und die relative Dualitätslücke bei Abbruch. Die letzte Spalte zeigt den Faktor θ, mit dem die Dualitätslücke pro Newton–Schritt verringert wird.

Die Anzahl der benötigten Newton–Schritte steigt wieder sehr langsam mit der Größe des behandelten Problems. Sie wächst dabei sogar noch langsamer als bei den Klee–Minty Problemen und ist für $n \geq 25$ fast konstant. Weiterhin kann man aus der Tabelle ersehen, daß die Anzahl der Newton–Schritte für ein durchschnittliches Zuordnungsproblem etwa um die Hälfte geringer ist als für ein Klee–Minty Problem gleicher Größe. Die Anzahl der Multiplikator–Updates hingegen war im wesentlichen für die Klee–Minty Probleme und die Zuordnungsprobleme identisch und lag etwa bei 6.

Um auch für die Zuordnungsprobleme den Einfluß des Barriereparameters zu untersuchen, wurde ein zufälliges Problem mit $n = 36$ Variablen erzeugt und durch den Algorithmus KP–MBFM mit verschiedenen Barriereparametern gelöst. Die Tabelle 6.11 zeigt die dabei erhaltenen Ergebnisse. Das Format dieser Tabelle entspricht dem von Tabelle 6.3 bei den Klee–Minty Problemen.

Zunächst ist aus Tabelle 6.11 zu sehen, daß die Qualität θ eines Newton–Schrittes für das Zuordnungs–Problem höher liegt als bei dem Klee–Minty Problem des letzten Abschnittes. Dies ist auch zu erwarten, da die Zuordnungs–Probleme generell nur halb so viele Newton–Schritte benötigten wie die Klee–Minty Probleme gleicher Größe. Im Gegensatz zu den Klee–Minty Problemen wächst bei den Zuordnungs–Problemen die Gesamtzahl der Newton–Schritte nicht so stark mit dem verwendeten Barriereparameter. Wie die drittletzte Spalte zeigt, bleibt die durchschnittliche Anzahl von Schritten pro (unrestringiertem) Minimierungsproblem sogar im wesentlichen konstant bei 3, wenn man die Schritte im ersten Minimierungsproblem jeweils außer acht läßt.

Erstaunen mag die Tatsache, daß die Anzahl der Multiplikator–Updates bei dem betrachteten Zuordnungsproblem mit steigendem Barriereparameter zunächst wie erwartet sinkt, dann jedoch wieder wächst. Zur Lösung der letzten drei bis vier (unrestringierten) Minimierungsprobleme genügte, ausgehend vom letzten Iterationspunkt x^ν, ein einziger Newton–Schritt, da sich x^ν bereits sehr nahe an der Optimallösung befand. Nach dem Multiplikator–Update war dann jeweils eines der drei Abbruchkriterien knapp nicht erfüllt, so daß weitere Iterationen erfolgten. Dieses Verhalten konnte in dieser Form nur bei den Zuordnungs–Problemen festgestellt werden.

Zu erwähnen ist noch, daß sich die Größe des relative Fehlers f_{rel} bei der Lösung des linearen Gleichungssystems zur Bestimmung des Newton–Schrittes in der gleichen Größenordnung wie bei den Klee–Minty Problemen bewegte: Für $k \leq 500$ war der relative Fehler von der Größe 10^{-17} bis 10^{-10}, für $k = 5000$ lag er im Bereich 10^{-15} bis 10^{-9}.

Anschließend an die Versuche mit dem Algorithmus KP–MBFM wurde wieder der Algorithmus US–MBFM auf die gleichen Testprobleme angewandt. Wiederum wurde als Startvektor der unzulässige Vektor $e = (1, \ldots, 1)^T$ verwendet. Die Tabellen 6.12 und 6.13 zeigen die Ergebnisse für den Fall, daß die Barriereparameter von oben durch $k_{\max} = 100$ beschränkt wurden. Aus den Tabellen 6.14 und 6.15 kann man die analogen Werte für unbeschränkte Barriereparameter entnehmen.

Der unzulässige Startvektor e führte zu durchschnittlichen Startbarrieren zwischen 0.06 und 0.4. Wie bereits bei den Klee–Minty Problemen erforderte der Algorithmus US–MBFM bei den Zuordnungsproblemen geringfügig mehr Newton–Schritte als KP–MBFM für die gleichen Testprobleme. Vergleicht man die Tabellen 6.12 und 6.14, so sieht man, daß die Anzahl der Newton–Schritte für US–MBFM im wesentlichen unabhängig davon war, ob die Barriereparameter beschränkt oder unbeschränkt gewählt wurden.

Deutlich ist der Unterschied zwischen KP–MBFM und US–MBFM bei der Bewertung der Newton–Qualität θ für die Zuordnungsprobleme. Während der Wert für KP–MBFM für $n \leq 64$ immer unter 0.4 liegt und für $n = 4, 9, 16, 36$ sogar kleiner als 0.3 ist, erreicht US–MBFM mit Barrierebeschränkung $k_{\max} = 100$ nur Werte um 0.45 und ohne Barrierebeschränkung nur um 0.5. Diese Differenzen sind darauf zurückzuführen, daß für θ der Quotient aus der relativen Dualitätslücke bei Abbruch und der Lücke nach dem ersten Multiplikator–Update mit dem Kehrwert der Anzahl von Newton–Schritten *ohne das erste Minimierungsproblem* potenziert wird. Wie wir bereits weiter oben festgestellt hatten, ist vor allem der Aufwand für das erste Minimierungsproblem davon abhängig, wie groß der erste Barriereparameter gewählt wurde. Für konstanten Barriereparameter in KP–MBFM ist das erste Minimierungsproblem daher deutlich aufwendiger zu lösen als bei US–MBFM, wo der unzulässige Startvektor eine kleine Startbarriere bedingt. Daher ist bei vergleichbarer Gesamtanzahl von Newton–Schritten die Anzahl der Newton–Schritte nach dem ersten Problem bis zum Abbruch für KP–MBFM deutlich geringer als für US–MBFM. Dies bedingt dann einen besseren Wert von θ für den Algorithmus KP–MBFM als für US–MBFM.

Im Vergleich zwischen den Algorithmen KP–MBFM und US–MBFM zeigte sich wie bei den Klee–Minty Problemen, daß in der Iteration von US–MBFM mehr Multiplikator–Updates erfolgen. Wie weiter oben bereits bemerkt wurde, wurden beim Algorithmus KP–MBFM (Tabelle 6.10) jeweils weniger als 7 Updates benötigt. Die Anzahl der Updates blieb dabei im wesentlichen unabhängig von der Problemgröße. Beim Algo-

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	34.3	5.2	1.81898940355e-12	1.22077126186e-10	1.72245783261e-12	0.323075
20	53.2	7.1	5.14955900144e-09	6.52165965676e-09	2.41608987105e-12	0.456612
30	68.4	8.0	7.27963833748e-08	1.02706013694e-08	3.09313414540e-12	0.558122
40	76.5	11.7	3.18172954748e-08	7.20606911919e-08	5.45624168556e-11	0.620049
50	99.1	22.8	7.70996848587e-08	4.18532408730e-08	6.30136746833e-12	0.680563

Tabelle 6.16: Zufällige lineare Probleme bei konstantem Barriereparameter $k = 100$

k	*Newtonschritte*							*Up*	θ
	gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	Ø	*Ø ohne 1. Problem*		
10	57	45	12	24	1	5.1818	4.5000	10	0.527914
20	54	37	17	22	1	6.0000	4.6250	8	0.505031
30	52	37	15	20	2	7.4286	6.1667	6	0.636069
40	43	29	14	21	1	6.1427	4.8333	6	0.565820
50	51	32	19	19	4	8.5000	6.4000	5	0.674601
100	44	28	16	21	1	7.3333	5.6000	5	0.585060
500	40	23	17	19	1	8.0000	5.7500	4	0.604717
1000	40	21	19	17	1	6.6667	4.2000	5	0.654300
2000	36	20	16	17	1	7.2000	5.0000	4	0.550976
5000	41	21	20	18	1	6.8333	4.2000	5	0.701995

Tabelle 6.17: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{LP})_{\text{Zufall}}$

rithmus US–MBFM mit Barrierebeschränkung (Tabelle 6.12) ist ein schwaches (sublineares) Wachstum der Anzahl der Updates mit der Zahl der Variablen erkennbar. Hingegen benötigt der Algorithmus US–MBFM ohne Barrierebeschränkung (Tabelle 6.14) zwar ebenfalls durchschnittlich mehr Updates als KP–MBFM, jedoch ist hier keine Tendenz in bezug auf die Problemgröße erkennbar.

6.2.3 Zufällige lineare Programme

Die zufälligen linearen Probleme sind von der Form

$$\begin{array}{rl} (\mathrm{LP})_{\text{Zufall}} & \min -\sum_{j=1}^{n} x_j \\ x: & Ax \leq 10^4 e \\ & x \geq 0, \end{array}$$

wobei A eine $n \times n$–Matrix ist, deren Einträge als zufällige ganze Zahlen aus dem Intervall $[1, 1000]$ gewählt wurden (siehe wiederum auch [15] oder [18]).

Für $n = 10, 20, 30, 40, 50$ wurden jeweils zehn zufällige Probleme erzeugt und die Algorithmen KP–MBFM und US–MBFM auf diese Probleme angewandt. Die in den

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	40.3	9.0	1.81898940355e-12	6.23791716392e-08	8.12202335323e-10	0.599652
20	51.8	6.4	5.45696821064e-12	3.37234987491e-08	3.11260021782e-08	0.689093
30	67.5	6.4	3.63797880709e-12	3.85637137224e-08	6.03714782872e-10	0.709261
40	86.8	11.2	9.09494701773e-12	2.29609727564e-08	1.21285501481e-11	0.794913
50	97.1	10.4	1.45825514457e-11	2.20604128298e-08	8.33249535325e-11	0.777272

Tabelle 6.18: Zufällige lineare Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	100.00	5.0
20	1.11111	100.00	5.1
30	1.11111	100.00	5.0
40	1.11111	100.00	5.2
50	1.11111	100.00	5.1

Tabelle 6.19: Zufällige lineare Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	36.4	9.8	2.74554809010e-08	1.98259861128e-08	1.30506070900e-07	0.627554
20	52.1	8.1	2.41780071519e-08	2.17553090687e-08	2.75617033401e-07	0.731251
30	67.2	10.4	1.81898940355e-12	4.30753034353e-08	9.93794618666e-09	0.744354
40	88.6	12.1	3.63797880709e-12	1.65434052796e-08	2.79086563679e-07	0.836169
50	97.3	10.2	7.27595761418e-12	6.58272349963e-08	1.93548755013e-10	0.784667

Tabelle 6.20: Zufällige lineare Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	10384.51	9.8
20	1.11111	9722.27	8.1
30	1.11111	17435.70	10.4
40	1.11111	40376.80	12.1
50	1.11111	1802.36	10.2

Tabelle 6.21: Zufällige lineare Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

Tabellen gezeigten Ergebnisse ergaben sich wieder als Durchschnitte aus den Resultaten für die individuellen Probleme.

Die Parameter für den Algorithmus KP–MBFM wurden wie bei den Klee–Minty Problemen und den Zuordnungsproblemen festgelegt. Für US–MBFM wurde als unzulässiger Startvektor $(-0.2, -0.2, \ldots, -0.2)^T$ gewählt. Der bei den letzten beiden Testproblemklassen verwendete Vektor $e = (1, \ldots, 1)^T$ erwies sich als ungeeignet für die zufälligen linearen Programme. Oft mußte für diesen Startvektor der erste Barriereparameter sehr klein (im Bereich 10^{-7} bis 10^{-5}) gewählt werden, und die vom Algorithmus US–MBFM gelieferte Folgen konvergierten nur extrem langsam oder überhaupt nicht. Dabei strebte vor allem die primäre Unzulässigkeit oftmals nicht so schnell gegen Null, daß die kleine Startbarriere genügend rasch erhöht werden konnte.

Tabelle 6.16 zeigt die Ergebnisse für den Algorithmus KP–MBFM mit konstantem Barriereparameter $k = 100$. Aus der zweiten Spalte ersieht man, daß für die zufälligen linearen Programme die Anzahl der benötigten Newton–Schritte im wesentlichen linear mit der Anzahl der Variablen steigt. Dies steht im Gegensatz zu den Klee–Minty Problemen und den Zuordnungs–Problemen, wo der Anstieg der Newton–Schritte sublinear war. Auch die Anzahl der Multiplikator–Updates steigt bei den zufälligen linearen Programmen deutlich stärker mit der Problemgröße als bei den zwei anderen betrachteten Klassen von linearen Programmen.

In Tabelle 6.17 ist wieder das Verhalten von KP–MBFM für verschiedene Barriereparameter notiert. Hier wurde ein zufälliges lineares Programm mit 32 Variablen verwendet. Hier zeigt sich in Anzahl der Multiplikator–Updates das gleiche Verhalten wie bei den Klee–Minty Problemen: Mit wachsendem Barriereparameter erfolgen weniger Multiplikator–Updates. Die Entwicklung der durchschnittlichen Anzahl von Newton–Schritten pro (unrestringiertem) Minimierungsproblem ist mit den Ergebnissen für das Zuordnungs–Problem des letzten Abschnittes vergleichbar. Läßt man bei der Durchschnittsbildung das erste (unrestringierte) Problem außer acht, so bleibt die durchschnittliche Anzahl von Newton–Schritten pro Minimierungsproblem weitgehend konstant, wenn der Barriereparameter erhöht wird. Dieses Verhalten konnte bereits bei den Zuordnungsproblemen beobachtet werden (siehe Tabelle 6.11). Im Hinblick auf die Qualität θ eines einzelnen Newtonschrittes (letzte Spalte der Tabelle 6.11) ist bei den zufälligen linearen Programmen keine klare Tendenz erkennbar. Die Werte schwanken zwischen 0.5 und 0.7.

Wie bei den beiden anderen Klassen von linearen Problemen konnte auch bei den zufälligen linearen Programmen beobachtet werden, daß der relative Fehler f_{rel} bei der Bestimmung des Newton–Schrittes durch Lösen des linearen Gleichungssystems (6.3) mit dem verwendeten Barriereparameter wächst.

Die Tabellen 6.18 und 6.19 zeigen die Ergebnisse für den Algorithmus US–MBFM, wobei die Bariereparameter durch $k_{\max} = 100$ von oben beschränkt wurden. Die entsprechenden Resultate für unbeschränkte Barriereparameter sind in den Tabellen 6.20 und 6.21 aufgeführt. Außer bei $n = 50$ benötigte der Algorithmus US–MBFM wieder geringfügig mehr Newton–Schritte zur Lösung der Probleme als KP–MBFM. Die Anzahl der Newton–Schritte war dabei wiederum im wesentlichen unabhängig davon, ob die Barriereparameter beschränkt oder unbeschränkt gewählt wurden. Der durchschnittliche Faktor θ, mit dem die Dualitätslücke pro Newton–Schritt vermindert wird (siehe letzte Spalte der Tabellen 6.18 und 6.20) lag wie bereits bei den Klee–Minty Problemen und den Zuordnungsproblemen deutlich über dem Wert des Algorithmus KP–MBFM mit festem Barriereparameter. Wie bereits auf Seite 119 dargelegt wurde, liegt dies daran, daß ein großer Teil der Newton–Schritte beim Algorithmus KP–MBFM im ersten (unrestringierten) Minimierungsproblem, also *vor* dem ersten Multiplikator–Update, benötigt wird, während beim Algorithmus US–MBFM relativ wenige Schritte in der ersten Minimierung erfolgen.

Bei den zufälligen linearen Programmen trat für den Algorithmus US–MBFM des öfteren der Fall ein, daß der Startbarriereparameter $k^0 = 1.1111$ bei den ersten zwei oder drei Multiplikator–Updates nicht um den Faktor drei erhöht werden konnte, da die primäre Unzulässigkeit noch zu groß war. Ab dem vierten Multiplikator–Update sank dann aber die primäre Unzulässigkeit so schnell, daß für den Rest der Iteration der Barriereparameter bei jedem Update um den Faktor drei wachsen konnte (sofern die Barriereparameter nicht von oben beschränkt wurden und bereits $k_{\max}$ erreicht war).

6.2.4 Least–Squares Probleme

Diese Probleme sind von der Form:

$$\begin{array}{rl} (\mathrm{QP})_{\mathrm{Square}} & \min \sum_{j=1}^{n} x_j^2 - \sum_{j=1}^{n} c_j x_j \\ x: & Ax \le 10^4 e \\ & x \ge 0. \end{array}$$

Wiederum ist A eine $n \times n$–Matrix mit zufälligen Einträgen aus dem Intervall $[1, 1000]$. Die c_j sind zufällige Zahlen aus dem Intervall $[-10, 10]$.

Für die numerischen Experimente wurden für $n = 10, 20, 30, 40, 50$ wieder je zehn zufällige Least–Squares Probleme erzeugt und die Algorithmen KP–MBFM und US–MBFM für diese Probleme getestet. Es zeigte sich dabei, daß $\varepsilon_{\mathrm{newton}} := 10^{-4}$ gesetzt werden mußte, um in allen Fällen Konvergenz zu erhalten. Für größeres $\varepsilon_{\mathrm{newton}}$ wuchs des öfteren während der Iteration die duale Unzulässigkeit und die duale Folge divergierte.

Bei den Testläufen konnte für einige wenige Probleme die duale Zulässigkeit während

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	41.8	5.7	2.72608893136e-10	7.09834097279e-07	1.41538908124e-10	0.347831
20	59.3	8.4	5.01027358490e-10	3.68406404050e-07	3.53397974748e-14	0.363697
30	62.2	7.1	4.46016201749e-09	4.28360141136e-07	1.40519452140e-09	0.494026
40	66.4	9.5	3.36514105470e-09	9.80210905333e-07	1.02339449723e-13	0.419209
50	69.9	4.2	4.00245477206e-08	5.28188110138e-07	7.85596097679e-08	0.676060

Tabelle 6.22: Least–Squares Probleme bei konstantem Barriereparameter $k = 100$

k	*Newtonschritte*							*Up*	θ
	gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	Ø	*Ø ohne 1. Problem*		
10	52	36	16	14	3	2.8889	2.1177	17	0.517604
20	55	25	30	13	1	3.0000	1.7857	14	0.294274
30	57	23	34	12	3	6.3333	2.8750	8	0.477001
40	53	20	33	12	1	5.8889	2.5000	8	0.286392
50	62	18	44	12	1	7.7500	2.5714	7	0.283852
100	59	16	43	11	1	8.4286	2.6667	6	0.253388
500	75	15	60	12	1	15.0000	3.7500	4	0.307026
1000	84	15	69	12	1	14.0000	3.0000	5	0.485258
2000	97	17	80	11	2	16.1667	3.4000	5	0.441799

Tabelle 6.23: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{QP})_{\mathrm{Squares}}$

der Iteration auch für $\varepsilon_{\text{newton}} = 10^{-4}$ nicht unter 10^{-7} gesenkt werden. Daher wurde für die hier vorgestellten Versuchsreihen das Abbruchkriterium δ_{dual} für die duale Unzulässigkeit durch $\delta_{\text{dual}} := 10^{-6}$ festgelegt. Die Abbruchkriterien $\delta_{\text{primär}}$ für die primäre Unzulässigkeit bzw. δ_{kompl} für die Komplementarität blieben wie bei den bisherigen Testreihen auf 10^{-7}. Der Algorithmus US–MBFM wurde wie bei den zufälligen linearen Problemen mit dem unzulässigen Startvektor $(-0.2, \ldots, -0.2)^T$ initialisiert.

In der Tabelle 6.22 sind die Ergebnisse für KP–MBFM mit konstanten Barriereparameter $k = 100$ festgehalten. Zu beachten ist hier, daß die Gesamtanzahl der benötigten Newton–Schritte wieder langsam mit der Anzahl der Variablen der Probleme steigt. Während noch ein größerer Sprung von $n = 10$ auf $n = 20$ beobachten ist, sind die

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	36.5	12.2	7.41756038223e-12	7.19466301646e-07	7.28629728163e-11	0.512702
20	61.3	8.1	6.43340172246e-08	6.35773445445e-07	3.46578636302e-08	0.836530
30	71.8	9.9	3.53611540049e-09	3.73158511202e-07	1.28862719015e-10	0.709490
40	69.5	10.3	2.9158033216e-09	3.11137678288e-07	1.81781748560e-11	0.667766
50	84.3	17.6	9.63336788118e-09	4.35067001092e-07	2.69229184659e-10	0.755719

Tabelle 6.24: Least–Squares Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	100.00	6.0
20	1.11111	100.00	5.1
30	1.11111	100.00	5.1
40	1.11111	100.00	5.3
50	1.11111	100.00	5.1

Tabelle 6.25: Least–Squares Probleme bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	39.5	12.8	2.69115178734e-16	2.15689111146e-07	5.47350590177e-08	0.703444
20	66.4	12.3	3.54822167955e-13	2.03020484868e-07	2.78804700937e-08	0.756461
30	56.3	10.4	3.63797880709e-12	3.72941009452e-07	4.83268570628e-10	0.638752
40	81.2	12.1	1.81898940355e-12	1.70582877157e-07	3.37936320790e-09	0.775426
50	75.9	11.8	3.63797880709e-12	5.61119501974e-07	3.78198786307e-09	0.754031

Tabelle 6.26: Least–Squares Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	98528.20	12.8
20	1.11111	85707.80	12.3
30	1.11111	9339.11	10.4
40	1.11111	56789.80	12.1
50	1.11111	25466.60	11.8

Tabelle 6.27: Least–Squares Probleme bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

Unterschiede für $n \geq 20$ sehr gering. Das Wachstum ist insgesamt höchstens linear.

Der Gesamtaufwand, gemessen an der Anzahl der Newton–Schritte für ein Least–Squares Problem, war damit ab $n = 30$ geringer und für $n = 40, 50$ deutlich kleiner als für ein zufälliges lineares Programm gleicher Größe. Auch der Faktor θ, mit dem im Durchschnitt die relative Dualitätslücke pro Newton–Schritt verringert wurde (siehe Spalte 7 der Tabelle 6.22), war bei den den Least–Squares Problemen geringer als bei den zufälligen linearen Programmen, d.h. die Qualität eines durchschnittlichen Newton–Schrittes war höher.

Das Verhalten von Algorithmus KP–MBFM für verschiedene Barriereparameter wurde an einem zufälliges Least–Squares Problem mit 32 Variablen getestet. Wir haben in Tabelle 6.23 die erhaltenen Resultate aufgelistet. Diese Ergebnisse stützen wieder die Theorie aus Abschnitt 5.1: Für größere Barriereparameter werden weniger Multiplikator–Updates bis zum Abbruch benötigt. Zu sehen ist auch wieder, daß vor allem der Aufwand für das erste (unrestringierte) Minimierungsproblem mit dem verwendeten Barriereparameter wächst. Läßt man das erste (unrestringierte) Minimierungsproblem außer acht, so wächst die durchschnittliche Anzahl von Newton–Schritten pro Minimierungsproblem (siehe drittletzte Spalte der Tabelle 6.23) sehr langsam mit dem verwendeten Barriereparameter.

Wir wollen hier noch erwähnen, daß wiederum der relative Fehler f_{rel} bei der Bestimmung des Newton–Schrittes mit dem Barriereparameter in gleichem Maße wie bei linearen Programmen wuchs.

Die Tabellen 6.24 und 6.25 zeigen die Ergebnisse für den Algorithmus US–MBFM mit beschränkten Barriereparametern, aus den Tabellen 6.26 und 6.27 kann man die analogen Werte für unbeschränkte Barriereparameter entnehmen. Es zeigte sich für den Algorithmus US–MBFM ohne Barrierebeschränkung keine so klare Tendenz für Gesamtanzahl von Newton–Schritten bei wachsender Problemgröße, wie etwa bei den bisherigen Problemen. Die Tabelle 6.26 läßt zwar den Schluß zu, daß die Zahl schwach mit der Problemgröße steigt, jedoch wurden im Schnitt für $n = 30$ zehn Newton–Schritte weniger benötigt als für $n = 20$. Gleichermaßen wurden für $n = 50$ im Schnitt etwa sechs Schritte weniger gebraucht als für $n = 40$.

Bei beschränkten Barriereparametern ist aus Tabelle 6.24 ein höchstens lineares Wachstum der Gesamtanzahl von Newton–Schritten erkennbar. Zwar wurden auch hier für $n = 40$ im Schnitt weniger Schritte benötigt als für $n = 30$, jedoch liegt die Differenz hier aller Wahrscheinlichkeit nach im Bereich der Streuung durch die zufällige Erzeugung der Probleme.

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	40.3	6.5	1.40036618778e-11	4.02845017344e-08	2.39978013273e-13	0.383687
20	66.2	7.4	3.93357227580e-09	9.20502651880e-08	9.74573326753e-13	0.465243
30	74.5	6.1	4.87671059091e-09	8.43367473635e-08	1.47374535375e-10	0.585079
40	78.6	6.1	2.11755712520e-08	1.25394450379e-08	3.03816336867e-08	0.560212
50	81.9	7.2	7.92897481006e-09	1.77072922725e-07	9.53709248688e-14	0.474986

Tabelle 6.28: Zufällige quadratische Probleme mit $\mu_{\min} = 0.0$, $\mu_{\max} = 1.0$ bei konstantem Barriereparameter $k = 100$

k	*Newtonschritte*							*Up*	θ
	gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	Ø	*Ø ohne 1. Problem*		
10	57	30	27	22	1	7.1250	4.2857	7	0.43414
20	59	27	32	22	1	8.5000	4.8250	6	0.47382
30	60	25	35	21	1	10.0000	5.0000	5	0.52342
40	64	27	37	20	1	8.7250	4.8250	6	0.53152
50	62	24	38	20	1	10.3333	4.8000	5	0.45246
100	67	22	45	17	2	13.4000	5.5000	4	0.49115
500	91	20	71	17	1	15.1667	4.0000	5	0.47866
1000	108	17	91	15	1	21.6000	4.2500	4	0.41982
2000	114	17	97	14	1	22.8000	4.2500	4	0.41083
5000	144	24	120	9	1	11.0769	2.0000	12	0.86574

Tabelle 6.29: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{QP})_{\mathrm{Zufall}}$ mit $\mu_{\min} = 0.0$ und $\mu_{\max} = 1.0$

6.2.5 Zufällige quadratische Programme

Die hier betrachteten zufälligen quadratischen Probleme besitzen die Form:

$$\begin{array}{rl} (\mathrm{QP})_{\mathrm{Zufall}} & \min \frac{1}{2} x^T (Q + Q^T) x + c^T x \\ x: & Ax \leq 10^4 e \\ & x \geq 0. \end{array}$$

Die Matrix A und der Vektor $c = (c_1, \ldots, c_n)^T$ sind dabei wie bei den Least–Squares Problemen. Die Matrix Q ist durch $Q = G^{-1} D G$ definiert, wobei G eine diagonaldominante Matrix ist, deren Einträge zufällig aus $[-10, 10]$ gewählt wurden. Weiterhin ist D eine Diagonalmatrix mit zufälligen Einträgen aus $[\mu_{\min}, \mu_{\max}]$ $(0 \leq \mu_{\min} \leq \mu_{\max})$.

Zunächst wurden für $n = 10, 20, 30, 40, 50$ jeweils zehn Testprobleme mit $\mu_{\min} = 0.0$ und $\mu_{\max} = 1.0$ erzeugt, und die Algorithmen KP–MBFM und US–MBFM für diese Probleme getestet. Die Parameter für KP–MBFM und US–MBFM wurden wie bei den Least–Squares Problemen gesetzt.

Die Tabelle 6.28 zeigt die Resultate für konstanten Barriereparameter $k = 100$. Die

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	37.4	9.1	4.30956507713e-12	4.25359126588e-07	4.15815397110e-11	0.585148
20	49.8	9.2	4.65661287308e-10	4.09441092586e-07	9.39278915864e-12	0.558758
30	65.6	11.5	5.60248736292e-10	1.74951667375e-07	2.70509764929e-11	0.662123
40	72.4	11.4	3.03225533571e-09	2.96100514774e-07	5.80931399091e-11	0.691783
50	80.2	10.9	9.05499650975e-09	1.07415338846e-07	1.42033719540e-10	0.687946

Tabelle 6.30: Zufällige quadratische Probleme mit $\mu_{\min} = 0.0$, $\mu_{\max} = 1.0$ bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	100.00	6.0
20	1.11111	100.00	6.2
30	1.11111	100.00	9.1
40	1.11111	100.00	9.2
50	1.11111	100.00	8.7

Tabelle 6.31: Zufällige quadratische Probleme mit $\mu_{\min} = 0.0$, $\mu_{\max} = 1.0$ bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	43.1	10.1	5.73254116340e-15	2.19484037611e-08	9.45951924886e-10	0.637141
20	51.3	9.6	3.63797880709e-12	8.13272977851e-08	1.90601350914e-10	0.611236
30	66.9	11.1	1.81898940355e-12	7.02512895856e-07	1.28275578535e-09	0.712725
40	73.5	11.2	1.45519152284e-11	6.91414053788e-07	8.64759895677e-09	0.753352
50	76.3	10.9	3.63797880709e-12	1.50918548236e-07	3.51077414243e-09	0.730262

Tabelle 6.32: Zufällige quadratische Probleme mit $\mu_{\min} = 0.0$, $\mu_{\max} = 1.0$ bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	8703.28	10.1
20	1.11111	5152.93	9.6
30	1.11111	4586.41	11.1
40	1.11111	15678.60	11.2
50	1.11111	7714.41	10.9

Tabelle 6.33: Zufällige quadratische Probleme mit $\mu_{\min} = 0.0$, $\mu_{\max} = 1.0$ bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	45.3	8.2	1.15467540305e-10	6.44947108497e-07	1.27546976108e-09	0.503734
20	63.2	7.1	2.36468622461e-09	2.83342785679e-07	6.87679551643e-13	0.486632
30	73.8	6.8	2.25646257235e-09	9.79174207494e-07	5.84863531892e-12	0.441012
40	88.4	8.1	4.81906563321e-09	5.21317645052e-07	6.93124622837e-08	0.588243
50	91.0	6.2	1.40766465151e-09	9.43823636135e-07	3.28669932644e-10	0.555576

Tabelle 6.34: Zufällige quadratische Probleme mit $\mu_{\min} = 1.0$, $\mu_{\max} = 2.0$ bei konstantem Barriereparameter $k = 100$

k	*Newtonschritte*							*Up*	θ
	gesamt	*ohne 1. Problem*	*im 1. Problem*	*im 2. Problem*	*letztes Problem*	*Ø*	*Ø ohne 1. Problem*		
10	50	26	24	14	2	7.1429	4.3333	10	0.454301
20	54	23	31	12	1	5.4000	2.5556	9	0.343182
30	55	24	31	13	1	6.1111	3.0000	8	0.362478
40	59	21	38	13	1	7.3750	3.0000	7	0.348756
50	55	21	34	13	1	6.8750	3.0000	7	0.324086
100	58	18	40	13	1	8.2857	3.0000	6	0.291266
500	63	14	49	11	1	10.5000	2.8000	5	0.345681
1000	80	16	64	11	2	13.3333	3.2000	5	0.524524
2000	80	15	65	9	1	10.0000	3.1429	5	0.395140
5000	120	17	103	7	1	13.3333	2.8725	5	0.416384

Tabelle 6.35: Einfluß des Barriereparameters auf das Verhalten bei $(\mathrm{QP})_{\text{Zufall}}$ mit $\mu_{\min} = 1.0$ und $\mu_{\max} = 2.0$

Tabellen 6.30, 6.31 sowie 6.32 und 6.33 enthalten die entsprechenden Ergebnisse für den Algorithmus US–MBFM. Als unzulässiger Startvektor wurde wie bereits bei den zufälligen linearen Programmen und den Least–Squares Problemen $(-0.2, \ldots, -0.2)^T$ gewählt.

Das Verhalten von KP–MBFM für verschiedene Barriereparameter wurde an einem Programm mit 32 Variablen getestet. Die Ergebnisse sind in Tabelle 6.29 notiert.

Zuletzt wurden für $\mu_{\min} = 1.0$ und $\mu_{\max} = 2.0$ wieder jeweils 10 Testprobleme mit $n = 10, 20, 30, 40, 50$ Variablen erzeugt und mit den Algorithmen KP–MBFM und US–MBFM gelöst. Die Ergebnisse für festen Barriereparameter $k = 100$ sind in der Tabelle 6.34 festgehalten. Für ein zufälliges Problem mit 32 Unbekannten ist das Verhalten von KP–MBFM mit verschiedenen Barriereparametern in der Tabelle 6.35 notiert. Die Tabellen 6.36, 6.37 sowie 6.38 und 6.39 zeigen die Ergebnisse für den Algorithmus US–MBFM.

Bei den zufälligen quadratischen Programmen konnte das gleiche Verhalten der Algorithmen KP–MBFM und US–MBFM wie bei den bisherigen Testproblemen beobachtet

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	41.1	9.6	4.06726030633e-09	4.65920775294e-07	1.69796455372e-10	0.560928
20	56.9	11.2	3.27418092638e-10	4.28274709807e-07	1.39987903399e-11	0.623132
30	67.3	13.3	1.27511157189e-09	1.33358110332e-07	2.59709600339e-10	0.701020
40	70.2	12.5	4.50890440409e-08	4.12181479170e-07	2.78916769836e-11	0.654175
50	77.7	16.7	8.85302142706e-09	4.24830574275e-07	4.15643244231e-10	0.735231

Tabelle 6.36: Zufällige quadratische Probleme mit $\mu_{\min} = 1.0$, $\mu_{\max} = 2.0$ bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	100.00	6.4
20	1.11111	100.00	7.5
30	1.11111	100.00	6.1
40	1.11111	100.00	6.3
50	1.11111	100.00	7.4

Tabelle 6.37: Zufällige quadratische Probleme mit $\mu_{\min} = 1.0$, $\mu_{\max} = 2.0$ bei Start mit unzulässigem Vektor und $k_{\max} = 100$

n	*Newton*	*Up*	$\mathcal{U}_{\text{primär}}$	$\mathcal{U}_{\text{dual}}$	*Dualitätslücke*	θ
10	42.5	11.5	8.43636293115e-14	5.40561405138e-07	1.94453734218e-08	0.716761
20	53.4	10.5	6.24596745579e-09	1.74775369821e-07	7.89224201366e-09	0.674237
30	67.1	14.2	7.97462145455e-18	5.26603449602e-05	3.35621299168e-08	0.780936
40	68.1	12.9	1.81898940355e-12	2.23853006354e-07	7.42242042080e-09	0.724074
50	73.6	13.1	1.81898940355e-12	2.16918271434e-07	6.78020377046e-08	0.780991

Tabelle 6.38: Zufällige quadratische Probleme mit $\mu_{\min} = 1.0$, $\mu_{\max} = 2.0$ bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

n	k_{start}	k_{ende}	*Barriere–Erhöhungen*
10	1.11111	97599.60	11.5
20	1.11111	12898.60	10.5
30	1.11111	679511.30	14.2
40	1.11111	28252.70	12.9
50	1.11111	67735.00	13.1

Tabelle 6.39: Zufällige quadratische Probleme mit $\mu_{\min} = 1.0$, $\mu_{\max} = 2.0$ bei Start mit unzulässigem Vektor ohne Barrierebeschränkung

werden. Die Anzahl der benötigten Newton–Schritte bis zum Programmabbruch steigt schwach (weniger als linear) mit der Problemgröße. Beim Algorithmus KP–MBFM sinkt die Anzahl der Multiplikator–Updates mit steigendem Barriereparameter. Dies entspricht den Ergebnissen, die wir für alle Problemklassen außer den Zuordnungsproblemen erhalten haben und untermauert die theoretischen Ergebnisse aus Abschnitt 5.1.

Bei den zufälligen quadratischen Programmen konnte wieder für den Algorithmus US–MBFM beobachtet werden, daß bei vielen Testproblemen der (niedrige) Startbarriereparameter $k = 1.11111$ bei den ersten zwei bis drei Multiplikator–Updates nicht um den Faktor drei erhöht werden konnte. In der Regel sank die primäre Unzulässigkeit erst ab dem vierten Update genügend schnell, um eine Verdreifachung des Barriereparameters zu ermöglichen.

Im Gegensatz zu den vier anderen Klassen von Testproblemen benötigte der Algorithmus US–MBFM bei den zufälligen quadratischen Programmen bis zum Abbruch weniger Newton–Schritte als die Variante KP–MBFM mit konstantem Barriereparameter. Am deutlichesten zeigt sich dies in den Ergebnissen für $\mu_{\min} = 1.0$ und $\mu_{\max} = 2.0$, die in den Tabellen 6.34, 6.36 und 6.38 festgehalten sind. Für $n = 40$ benötigte der Algorithmus US–MBFM mit oder ohne Beschränkung der Barriereparameter im Schnitt ungefähr 20 Newton–Schritte weniger als KP–MBFM. Für $n = 50$ liegt die Differenz etwa bei 15 Newton–Schritten.

6.2.6 Zusammenfassung der Ergebnisse

Wir fassen im folgenden noch einmal die wichtigsten Beobachtungen über das Verhalten der Algorithmen KP–MBFM und US–MBFM bei Anwendung auf die Testprobleme der Abschnitte 6.2.1 bis 6.2.5 zusammen:

1. Bei beiden Algorithmen stieg die Anzahl der Newton–Schritte, die bis zum Abbruch benötigt wurden, langsam mit der Problemgröße. Das Wachstum war höchstens linear, meistens jedoch sublinear. Am stärksten stieg die Anzahl der Newton–Schritte bei den zufälligen linearen Programmen.

2. Der Algorithmus US–MBFM, der mit einem unzulässigen Startvektor initialisiert wurde, benötigte bei den ersten vier Problemklassen geringfügig mehr Newton–Schritte als KP–MBFM. Eine Beschränkung der Barriereparameter bei US–MBFM hatte keinen sichtbaren Einfluß auf die Zahl der Newton–Schritte. Bei den zufälligen quadratischen Programmen war der Aufwand für US–MBFM hingegen kleiner als für KP–MBFM. Am deutlichsten war der Unterschied für größere n und größere Eigenwerte der Matrix D.

3. Beim Algorithmus KP–MBFM wurde ein großer Teil der Newton–Schritte im ersten (unrestringierten) Minimierungsproblem benötigt. In den folgenden Minimierungsproblemen war der Aufwand, gemessen in Newton–Schritten, dann deutlich geringer. Während der Iteration des Algorithmus US–MBFM erfolgten im ersten (unrestringierten) Minimierungsproblem deutlich weniger Newton–Schritte als bei KP–MBFM. Dies ist auf die kleinere Startbarriere zurückzuführen. Auch bei US–MBFM nahm die Anzahl der Newton–Schritte für die (unrestringierten) Minimierungsprobleme im Verlauf der Iteration ab. Hier war der Abfall aber nicht so stark wie bei KP–MBFM. Die Tatsache, daß beim Algorithmus KP–MBFM viele Newton–Schritte bei Minimierung der ersten Modifizierten Barrierefunktion benötigt wurden, führte zu einer besseren Bewertung der Newton–Qualität θ. Da wir mit θ den Faktor bezeichnen, mit dem die relative Dualitätslücke *nach dem ersten Multiplikator–Update* bis zum Abbruch verringert wurde, steht ein kleinerer Wert von θ für eine bessere Qualität der Newton–Schritte.

4. Die Anzahl der Multiplikator–Updates beim Algorithmus KP–MBFM wuchs allein bei den zufälligen linearen Programmen mit der Problemgröße. Bei allen anderen Problemklassen wurden selbst bei einem Barriereparameter von $k = 100$ meist deutlich weniger als 10 Updates benötigt. Beachtet man, daß für den Modellalgorithmus 4.1.1 die Anzahl der Updates gerade der Anzahl der Iterationen entspricht, so ist dies besser, als dies vielleicht die Ergebnisse von Polyak aus Abschnitt 5.1 zunächst erwarten lassen. Die Konstanten in den Beweisen von Polyak werden im wesentlichen durch das implizite Funktionentheorem geliefert. Man könnte erwarten, daß diese eine weit größere Wahl des Barriereparameters erfordern, um eine derart schnelle Konvergenz (im Sinne der Iterationen des Modellalgorithmus 4.1.1) zu erhalten. Die theoretischen Resultate von Polyak legen nahe, daß für größere Barriereparameter weniger Multiplikator–Updates erfolgen. Dies konnte bei den Testproblemen in der Praxis grundsätzlich verifiziert werden. Allein bei den Zuordnungs–Problemen stieg die Anzahl der Multiplikator–Updates wieder ab einer bestimmten Größe des Barriereparameters.

5. Mit wachsendem Barriereparameter vergrößerten sich auch die relativen Fehler f_{rel}, die bei der Berechnung der Newton–Schritte durch das lineares Gleichungssystem (6.3) auftraten. Dies bestätigt die Beobachtungen (und theoretischen Resultate) von Polyak aus [28]. Die Größe der relativen Fehler war für moderate Barriereparameter jedoch so gering (10^{-17} bis 10^{-9}), daß die Algorithmen US–MBFM und KP–MBFM numerisch sehr stabil liefen.

6. Der Algorithmus US–MBFM erforderte während der Iteration mehr Multiplikator–Updates als KP–MBFM. Die Anzahl der Updates bei US–MBFM stieg bei allen drei Klassen von linearen Problemen und bei den zufälligen quadratischen Pro-

grammen mit der Problemgröße.

7. Beim Algorithmus US–MBFM konnte bei den zufälligen linearen Programmen und den beiden Klassen von quadratischen Problemen beobachtet werden, daß die primäre Unzulässigkeit zunächst recht langsam gegen Null strebte. Dies führte dazu, daß des öfteren der Barriereparameter bei den ersten drei Multiplikator–Updates nicht verdreifacht werden konnte. Ab dem vierten Update war hingegen dann eine Verdreifachung der Barriere möglich. In diesem Zusammenhang traten dann auch Probleme auf, wenn der Startvektor so schlecht gewählt war, daß der erste Barriereparameter sehr klein (im Bereich 10^{-7} bis 10^{-5}) gewählt werden mußte. Die Konvergenz war dann sehr langsam, oder die von US–MBFM gelieferten Folgen konvergierten gar nicht.

Bemerkung 6.2.2 *Wir wollen an dieser Stelle abschließend darauf hinweisen, daß es beim Test der Algorithmen KP–MBFM und US–MBFM durchaus auch vereinzelt Probleme gab, bei denen die Algorithmen mit den gewählten Einstellungen nicht terminierten, oder die von den Algorithmen gelieferten Folgen divergierten. Durch Experimentieren mit den Parametern konnte dann jedoch in allen Fällen Konvergenz gegen eine Optimallösung und einen zugehörigen Lagrangemultiplikator erreicht werden. Beim Algorithmus KP–MBFM mußten in diesen Fällen die geforderte Genauigkeit* $\varepsilon_{\text{newton}}$ *beim Minimieren der Modifizierten Barrierefunktionen, der Faktor* γ*, mit dem* $\varepsilon_{\text{newton}}$ *bei jedem Update verringert wird, die Konstante* M *und der Barriereparameter* k *variiert werden. Beim Algorithmus US–MBFM war es nötig, neben unterschiedlichen Startvektoren auch mit verschiedenen Werten für* $\varepsilon_{\text{newton}}$*,* γ*,* ϱ_{faktor} *und* $k_{\max}$ *zu experimentieren.*

Kapitel 7

Fazit und Ausblick

Wir haben in dieser Arbeit sowohl die Theorie als auch praktische Aspekte der von Polyak vorgeschlagenen Modifizierten Barrieremethode untersucht. Dieses Verfahren basiert auf Modifizierten Barrierefunktionen, die ein Versuch sind, die Vorzüge der klassischen Barrierefunktionen mit denjenigen der erweiterten Lagrangefunktionen zu verbinden. Die Grundidee dieser Modifizierten Barrierefunktionen ist es, den zulässigen Bereich durch eine monotone Transformation der Nebenbedingungen von (QP) zu vergrößern und dann die Lagrangefunktion zu dem transformierten Problem zu bilden. Im Gegensatz zu den klassischen Barrierefunktionen existiert dann die Modifizierte Barrierefunktion an der Optimallösung des Problems (QP) und ist dort sogar beliebig oft differenzierbar. Ein positiver Aspekt ist, daß bei der Transformation der Nebenbedingungen innere Punkte entstehen, sofern der zulässige Bereich des quadratischen Programms (QP) nichtleer ist. Wie die Multiplikator–Verfahren erzeugt die Modifizierte Barrieremethode in der Iteration neben einer primären Folge auch Folge von Approximationen an einen optimalen Lagrangemultiplikator: die duale Folge. Unter bestimmten Voraussetzungen konvergiert die primären Folge auch dann gegen eine Optimallösung, wenn die Barriereparameter beschränkt (oder sogar konstant) bleiben und nicht gegen Unendlich divergieren.

Auf der theoretischen Seite konnten einige neue Konvergenzresultate für die Modifizierte Barrieremethode bewiesen werden. Wir haben in dieser Arbeit gezeigt, daß die vom Verfahren gelieferte duale Folge gegen eine Optimallösung des (Lagrange–) dualen Problems (D) konvergiert, falls die Menge der Optimallösungen von (QP) nichtleer und beschränkt ist. Im Fall, daß die Zielfunktion des Problems (QP) linear ist, d.h. daß sich (QP) auf ein lineares Programm reduziert, entspricht (D) dem klassischen dualen linearen Programm. Für diesen Fall zeigen unsere Resultate, daß dann die dualen Funktionswerte mindestens linear gegen den optimalen Funktionswert streben. Wählt man die Barriereparameter als eine unbeschränkte Folge, so erhält man sogar superlineare Konvergenz. Die Modifizierte Barrieremethode kann daher auch als ein

Verfahren zur Lösung des dualen linearen Programmes angesehen werden.

Unsere Konvergenzresultate besagen, daß für strikt konvexe quadratische Zielfunktionen q, d.h. strikt positiv definites Q, die primäre Folge gegen eine Optimallösung von (QP) konvergiert. Für den Fall, daß q nicht strikt konvex ist, konnten ebenfalls einige hinreichende Kriterien für die Konvergenz gefunden werden. Dies stärkt die Ergebnisse von Polyak und ergänzt die Resultate von Powell, welche Konvergenz des Verfahrens für lineare Programme bei beschränkter zulässiger Menge und konstantem Barriereparameter zeigen. Die in dieser Arbeit bewiesenen Ergebnisse liefern unter anderem Ansatzpunkte für weitere Untersuchungen: Wir haben gezeigt, daß falls die primäre Folge konvergiert, sie gegen eine Optimallösung konvergieren muß.

Während Powell die Konvergenz für lineare Programme, d.h. $Q = 0$, beweist und unsere Ergebnisse Konvergenz für strikt positiv definites Q sichern, bleibt nach wie vor ungeklärt, ob die primäre Folge auch für beliebiges positiv semidefinites Q konvergiert, wenn man einzig und allein voraussetzt, daß die Menge der Optimallösungen von (QP) nichtleer und beschränkt ist. Hier ist sicherlich ein Punkt für weitere Untersuchungen zu sehen. Interessant wäre auch zu klären, ob die Voraussetzung, daß die Menge der Optimallösungen beschränkt ist, weiter abgeschwächt werden kann. Daß man diese Voraussetzung nicht ersatzlos streichen kann, zeigen unsere Ergebnisse über die Modifizierten Barrierefunktionen.

Da bei unserer Analyse des Verfahrens nur an sehr wenigen Stellen einging, daß die Zielfunktion von (QP) nicht nur konvex sondern darüberhinaus auch quadratisch ist, erscheint eine Verallgemeinerung der hier erarbeiteten Resultate auf allgemeine konvexe Funktionen zumindest bei linearen Nebenbedingungen ohne allzu großen Aufwand möglich.

In dieser Arbeit wurde auch die Implementierung des Verfahrens diskutiert. Die unterschiedliche Wahl der Barriereparameter führte zu verschiedenen Varianten der Modifizierten Barrieremethode. Dabei haben wir unter anderem eine Variante eingeführt, die mit einem unzulässigen Startvektor initialisiert werden kann. In einer konkreten Implementierung haben wir das Verhalten dieser Variante und der Variante mit konstantem Barriereparameter untersucht. Dabei stieg die Anzahl der Newton–Schritte, die bis zum Abbruch benötigt wurden, höchstens linear mit der Anzahl der Unbekannten des behandelten Problems. Im Rahmen dieser Arbeit konnten nur relativ kleine Probleme (mit $n \leq 128$ Unbekannten) numerisch behandelt werden. Man könnte daher hier ansetzen und untersuchen, ob für steigendes n, wobei z.B. $n \geq 5000$, das gleiche Verhalten der Algorithmen KP–MBFM und US–MBFM beobachtet werden kann. Eine Implementierung für Probleme dieser Größe müßte dann selbstverständlich spezielle Techniken, etwa bei der Speicherung von dünn besetzten Matrizen, verwenden. Von Interesse wäre im Zusammenhang mit der Anzahl der Newton–Schritte, welche die Barrieremethode bis zum Abbruch benötigt, auch die theoretische Konvergenzgeschwindigkeit des Newton–Verfahrens, wenn es, wie in unserer Implementierung, zur Minimierung der Modifizierten Barrierefunktionen eingesetzt wird.

Insgesamt besitzt die Modifizierte Barrieremethode eine Reihe interessanter Eigenschaften. Im Gegensatz zu den Verfahren, die auf den klassischen Barrierefunktionen basieren, benötigt sie keine inneren Punkte und es ist ebenfalls unnötig, die Barriereparameter gegen Unendlich divergieren zu lassen. Werden die Barriereparameter nicht zu groß gewählt, so ist das lineare Gleichungssystem zur Bestimmung des Newton–Schrittes im laufenden Iterationspunkt numerisch sehr stabil, selbst wenn sich der Lösungsprozeß der Optimallösung nähert. Auch das Vorhandensein von Approximationen an einen optimalen Lagrangemultiplikator erweist sich in der Praxis als sehr hilfreich, etwa im linearen Fall, wo das Verfahren dann auch eine Lösung des dualen linearen Programmes liefert.

Andererseits bleiben beim momentanen Stand der Theorie noch einige Fragen ungeklärt, etwa, wie die Barriereparameter am effektivsten zu wählen sind. Polyaks Ergebnisse legen nahe, daß man die Barriereparameter größer als einen bestimmten Schwellenwert k_0 wählen muß, um zufriedenstellend schnelle Konvergenz zu erhalten. Dieser Schwellenwert hängt von der Degeneriertheit des jeweiligen Problems und einigen Konstanten ab, die im wesentlichen durch das implizite Funktionetheorem geliefert werden. Wie der Schwellenwert k_0 in der Praxis zu finden ist, bleibt offen. Der von Polyak vorgeschlagene Algorithmus AP–MBFM, in dem versucht wird, die Barriereparameter anzupassen, stellte sich als ungeeignet für die verwendeten Testprobleme heraus. Die Theorie deckt ebenfalls noch nicht die Wahl von $\varepsilon_{\mathrm{newton}}$, d.h. der Genauigkeit, mit der in einem praktischen Algorithmus die Minima der Modifizierten Barrierefunktionen approximiert werden, ab. Wählt man $\varepsilon_{\mathrm{newton}}$ zu klein, so entsteht eventuell unnötiger Rechenaufwand. Wird $\varepsilon_{\mathrm{newton}}$ zu groß gewählt, so divergieren möglicherweise die Folgen, die der Algorithmus liefert, wie sich bei den Testproblemen zeigte.

Daher sollte man die Modifizierte Barrieremethode beim momentanen Stand vor allem als eine äußerst interessante Basis für weitere Entwicklungen sehen. Die Modifizierten Barrierefunktionen, so wie sie von Polyak eingeführt wurden, erscheinen als ein mächtiges Hilfsmittel zur Gewinnung von effizienten Algorithmen. Dennoch ist zumindest für lineare und quadratische Probleme noch einige Feinarbeit in Theorie und Praxis nötig, um hier mit den momentan besten Verfahren konkurrieren zu können.

Anhang A

Beschreibung des Programmprojektes

Im Programm `modbfm` wurde die Modifizierte Barrieremethode in drei Varianten, AP–MBFM, KP–MBFM und US–MBFM, die wir in Abschnitt 6.1.4 vorgestellt haben, implementiert. Dieses Kapitel beschreibt kurz das Implementierungs–Konzept und stellt das Programm `modbfm` vor.

Als Implementierungssprache wurde C++ gewählt, obwohl die „klassische" Programmiersprache für numerische Probleme FORTRAN ist. Für diese Entscheidung gab es mehrere Gründe.

C++ erfreut sich einer wachsenden Popularität. Die Attraktivität dieser Programmiersprache begründet sich nicht allein mit der großen Verbreitung und Akzeptanz ihrer Muttersprache C, die vor allem im Bereich der UNIX–Systeme dominierend ist. Die zunehmende Verbreitung von C++ erklärt sich vor allem auch durch die in C++ vorhandenen Möglichkeiten der Datenabstraktion und des objektorientierten Designs.

C++ verbindet hohe Portabilität zwischen unterschiedlichsten Plattformen mit hoher Geschwindigkeit. Nun sind Portabilität und Geschwindigkeit ebenfalls Stärken von FORTRAN. Der entscheidende Grund, C++ gegenüber FORTRAN zu bevorzugen, lag hier darin, daß C++ durch die Möglichkeiten des objektorientierten Designs und auch der Strukturierung des Codes eine deutlich übersichtlichere Programmierung ermöglicht. Damit werden viele Fehlerquellen bei der Codeerstellung durch den Programmierer von vorneherein ausgeschlossen oder zumindest eingeschränkt.

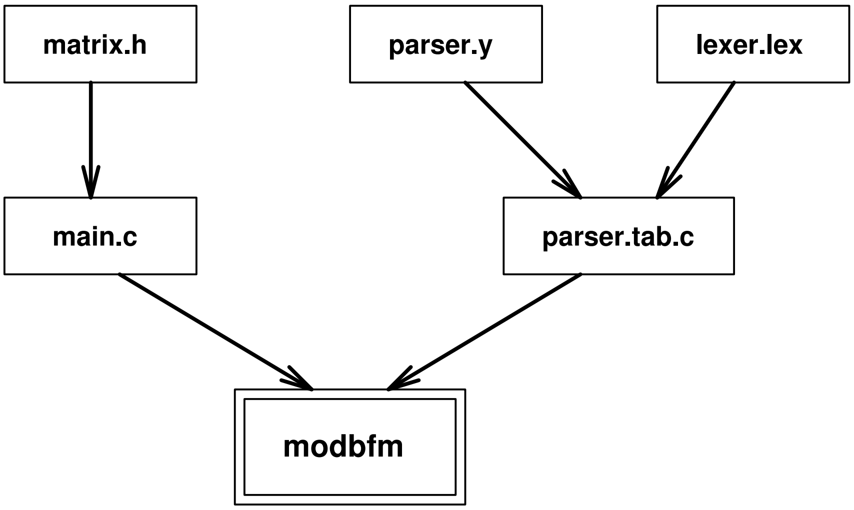

Abbildung A.1: Aufbau des Programmprojekts `modbfm`

A.1 Konzept der Implementierung

Das Programm `modbfm` setzt sich aus verschiedenen Modulen zusammen. Dies sind die Implementierung einer Matrix– und Vektorklasse (in `matrix.h`), eines Parsers zur syntaktischen und sematischen Analyse der Konfigurationsdatei (in `parser.y` und `lexer.lex`) und des eigentlichen Verfahrens (in `main.c`).

Die Abbildung A.1 zeigt die Zusammenhänge oder Abhängigkeiten zwischen den einzelnen Modulen. Es soll hier nun kurz auf einige wichtige Details eingegangen werden.

Implementierung einer Matrix– und Vektorklasse

In der Datei `matrix.h` ist eine Matrix– und Vektorklasse implementiert. Hier wurden die am Anfang dieses Kapitel erwähnten Möglichkeiten von C++ genutzt und eine Matrix–/Vektor–Arithmetik über die Redefinition (das Überladen) der Standard–Operatoren der Sprache C++ realisiert. Eine ausführliche Beschreibung der Implementierung in `matrix.h` würde den Rahmen dieser Arbeit hier sprengen. Es soll jedoch kurz die Mächtigkeit der Klassen `matrix` und `vector` angedeutet werden. Die Abbildung A.2 zeigt an einem einfachen Beispiel, wie die Verwendung der beiden Klassen eine äußerst übersichtliche Formulierung von Programmtext ermöglicht.

Der Programmtext der Datei `matrix.h` ist im Anhang B.1 abgedruckt.

```
#include "matrix.h"
main()
{
   cout << "Bitte geben Sie n und m ein :";
   cin >> n >> m;

   matrix A(m,n), B(m,m);
   vector x(n), y(m), z(m);

   cout << "Bitte geben Sie die Matrix A ein :";
   cin >> A;
   cout << "Bitte geben Sie die Matrix B ein :";
   cin >> B;
   cout << "Bitte geben Sie die Vektoren x und y ein :";
   cin >> x >> y;

   if ( x >= 0.0 )
        cout << "x ist komponentenweise nichtnegativ";

   cout << " y + B*A*x = " << y + B*A*x;
   cout << " A^T * y = " << A.T()*y;
   z = 0.5*A*x + 2.0*y;
}
```

Abbildung A.2: Beispiel für die Benutzung der Klassen `matrix` und `vector`

Der Parser zum Auswerten der Konfigurationsdatei

Ein Ziel bei der Implementierung war es, dem Benutzer zu ermöglichen, das Programm `modbfm` für seine Bedürfnisse (neu) zu konfigurieren, ohne die Quelltexte neu übersetzen zu müssen. Daher wurde die Möglichkeit geschaffen, in einer Konfigurationsdatei die wesentlichen Parameter des Verfahrens, z.B. die verwendete Variante der Modifizierten Barrieremethode oder die Newton–Genauigkeit, zu spezifizieren. Um diese Konfiguration so einfach und durchsichtig wie möglich zu machen, kann dies in der Art von Variablenzuweisungen geschehen. Es wurde ein Parser entworfen und implementiert, der die Einträge in der Konfigurationsdatei beim Programmstart von `modbfm` auswertet.

Der Standardname der Konfigurationsdatei ist `specfile.mbf`. Mit Hilfe der Kommandozeilenoption `-l` kann jedoch auch eine andere Datei ausgewählt werden (siehe dazu Tabelle A.3). Die Datei `specfile.mbf` muß nicht existieren. Fehlt sie, oder ist sie

```
# Benutze die Barrieremethode mit festem Parameter 1000
# und benutze bei der Transformation den Parameter M = 100
barriere_methode = konstant
barriere_parameter = 1000
M = 100.0

# Setze die Newton-Genaugikeit auf 1e-5 fest
eps_newton = 1e-5

# Nach jeden Teilproblem erhoehe die Newton-Genauigkeit
# um den Faktor 0.95
gamma = 0.95

# Betrachte alle Dateien als Dateien mit Problemdaten fuer
# ein lineares Programm
standard_problem = LP
```

Abbildung A.3: Beispiel einer Konfigurationsdatei

nicht lesbar, so benutzt `modbfm` Standardwerte für die Parameter.

Ein Eintrag in der Konfigurationsdatei besitzt die Form:

$$\text{PARAMETER} = \text{WERT}$$

In der Konfigurationsdatei sind darüberhinaus Kommentare zugelassen. Ein Kommentar beginnt per Definition mit einem Gitter (`#`) und endet mit dem Zeilenende.

In Tabelle A.1 sind die zulässigen Parameter mit ihrer Bedeutung sowie den zulässigen Werten aufgeführt. Dabei steht *float* für beliebige nichtnegative Fließkommawerte. Die restlichen Einträge in der Spalte „zulässige Werte“ stehen für die entsprechende Zeichenkette in beliebiger Groß–/Kleinschreibung. In Tabelle A.2 sind die Standardwerte für die Parameter angegeben, die Abbildung A.3 zeigt ein Beispiel einer Konfigurationsdatei.

Bei fehlerhaften Eintragungen in der Konfigurationsdatei wird die Zeilennummer des ersten Fehlers ausgegeben und der Programmlauf abgebrochen.

Der Parser zum Auswerten der Konfigurationsdatei wurde mit Hilfe der UNIX–Programme `bison` und `flex` erstellt, die frei verfügbare Versionen der Programme `yacc` bzw. `lex` sind. Der Quelltext des Moduls zur lexikalischen Analyse der Konfigurationsdatei befindet sich in der Datei `lexer.lex`, die Quellen zur syntaktischen und semantischen Analyse in `parser.y`. Beide Quelltexte sind im Anhang B.2 zu finden.

Parameter	zul. Werte	Bedeutung

`barriere_methode`	`variabel`
	`konstant`
	`unzulaessig`
		Wählt Algorithmus AP–MBFM.
		Wählt Algorithmus KP–MBFM.
		Wählt Algorithmus US–MBFM.

`barriere_parameter`	*float*	Start–Barriereparameter k_{start}
`barriere_inkrement`	*float*	Setzt ϱ_{Summand} (vgl. Seite 103).
`barriere_faktor`	*float*	Setzt ϱ_{Faktor} (vgl. Seite 103).
`maximale_barriere`	*float* `keine`	Bestimmt eine Obergrenze für die Barriereparameter. Erlaubt, daß die Barriereparameter beliebig groß werden können.
`eps_newton`	*float*	Setzt die Newton–Genauigkeit $\varepsilon_{\text{newton}}$.
`gamma`	*float*	Bestimmt den Parameter γ der Algorithmen AP–MBFM, KP–MBFM bzw. US–MBFM.
`term_prim_unzulaessigkeit`	*float*	Setzt $\delta_{\text{primär}}$, das Abbruchkriterium für die primäre Unzulässigkeit (vgl. hierzu Seite 100).
`term_dual_unzulaessigkeit`	*float*	Setzt δ_{dual} (vgl. Seite 100).
`term_komplementaritaet`	*float*	Bestimmt δ_{kompl} (vgl. Seite 100).
`M`	*float*	Setzt die Größe von M („big M“) bei der Transformation von QP auf $(\text{QP})^{(\text{M})}$ fest (vgl. Seite 100).

`standard_problem`	`LP`
	`QP`
	`auto`
		Alle Eingabedaten werden als Daten eines linearen Programmes betrachtet.
		analog für quadratische Programme
		Bei der Dateiendung `.lp` wird ein lineares Programm erwartet, bei der Endung `.qp` ein quadratisches Programm.

Tabelle A.1: Mögliche Parameter in der Konfigurationsdatei

Parameter	Standardwert
`barriere_methode`	`konstant`
`barriere_parameter`	100.0
`barriere_inkrement`	0.0
`barriere_faktor`	3.0
`maximale_barriere`	10^4
`eps_newton`	10^{-3}
`gamma`	0.99
`term_prim_unzulaessigkeit`	10^{-7}
`term_dual_unzulaessigkeit`	10^{-7}
`term_komplementaritaet`	10^{-7}
`M`	100.0
`standard_problem`	`auto`

Tabelle A.2: Standardwerte der Parameter

Das Hauptmodul

In der Datei `main.c` befindet sich die Realisierung des eigentlichen Algorithmus samt der notwendigen Ein–/Ausgaberoutinen. Wie bereits erwähnt, wurden drei Varianten der Modifizierten Barrieremethode implementiert:

1. Die Modifizierte Barrieremethode mit Parameteranpassung (Algorithmus AP–MBFM)

2. Die Modifizierte Barrieremethode mit konstantem Barriereparameter (Algorithmus KP–MBFM)

3. Die Modifizierte Barrieremethode mit unzulässigem Startpunkt (Algorithmus US–MBFM)

Durch die Verwendung der Matrix– und Vektorklasse konnten die Algorithmen übersichtlich und relativ kurz in Programmcode gefaßt werden. Der Quelltext der Datei `main.c` ist im Anhang B.3 abgedruckt.

A.2 Benutzung des Programms `modbfm`

Der Aufruf von `modbfm` erfolgt durch

`modbfm` [*Optionen*] `Dateiname`

von der Kommandozeile aus. Dabei steht in der letzten Zeile [*Optionen*] für optionale Kommandozeilenparameter und `Dateiname` für den Namen der Datei mit den Daten für

Option	Bedeutung
`-?`	Aufruf einer Hilfestellung zu den Kommandozeilenoptionen
`-h`	Synonym für `-?`
`-l name`	Anstelle der Konfigurationsdatei `specfile.mbf` wird die Datei `name` ausgewertet
`-q`	Alle Konfigurationsdateien werden ignoriert
`-s`	Der Startvektor wird aus der Datei mit den Problemdaten ausgelesen
`-v`	Einschalten von ausführlichen Text– und Datenausgaben während des Programmlaufs
`-x`	Unterdrücken jeglicher Text– und Datenausgabe während des Programmlaufs. Es wird nur das Endergebnis und die Statistik ausgegeben.

Tabelle A.3: Kommandozeilenoptionen für `modbfm`

das zu behandelnde Optimierungsproblem. Die möglichen Kommandozeilenoptionen und ihre Bedeutung kann man aus Tabelle A.3 entnehmen.

Das Format für eine Datei mit Problemdaten ist das folgende: Zu Beginn der Datei ist eine beliebige Anzahl von (optionalen) Kommentaren erlaubt. Wiederum beginnt ein Kommentar per Definition mit einem Gitter (`#`) und endet mit dem Zeilenende. Danach werden zwei ganze Zahlen n und m erwartet, welche die Anzahl der Variablen (n) bzw. die Anzahl der Nebenbedingungen (m) des Problems spezifizieren. Bei quadratischen Programmen ist als nächstes die Matrix Q der quadratischen Form anzugeben[1]. Dabei müssen die Elemente von Q zeilenweise angegeben werden.

Danach folgen der Vektor c, die Matrix A und der Vektor b. Letztendlich kann bei Bedarf noch ein Startvektor angegeben werden. Die einzelnen Elemente der Vektoren und Matrizen dürfen durch beliebig viele Leerzeichen, Kommata, Tabulatoren oder Zeilenumbrüche getrennt werden.

Als Beispiel betrachten wir das quadratische Optimierungsproblem

$$\begin{aligned} \min 2x_1^2 + x_2^2 + x_1x_2 - 4.2x_1 + 6.2x_2 \\ x_1 \geq \quad 2.5 \\ x_2 \geq -5.3 \\ x_2 \leq \quad 6 \end{aligned}$$

Die Abbildung A.4 zeigt, wie eine mögliche Datei `beisp.qp` mit den Problemdaten für dieses quadratische Programm aussehen könnte. Der Übersichtlichkeit halber haben wir die Daten in der Beispielsdatei so angeordnet, daß die Struktur des Problems ersichtlich ist.

[1]Bei linearen Programmen entfällt die Angabe der Matrix Q

```
#
# Datei beisp.qp mit Problemdaten fuer das Beispiel
#
 2 3

 4.0, 1.0
 1 2.0

 -4.2 , 6.2

 1.0 0.0
 0.0 1.0
 0.0 -1.0

 -2.5 -5.3 -6
```

Abbildung A.4: Beispiel für eine Datei mit Problemdaten

A.3 Der Problemgenerator `gen`

In diesem Abschnitt soll kurz auf den Problemgenerator `gen` hingewiesen werden, der benutzt wurde, um die verwendeten Testprobleme aus Abschnitt 6.2 zu erzeugen. Der Aufruf des Generators erfolgt aus der Kommandozeile durch

`gen -t` *Problemtyp Daten1 Daten2 ...*

Im folgenden sind die einzelnen Möglichkeiten für den Programmaufruf aufgelistet. Dabei stehen n, m und N für natürliche Zahlen, ε, $\mu_{\min}$ und $\mu_{\max}$ für Fließkommawerte.

- `gen -t lp-minty` ε n

  Dieser Aufruf erzeugt die Datei `minty[n].lp`, welche die Daten für das Klee–Minty Problem mit n Variablen zum Wert ε (siehe Abschnitt 6.2.1 auf Seite 110) enthält.

- `gen -t lp-zuord` n

  Bei diesem Aufruf erzeugt `gen` die Datei `zuord[n].lp` mit Daten eines zufälligen Zuordnungs–Problems (siehe Abschnitt 6.2.2 auf Seite 115). Das Zuordnungs–Problem besitzt n^2 Variablen.

- `gen -t lp-zufall` n N

Es wird die Datei `zufall[n]_[N].lp` generiert. Diese enthält die Daten eines zufälligen linearen Programms mit n Variablen und Zufallszahlen in der Matrix der Nebenbedingungen aus $[1, N]$ (siehe Abschnitt 6.2.3 auf Seite 120).

- `gen -t qp-square` $n \quad N$

  Durch dieses Kommando wird die Datei `square[n]_[N].qp` produziert. Diese enthält die Problemdaten eines zufälligen Least–Squares–Problems mit n Variablen und Zufallszahlen in der Nebenbedingungsmatrix aus dem Intervall $[1, 1000]$ (siehe Abschnitt 6.2.4 auf Seite 123).

- `gen -t qp-zufall` $n \quad N \quad \mu_{\min} \quad \mu_{\max}$

  Mit diesem Aufruf können zufällige quadratische Programme, wie sie in Abschnitt 6.2.5 auf Seite 127 beschrieben sind, erzeugt werden. Der Dateiname ist `zufall[n]_[N].qp`.

Anhang B

Quelltexte

B.1 Die Matrix– und Vektorklasse: `matrix.h`

```
/*
 *******************************************************************
 *******************************************************************
 **                                                               **
 **    Diplomarbeit von   Sven Oliver Krumke,                     **
 **                       geb. am 22.02.68                        **
 **                       in Frankfurt am Main                    **
 **    Betreuer         : Prof. Dr. J. Stoer                      **
 **                                                               **
 **    Datei            : matrix.h                                **
 **                                                               **
 **    erstellt am      : 24.11.93                                **
 **    letzte Aenderung : 19.01.94                                **
 **                                                               **
 **    Aufgabe          : Enthaelt die Implementation der Klassen  **
 **                       - matrix                                **
 **                       - vector                                **
 **    Es wird eine komplette Matrix-/Vektorarithmetik zur        **
 **    Verfuegung gestellt, die bequem mit den neudefinierten     **
 **    (ueberladenen) Operatoren *, -, + etc. in einem Programmtext **
 **    benutzt werden kann.                                       **
 *******************************************************************
 *******************************************************************
*/

#ifndef __matrix__h
#define __matrix__h

#include <iostream.h>
#include <iomanip.h>
```

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define TP double

#define NO_INIT
#define NO_BOUND_CHECK

#ifndef EQUAL
#define EQUAL(a,b)         (a == b)
#endif

#ifndef NOT_EQUAL
#define NOT_EQUAL(a,b)     (a != b)
#endif

#ifndef LESS
#define LESS(a,b)          (a <  b)
#endif

#ifndef LESS_OR_EQUAL
#define LESS_OR_EQUAL(a,b) (a <= b)
#endif

#ifndef GREATER
#define GREATER(a,b)       ( a > b)
#endif

#ifndef GREATER_OR_EQUAL
#define GREATER_OR_EQUAL(a,b) (a >= b)
#endif


class matrix;
class vector;

vector LT_Solve(matrix&, vector&);
vector UT_Solve(matrix&, vector&);
istream& ws_komma (istream& );
istream& no_eof   (istream& );

/*
 ********************************************************************
 *  Klasse:  matrix                                                 *
 *                                                                  *
 *           Die Klasse matrix stellt den Grundstein fuer das ge-   *
```

```
 *          samte Modul dar. Sie verwendet fuer die Eintraege in   *
 *          den Matrizen den Typ TP, der mittels #define zu defi-  *
 *          nieren ist, z.B.                                       *
 *             #define TP double                                   *
 *          Die obige Definition legt fest, dass die Eintraeg in   *
 *          den Matrizen vom Typ double sind.                      *
 *          Weitere Optionen des Moduls sind ueber die Macros      *
 *          NO_INIT und NO_BOUND_CHECK verfuegbar. Ist NO_INIT     *
 *          durch                                                  *
 *              #define NO_INIT                                    *
 *          definiert, werden die Eintraege von neu erzeugten Ma-  *
 *          trizen nicht automatisch mit 0 initialisiert.          *
 *          Ist NO_BOUND_CHECK durch                               *
 *              #define NO_BOUND_CHECK                             *
 *          definiert, so wird bei Indizierungen nicht geprueft,   *
 *          ob die Indizes im erlaubten Bereich liegen.            *
 *                                                                 *
 *  Attribut  | Typ  | Beschreibung                                *
 *  ------------------------------------------------------------   *
 *            |      |                                             *
 *  m         | int  | Zeilenanzahl der Matrix                     *
 *  n         | int  | Spaltenanzahl der Matrix                    *
 *  a         | *TP  | Zeiger auf das  Array, welches die Eintraege *
 *            |      | der Matrix enthaelt                         *
 *                                                                 *
 *  Alle Attribute sind protected.                                 *
 *******************************************************************
 */

class matrix
{
 protected:

  int m,n;
  TP *a;

  void SizeCheck (matrix&);
  void MulCheck (matrix&);
  void RangeCheck (int,int);
  void DimCheck(int,int);
  TP abs(TP x) { return ((x < 0) ? -x : x); };

 public:

  matrix (int,int);
  matrix (const matrix&);
  virtual char* isa();
  matrix ident_mat();
```

```
  TP& operator() (int, int);
  inline int GetRows();
  inline int GetCols();
  matrix operator = (matrix&);
  matrix operator = (TP);
  matrix operator * (matrix&);
  matrix operator *= (matrix&);
  matrix operator * (TP);
  matrix operator *= (TP);
  friend matrix operator * (TP,matrix&);
  matrix operator + (matrix&);
  matrix operator += (matrix&);
  matrix operator + (TP);
  matrix operator += (TP);
  friend matrix operator + (TP, matrix&);
  matrix operator - ();
  matrix operator - (matrix&);
  matrix operator -= (matrix&);
  matrix operator - (TP);
  friend matrix operator - (TP, matrix&);
  matrix operator -= (TP);
  matrix T();
  matrix exp(int e);
  matrix Cholesky();
  virtual ~matrix();

  friend ostream& operator << (ostream&, matrix&);
  friend istream& operator >> (istream&, matrix&);
};


/*
 *******************************************************************
 *  Methode : SizeCheck (matrix& M), PROTECTED                     *
 *                                                                 *
 *  Aufgabe : Wird von den Operatoren +, - und der Funktion sp(,)  *
 *            aufgerufen und testet, ob das Objekt M eine passende *
 *            Dimension fuer die Operation besitzt.                *
 *            Falls dies nicht der Fall ist, wird das Programm mit *
 *            einer entsprechenden Fehlermeldung abgebrochen,      *
 *            ansonsten wird es ohne Meldung fortgefuehrt.         *
 *                                                                 *
 *  Ergebnis: keines                                               *
 *******************************************************************
 */

void matrix::SizeCheck(matrix& M)
{
```

```
  if ((m==M.m) && (n==M.n))    // Wenn die Dimensionen identisch sind,
    return;                    // dann normaler Programmablauf

  /* Falls die Dimensionen nicht identisch, d.h. nicht kompatibel
     fuer +, - oder die Zuweisung sind, gebe eine Fehlermeldung mit
     den verwendeten Objekttypen und ihren Dimensionen aus
  */

  cerr << isa();
  cerr << ":  inkompatible Dimensionen bei +, - oder Zuweisung" << endl;

  cerr << "1. Argument: " << m   << " x " << n   << " " << isa() << endl;
  cerr << "2. Argument: " << M.m << " x " << M.n << " " << M.isa();
  cerr << endl;

  /* Programmabbruch */

  exit (-1);
}

/*
 ******************************************************************
 *  Methode : MulCheck (matrix& M), PROTECTED                     *
 *                                                                *
 *  Aufgabe : Wird vom Operator *, d.h. der Matrixmultiplikation   *
 *            aufgerufen und testet, ob das Objekt M eine passende *
 *            Dimension fuer die Operation besitzt.               *
 *            Wie bei SizeCheck wird das Programm mit einer Fehler- *
 *            meldung abgebrochen, falls versucht wird, 2 Matrizen *
 *            miteinander zu multiplizieren, deren Dimensionen in-  *
 *            kompatibel sind.                                    *
 *                                                                *
 *  Ergebnis: keines                                              *
 ******************************************************************
 */

void matrix::MulCheck(matrix& M)
{
  if (n == M.m)            // Wenn die Dimensionen passend sind,
    return;                // fahre mit dem Programmablauf normal fort

  /* Falls die Dimensionen nicht passend sind, dann gebe eine
     Fehlermeldung mit Informationen fuer den Benutzer aus
  */

  cerr << "matrix: Inkompatible Dimensionen fuer die Multiplikation" << endl;
  cerr << "1. Operand: " << m   << " x " << n  << " " << isa() << endl;
  cerr << "2. Operand: " << M.m << " x " << M.n<< " " << isa() << endl;
```

```
  /* Programmabbruch */

  exit (-1);
}

/*
 *******************************************************************
 *  Methode : RangeCheck (matrix& M), PROTECTED                    *
 *                                                                 *
 *  Aufgabe : Wird von den Indexoperatoren (,) bzw. (), z.B.       *
 *            bei A(1,1) oder b(k) aufgerufen und testet zur       *
 *            Laufzeit des Programmes, ob der verwendete Index     *
 *            innerhalb der zulaessigen Grenzen fuer das Objekt    *
 *            liegt.                                               *
 *            Falls nicht der Fall ist, wird das Programm mit      *
 *            einer entsprechenden Fehlermeldung abgebrochen,      *
 *            ansonsten wird es ohne Meldung fortgefuehrt.         *
 *                                                                 *
 *  Ergebnis: keines                                               *
 *******************************************************************
 */

void matrix::RangeCheck (int i, int j)
{
  if (i<=0 || i>m)     // Wenn der Spalten-Index negativ oder zu gross ist
                       // dann breche mit einer Fehlermeldung ab
    {
      cerr << isa() << ": unzulaessiger Spalten-Index" << endl;
      cerr << "Objekt: " << m << " x " << n << " " << isa() << endl;
      cerr << "Angesprochenes Element: ( " << i << "," << j << ")";
      cerr << endl;

      exit (-1);
    }
  if (j<=0 || j>n)     // Hier erfolgt analog der Test auf die Gueltigkeit
                       // des Reihen-Index
    {
      cerr << isa() << ": falscher Reihen-Index" << endl;
      cerr << "Objekt: " << m << " x " << n << " " << isa() << endl;
      cerr << "Angesprochenes Element: ( " << i << "," << j << ")";
      cerr << endl;

      exit (-1);
    }
}

/*
```

```
 ****************************************************************
 *  Methode : DimCheck(), PROTECTED                             *
 *                                                              *
 *  Aufgabe : Wird von den Konstruktoren aufgerufen und prueft, *
 *            ob die Dimensionen der zu erstellenden Matrix positiv *
 *            sind.                                             *
 *            Falls negative Werte fuer die Zeilen- oder Spalten- *
 *            anzahl angegeben wurden, wird das Programm mit einer *
 *            entsprechenden Fehlermeldung abgebrochen.         *
 *                                                              *
 *  Ergebnis: keines                                            *
 ****************************************************************
 */

void matrix::DimCheck (int i, int j)
{
  if (i<1 || j<1)      // Falls negative Dimensionen angegeben wurden,
                       // breche mit einer Fehlermeldung ab.
    {
      cerr << isa()<< ": falsche Dimensionen fuer den Konstruktor" << endl;
      exit (-1);
    }
}

/*
 ****************************************************************
 *  Methode : matrix (int um, int un), KONSTRUKTOR              *
 *                                                              *
 *  Aufgabe : Wird automatisch aufgerufen, wenn z.B. durch      *
 *             matrix A(1,3)  oder                              *
 *             matrix B(i,j)                                    *
 *            eine neue Matrix erzeugt werden soll. Der erste   *
 *            Parameter, um, bestimmt die Zeilenanzahl, waehrend *
 *            durch den zweiten Parameter, un, die Spaltenanzahl *
 *            festgelegt wird. Der Konstruktor allokiert Speicher- *
 *            platz fuer die Matrixelemente. Ist aus irgendwelchen *
 *            Gruenden kein Speicher mehr verfuegbar, so wird das *
 *            Programm mit einer entsprechenden Fehlermeldung   *
 *            abgebrochen.                                      *
 *            Im Erfolgsfall zeigt dann das Attribut a auf das  *
 *            Array der Eintraege, die Attribute m und n werden mit *
 *            um, bzw. un initialisiert.                        *
 *            Die Eintraege der Matrix werden ferner mit Null   *
 *            initialisiert. War beim Uebersetzen des Moduls    *
 *            das Macro NO_INIT durch #def NO_INIT definiert, so *
 *            unterbleibt diese letzte Initialisierung. Dies bringt *
 *            im Programm eine geringe Geschwindigkeitssteigerung. *
 *                                                              *
```

```
 *  Ergebnis: keines                                              *
 *                                                                *
 *  Beispiel:  matrix A(1,3);                                     *
 *             erzeugt eine 1x3 Matrix mit Variablennamen A       *
 *             matrix B(i,j);                                     *
 *             erzeugt in der Variablen B eine ixj Matrix         *
 ******************************************************************
 */

matrix::matrix(int um,int un)
{
  DimCheck(um,un);   // Teste die verlangten Dimensionen auf ihre
                     // Gueltigkeit

  if ((a=new TP[um*un])!=NULL)  // Wenn Speicher korrekt allokiert werden
                                // konnte, dann fahre fort
    {
      m = um;                   // Zeilenanzahl abspeichern
      n = un;                   // Spaltenanzahl abspeichern

      /* Der folgende Block, der durch #ifndef ... #endif umschlossen
         ist, wird nur dann mituebersetzt, wenn das Macro NO_INIT
         nicht definiert war.
         */

#ifndef NO_INIT

      for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
          (*this)(i,j) = 0.0;
#endif
    }

     /* FALLS kein Speicher allokiert werden konnte, wird das
        Programm abgebrochen.
        */
  else {
    cerr << isa() << ": kein Speicher mehr verfuegbar." << endl;
    exit (-1);
  }
}

/*
 ******************************************************************
 *  Methode : matrix (const matrix& M), KONSTRUKTOR               *
 *                                                                *
 *  Aufgabe : Wird automatisch aufgerufen, wenn z.B.  durch       *
 *             matrix A(B)                                        *
```

```
 *           eine neue Matrix A erzeugt werden soll. Dabei dient   *
 *           die andere Matrix (hier im Beispiel B) als Muster.    *
 *           Die neue Matrix wird exakt nach dem Abbild der an-    *
 *           deren Matrix geschaffen. Sie erhaelt die selben Di-   *
 *           mensionen und ihre Eintraege werden mit den Werten    *
 *           der Muster-Matrix initialisiert.                      *
 *                                                                 *
 *  Ergebnis: keines                                               *
 *                                                                 *
 *  Beispiel:  matrix A(B);                                        *
 *             erzeugt die Matrix A und kopiert alle Werte von B   *
 *             nach A, d.h. A und B sind nun identisch.            *
 *******************************************************************
 */

matrix::matrix(const matrix& M)
{
  if ((a=new TP[M.m*M.n])!=NULL) // Wenn Speicherplatz verfuegbar ist ...
    {
      m = M.m;                   // Zeilenanzahl abspeichern
      n = M.n;                   // Spaltenanzahl abspeichern

      /* In der folgenden Schleife werden die Eintraege aus M
         in die neue Matrix kopiert
         */

      for (int i=0; i<m*n; i++)
        a[i] = M.a[i];
    }

  // Bei fehlerhafter Allockierung wird das Programm abgebrochen

  else{
    cerr << isa() << ": kein Speicher mehr verfuegbar" << endl;
    exit (-1);
  }
}

/*
 *******************************************************************
 *  Methode : isa(), VIRTUAL                                       *
 *                                                                 *
 *  Aufgabe : Wird von Fehlerroutinen aufgerufen und liefert den   *
 *           Typ des Objektes als Zeichenkette zurueck.            *
 *           Dies ist per Voreinstellung "Matrix", kann aber durch *
 *           Ueberschreiben der Methode isa() in abgeleiteten      *
 *           Klassen geaendert werden. So wird z.B. in der Vektor- *
 *           klasse die Funktion ueberschrieben und liefert dann   *
```

```
 *           fuer Vektoren die Zeichenkette "Vektor" zurueck, auch *
 *           wenn sie aus Methoden der Vaterklasse aufgerufen wird.*
 *                                                                 *
 *  Ergebnis: Zeiger auf Zeichenkette mit Objekttyp, "Matrix"      *
 *******************************************************************
 */

char* matrix::isa()
{
  return "Matrix";
}

/*
 *******************************************************************
 *  Methode : matrix ident_mat()                                   *
 *                                                                 *
 *  Aufgabe : Wird diese Methode fuer eine Matrix aufgerufen, so   *
 *           werden die Matrixelemente wie folgt geaendert:        *
 *                                                                 *
 *             (*this)(i,j) = 1.0, falls i = j                     *
 *             (*this)(i,j) = 0.0, sonst                           *
 *                                                                 *
 *           bei quadratischen Matrizen resultiert dies in einer   *
 *           Einheitsmatrix.                                       *
 *                                                                 *
 *  Ergebnis: Die soeben neu gebildete Matrix                      *
 *                                                                 *
 *  Beispiel:  matrix A(4,4);                                      *
 *                                                                 *
 *            A.ident_mat();   // Erzeuge in A eine 4x4 Einheits-  *
 *                             // matrix.                          *
 *******************************************************************
 */

matrix matrix::ident_mat()
{
  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
      if ( i==j )
        (*this)(i,j) = 1.0;
      else
        (*this)(i,j) = 0.0;

  return (*this);
}

/*
 *******************************************************************
```

```
 *  Methode : TP &matrix operator() (int i, intj)               *
 *                                                              *
 *  Aufgabe : Liefert eine Referenz auf das Element in der i-ten    *
 *            Zeile und j-ten Spalte der Matrix.                *
 *            Ist das Makro NO_BOUND_CHECK undefiniert, so wird     *
 *            auf die Gueltigkeit der Indizes geprueft.         *
 *                                                              *
 *  Ergebis : vom Typ TP, Referenz auf das Element (i,j)        *
 ****************************************************************
 */


TP& matrix::operator() (int i,int j)
{

#ifndef NO_BOUND_CHECK
  RangeCheck(i,j);
#endif

  return a[(i-1)*n+(j-1)];
}


/*
 ****************************************************************
 *  Methode : int GetRows (), INLINE                            *
 *                                                              *
 *  Aufgabe : Diese Methode erlaubt es, von aussen die Anzahl der   *
 *            Zeilen einer Matrix zu erfahren. Dies haette auch     *
 *            dadurch geschehen koennen, dass man das Attribut m    *
 *            nach aussen zugaenglich macht. Dies waere aber mit    *
 *            dem Nachteil verbunden, dass man dann auch von aussen *
 *            m veraendern koennte, was aus Kapselungsgruenden      *
 *            nicht als sicher erscheint.                       *
 *                                                              *
 *  Ergebis : vom Typ int, Anzahl der Zeilen der Matrix         *
 *                                                              *
 *  Beispiel: matrix A(4,5);                                    *
 *            int p;                                            *
 *                                                              *
 *            p = A.GetRows();        // p = 4;                 *
 ****************************************************************
 */

inline int matrix::GetRows()
{
  return m;
}
```

```
/*
 ******************************************************************
 *  Methode : int GetCols (), INLINE                              *
 *                                                                *
 *  Aufgabe : Wie GetRows() (s.o.), nur liefert diese Methode die  *
 *            Spalatenanzahl der Matrix.                          *
 *                                                                *
 *  Ergebis : vom Typ int, Anzahl der Spalten der Matrix          *
 *                                                                *
 *  Beispiel: matrix A(4,5);                                      *
 *            int p;                                              *
 *                                                                *
 *            p = A.GetCols();        // p = 5;                   *
 ******************************************************************
 */

inline int matrix::GetCols()
{
  return n;
}

/*
 ******************************************************************
 *  Methode : matrix operator = (const matrix& M)                 *
 *                                                                *
 *  Aufgabe : Der Operator = implementionert die Zuweisung zwischen *
 *            zwei Objekten des Typs matrix.                      *
 *            Es wird getestet, ob die Dimensionen des Objektes und *
 *            der Matrix M identisch sind, d.h. ob eine Zuweisung  *
 *            ueberhaupt moeglich ist. Falls dies nicht der Fall   *
 *            ist, wird eine entsprechende Fehlermeldung ausgegeben *
 *            und das laufende Programm abgebrochen.              *
 *                                                                *
 *  Ergebnis: vom Typ matrix, Wert der Zuweisung                  *
 *                                                                *
 *  Beispiel:  matrix A(2,3), B(2,3), C(2,3), R(3,3);             *
 *                                                                *
 *             A = B;        // In Ordnung: A := B                *
 *                                                                *
 *             C = B = A;    // In Ordnung: B := A; C:=B          *
 *                                                                *
 *             D = A;        // Fehler: Abbruch mit Fehlermeldung  *
 *                              wegen inkompatibler Dimensionen    *
 ******************************************************************
 */
```

```
matrix matrix::operator = (matrix& M)
{
  SizeCheck (M);    // Teste die Dimensionen

  // Die folgende Schleife kopiert die Eintraege aus M in die
  // Matrix *this

  for (int i=0; i < n*m; i++)
    a[i] = M.a[i];

  return *this;     // Rueckgabewert, damit Zuweisungen der Art
                    // A = B = C moeglich sind
}

/*
 *******************************************************************
 *  Methode : matrix operator = (TP x)                             *
 *                                                                 *
 *  Aufgabe : Ueberschreiben der Eintraege in einer Matrix mit     *
 *            einem bestimmten Wert.                               *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Zuweisung               *
 *                                                                 *
 *  Beispiel:  matrix A(2,3);                                      *
 *             double t;       // Der Elementtyp TP sei fuer dieses *
 *                                Beispiel double                  *
 *                                                                 *
 *             t = 5.0;                                            *
 *             A = t;          // Ueberschreibe alle Werte in A mit *
 *                             // t, d.h. mit 5                    *
 *             A = 0;          // A = Nullmatrix                   *
 *******************************************************************
 */

matrix matrix::operator = (TP x)
{
  for (int i = 0; i < n*m; i++)
    a[i] = x;

  return *this;   // Rueckgabewert fuer Zuweisungen der Art:
                  // A = B = 0.5;
}


/*
 *******************************************************************
 *  Methode : matrix operator * (matrix& M)                        *
 *                                                                 *
```

```
 *  Aufgabe : Der Operator * implementionert die Multiplikation    *
 *            zwischen zwei Matrizen.                               *
 *            Es wird getestet, ob die Dimensionen der zwei         *
 *            Matrizen so beschaffen sind, dass eine Multiplikation *
 *            moeglich ist. Falls dies nicht so ist, bricht das     *
 *            Programm mit einer Fehlermeldung ab.                  *
 *                                                                  *
 *  Ergebnis: vom Typ matrix, Ergebnis der Multiplikation           *
 *                                                                  *
 *  Beispiel:  matrix A(2,3), B(3,2), C(5,5), R(3,3);               *
 *                                                                  *
 *             R = A*B;          // In Ordnung                      *
 *             R = A*C;          // Fehler: Abbruch mit Meldung,    *
 *                               // dass die Dimensionen der beiden *
 *                               // zu multiplizierenden Matrizen   *
 *                               // nicht kompatibel sind.          *
 ********************************************************************
 */

matrix matrix::operator * (matrix& M)
{
  MulCheck(M);       // Teste, ob die Dimensionen der Matrix und der
                     // Matrix M miteinander vertraeglich fuer die
                     // Multiplikation sind.

  matrix C(m,M.n);   // Erzeuge die Ergebnismatrix

  /* Die folgende Schleife fuehrt die Matrixmultiplikation
     durch
     */

  for (int i=1; i<=C.m; i++)
    for (int j=1; j<=C.n; j++)
      {
        C(i,j) = (*this)(i,1) * M(1,j);
        for (int k=2; k<=M.m; k++)
          C(i,j) += (*this)(i,k) * M(k,j);
      }

  return C;           // Rueckgabe des Ergebnisses
}

/*
 ********************************************************************
 *  Methode : matrix operator *= (matrix& M)                        *
 *                                                                  *
 *  Aufgabe : Da in der Sprache C Kurzformen fuer gleichzeitige     *
 *            Multiplikation und Zuweisung erlaubt sind, wird diese *
```

```
 *           Art der Notation durch diesen Operator ebenfalls er-  *
 *           moeglicht. Die Matrix wird mit M multipliziert,       *
 *           sofern die Dimensionen dies zulassen, und an-         *
 *           schliessend mit dem Ergebnis ueberschrieben. Dies     *
 *           ist jedoch nur dann moeglich, wenn beide Matrizen     *
 *           quadratisch sind, wie man sich leicht aus den         *
 *           Dimensionen bei der Matrixmultiplikation ueberlegt.   *
 *           Alle moeglichen Fehler, d.h. unpassende Dimensionen,  *
 *           werden abgefangen.                                    *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Multiplikation          *
 *                                                                 *
 *  Beispiel:  matrix A(3,3), B(3,3), C(3,2), D(5,5);              *
 *                                                                 *
 *            A *= B;          // In Ordnung: A = A*B              *
 *                                                                 *
 *            C *= B;          // Fehler: C ist nicht quadratisch  *
 *            A *= D;          // Fehler: inkompatible Dimensionen *
 *                             // fuer die Multiplikation          *
 *******************************************************************
 */

matrix matrix::operator *= (matrix& M)
{
  if ( (n != m) || (M.n != M.m) ) // Falls eine der Matrizen nicht
                                  // quadratisch ist ...
    {
      cerr << isa() << ": A *= B ist nur fuer quadratische A, B moeglich";
      cerr << endl;
      cerr << "Benutzte Objekte: " << m << " x " << n << " " << isa() << endl;
      cerr << "                  " << M.m << " x " << M.n << " " << M.isa();
      cerr << endl;
      exit (-1);
    }

  // Fuehre die Multiplikation durch (sofern moeglich)

  (*this) = (*this) * M;

  return (*this);     // Rueckgabe des Ergebnisses
}


/*
 *******************************************************************
 *  Methode : matrix operator * (TP x)                             *
 *                                                                 *
 *  Aufgabe : Dieser Operator implementiert die Multiplikation     *
```

```
 *           einer Matrix mit einem Skalar.                       *
 *                                                                *
 *  Ergebnis: vom Typ matrix, Ergebnis der Multiplikation         *
 *                                                                *
 *  Beispiel:  matrix A(3,5), B(3,5);                             *
 *                                                                *
 *             B = A * 0.5;                                       *
 ******************************************************************
 */

matrix matrix::operator * (TP x)
{
  matrix C((*this));        // Ergebnismatrix wird mit dem aktuellen
                            // Objekt initialisiert

  /* In der folgenden Schleife wird jedes Element in C mit x
     multipliziert
     */

  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
      C(i,j) *= x;

  return C;                // Rueckgabe des Ergebniswertes
}

/*
 ******************************************************************
 *  Methode : matrix operator *= (TP x)                           *
 *                                                                *
 *  Aufgabe : Dieser Operator ist wieder die Kurzform fuer die    *
 *            gleichzeitige Multiplikation und Zuweisung mit einem *
 *            Skalar (vgl. Matrixmultiplikation).                 *
 *                                                                *
 *  Ergebnis: vom Typ matrix, Ergebnis der Multiplikation         *
 *                                                                *
 *  Beispiel:  matrix A(3,5);                                     *
 *                                                                *
 *             A *= 0.5;       // A := 0.5 * A                    *
 ******************************************************************
 */

matrix matrix::operator *= (TP l)
{
  (*this) = (*this) * (l);
  return (*this);
}
```

```
/*
 *****************************************************************
 *  Funktion: matrix operator * (TP x, matrix& M)                *
 *                                                               *
 *  Aufgabe : Dieser Operator wird benoetigt, falls bei einer Mul-  *
 *            tiplikation zwischen einer Matrix und einem Skalar    *
 *            das Skalar zuerst steht. Er tut nichts Anderes, als   *
 *            den Operator * (matrix&, TP) unter Vertauschung der   *
 *            Reihenfolge aufzurufen.                             *
 *                                                               *
 *  Ergebnis: vom Typ matrix, Ergebnis der Multiplikation         *
 *                                                               *
 *  Beispiel:  matrix A(3,5), B(3,5);                            *
 *                                                               *
 *             B = 0.5 * A;                                      *
 *****************************************************************
 */

matrix operator * (TP x,matrix& M)
{
  return M.operator * (x);
}

/*
 *****************************************************************
 *  Methode : matrix operator + (matrix& M)                      *
 *                                                               *
 *  Aufgabe : Hier wird die Addition von zwei Matrizen durch den  *
 *            Operator + implementiert.                          *
 *            Es wird getestet, ob die Dimensionen der zwei      *
 *            Matrizen identisch sind, d.h. ob die Addition aus-  *
 *            gefuehrt werden kann. Falls dies nicht moeglich ist, *
 *            bricht das Programm mit einer Fehlermeldung ab.    *
 *                                                               *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition.             *
 *                                                               *
 *  Beispiel:  matrix A(2,3), B(2,3), C(2,3), R(3,3);            *
 *                                                               *
 *             C = A+B;          // In Ordnung                   *
 *             C = A+R;          // Fehler: Abbruch mit Meldung,  *
 *                               // dass die Dimensionen der beiden *
 *                               // zu addierenden Matrizen nicht *
 *                               // kompatibel sind.             *
 *             R = A+B;          // Fehler: Abbruch mit Meldung,  *
 *                               // dass die Zuweisung nicht durch- *
 *                               // gefuehrt werden kann (R hat   *
 *                               // falsche Dimensionen)          *
 *****************************************************************
```

```
 */

matrix matrix::operator + (matrix& M)
{
  SizeCheck (M);  // Teste auf Vertraeglichkeit der Dimensionen
                  // beider Argumente

  matrix C(M);    // Die Ergebnismatrix wird mit dem einen Summanden
                  // initialisiert

  // In der folgenden Schleife wird elementweise addiert

  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
      C(i,j) += (*this)(i,j);

  return C;       // Rueckgabe des Ergebnisses
}

/*
 *******************************************************************
 *  Methode : matrix operator += (matrix& M)                       *
 *                                                                 *
 *  Aufgabe : Wie bei der Multiplikation von zwei Matrizen soll    *
 *            auch fuer die Addition zweier Matrizen die in C bzw. *
 *            C++ fuer die Standardtypen moegliche gleichzeitige   *
 *            Addition und Zuweisung zur Verfuegung gestellt werden.*
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition                *
 *                                                                 *
 *  Beispiel:  matrix A(3,3), B(3,3), C(3,2), D(5,5);              *
 *                                                                 *
 *             A += B;          // A := A + B;                     *
 *******************************************************************
 */

matrix matrix::operator += (matrix& M)
{
  return (*this) = (*this) + M;
}

/*
 *******************************************************************
 *                                                                 *
 *  Methode : matrix operator + (TP x)                             *
 *                                                                 *
 *  Aufgabe : Dieser Operator implementiert die Addition zwischen  *
 *            einem Skalar und einer Matrix. Dabei ist diese       *
```

```
 *           Operation so definiert, dass zu jedem Element der    *
 *           Matrix das Skalar addiert wird.                      *
 *                                                                *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition               *
 *                                                                *
 *  Beispiel:  matrix A(3,5), B(3,5);                             *
 *                                                                *
 *            B = A + 1.7; // B(i,j):= A(i,j) + 1.7 fuer alle i,j  *
 ******************************************************************
 */

matrix matrix::operator + (TP x)
{
  matrix C((*this));     // Ergebnismatrix; sie wird mit dem aktuellen
                         // Matrix-Objekt initialisiert

  // Die folgende Schleife addiert zu jedem Eintrag in der Ergebnismatrix
  // C das Skalar x

  for(int i = 1; i <= m; i++)
    for(int j = 1; j <= n; j++)
      C(i,j) += x;

  return C;             // Rueckgabe des Ergebnisses
}

/*
 ******************************************************************
 *  Methode : matrix operator += (TP x)                           *
 *                                                                *
 *  Aufgabe : Dieser Operator ist wieder die Kurzform fuer die    *
 *           gleichzeitige Addition und Zuweisung.                *
 *                                                                *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition               *
 *                                                                *
 *  Beispiel:  matrix A(3,5);                                     *
 *                                                                *
 *            A += 3.8;       // A(i,j):= A(i,j)+3.8 fuer alle i,j *
 ******************************************************************
 */

matrix matrix::operator += (TP x)
{
  return (*this) = (*this) + x;
}

/*
 ******************************************************************
```

```
 *  Funktion: matrix operator + (TP x, matrix& M)                   *
 *                                                                  *
 *  Aufgabe : Falls bei einer Addition zwischen einer Matrix und    *
 *            einem Skalar das Skalar zuerst steht, benoetigen wir  *
 *            wieder eine Implementation fuer die vertauschte       *
 *            Reihenfolge.                                          *
 *                                                                  *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition                 *
 *                                                                  *
 *  Beispiel:  matrix A(3,5), B(3,5);                               *
 *                                                                  *
 *             B = 2.3 + A;  // B(i,j):= A(i,j) + 2.3 fuer alle i,j *
 ********************************************************************
 */

matrix operator + (TP x,matrix& M)
{
  return M.operator + (x);
}

/*
 ********************************************************************
 *  Methode : matrix operator - (matrix& M)                         *
 *                                                                  *
 *  Aufgabe : Dieser Operator implementiert die Differenzbildung    *
 *            zwischen zwei Martrizen und funktioniert analog zur   *
 *            Addition.                                             *
 *            Es wird getestet, ob die Dimensionen der zwei         *
 *            Matrizen identisch sind, d.h. ob die Subtraktion      *
 *            ueberhaupt moeglich ist. Falls nicht, bricht das      *
 *            das Programm mit einer Fehlermeldung ab.              *
 *                                                                  *
 *  Ergebnis: vom Typ matrix, Ergebnis der Subtraktion              *
 *                                                                  *
 *  Beispiel:  matrix A(2,3), B(2,3), C(2,3), R(3,3);               *
 *                                                                  *
 *             C = A-B;          // In Ordnung, C = A-B             *
 *             C = A-R;          // Fehler: Abbruch mit Meldung,    *
 *                               // dass die Dimensionen der beiden *
 *                               // zu subtrahierenden Matrizen     *
 *                               // nicht kompatibel sind.          *
 *             R = A-B;          // Fehler: Abbruch mit Meldung,    *
 *                               // dass die Zuweisung nicht durch- *
 *                               // gefuehrt werden kann (R hat     *
 *                               // falsche Dimensionen)            *
 ********************************************************************
 */
```

```
matrix matrix::operator - (matrix& M)
{
  SizeCheck (M);       // Teste auf Vertraeglichkeit der Dimensionen
                       // der beteiligten Matrizen

  matrix C((*this));   // Initialisiere die Ergebnismatrix mit dem
                       // ersten Argument

  /* Fuehre in der folgenden Schleife die elementweise Subtraktion
     durch
     */

  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
      C(i,j) -= M(i,j);

  return C;            // Rueckgabe der Ergebnismatrix
}

/*
 *******************************************************************
 *  Methode : matrix operator -= (matrix& M)                        *
 *                                                                 *
 *  Aufgabe : Wie schon bei der  Multiplikation und Addition wird   *
 *            hier die Kurzform von Subtraktion und Zuweisung im-   *
 *            plementiert.                                         *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Subtraktion             *
 *                                                                 *
 *  Beispiel:  matrix A(4,2), B(4,2);                              *
 *                                                                 *
 *             A -= B;          // A := A - B;                     *
 *******************************************************************
 */

matrix matrix::operator -= (matrix& M)
{
  (*this) = (*this) - M;

  return (*this);
}

/*
 *******************************************************************
 *  Methode : matrix operator - (TP x)                             *
 *                                                                 *
 *  Aufgabe : Dieser Operator implementiert die Differenzbildung    *
 *            zwischen einer Matrix und einem Skalar, die analog    *
```

```
 *           zur entsprechenden Addition erklaert ist.           *
 *                                                               *
 *  Ergebnis: vom Typ matrix, Ergebnis der Subtraktion           *
 *                                                               *
 *  Beispiel:  matrix A(6,5), B(6,5);                            *
 *                                                               *
 *            B = A - 0.2; // B(i,j):= A(i,j) - 0.2 fuer alle i,j  *
 *****************************************************************
 */

matrix matrix::operator - (TP x)
{
  matrix C((*this));      // Initialisierung der Ergebnismatrix mit
                          // dem aktuellen Matrixobjekt

  // In der folgenden Schleife wird der Wert des Skalars x von
  // jedem Element der Ergebnismatrix abgezogen.

  for(int i = 1; i <= m; i++)
    for(int j = 1; j <= n; j++)
      C(i,j) -= x;

  return C;              //  Rueckgabe des Ergebnisses
}

/*
 *****************************************************************
 *  Methode : matrix operator -= (TP x)                          *
 *                                                               *
 *  Aufgabe : Dieser Operator ist wieder die Kurzform fuer die   *
 *            gleichzeitige Subtraktion und Zuweisung            *
 *                                                               *
 *  Ergebnis: vom Typ matrix, Ergebnis der Addition              *
 *                                                               *
 *  Beispiel:  matrix Z(3,5);                                    *
 *                                                               *
 *            Z -= 1.8;       // Z(i,j):= Z(i,j)-1.8 fuer alle i,j *
 *****************************************************************
 */

matrix matrix::operator -= (TP x)
{
  return (*this) = (*this) - x;
}

/*
 *****************************************************************
 *  Funktion: matrix operator - (TP x, matrix& M)                *
```

```
 *                                                                 *
 *  Aufgabe : Falls bei einer Differenz zwischen einer Matrix und   *
 *            einem Skalar das Skalar zuerst steht, benoetigen wir  *
 *            wieder eine Implementation fuer die vertauschte       *
 *            Reihenfolge.                                          *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Subtraktion              *
 *                                                                 *
 *  Beispiel:  matrix A(3,5), B(3,5);                               *
 *                                                                 *
 *             B = 2.3 - A;  // B(i,j):= 2.3 - A(i,j) fuer alle i,j *
 *******************************************************************
 */

matrix operator - (TP x,matrix& M)
{
  return M.operator - (x);
}

/*
 *******************************************************************
 *  Methode : matrix  - ()                                          *
 *                                                                 *
 *  Aufgabe : Dieser Operator implementiert das unaere Minus fuer   *
 *            Matrizen.                                             *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Ergebnis der Operation                *
 *                                                                 *
 *  Beispiel:  matrix A(6,5), B(6,5);                               *
 *                                                                 *
 *             B = -A;     // B(i,j) := - A(i,j) fuer alle i,j      *
 *******************************************************************
 */

matrix matrix::operator - ()
{
  matrix C(m,n);      // Ergebnismatrix

  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
       C(i,j) = -(*this)(i,j);

  return C;           // Rueckgabe des Ergebnisses
}

/*
 *******************************************************************
 *  Methode : matrix  T()                                           *
```

```
 *                                                                  *
 *  Aufgabe : Diese Methode implementiert das Transponieren einer   *
 *            Matrix.                                               *
 *                                                                  *
 *  Ergebnis: vom Typ matrix, Transponierte der Matrix              *
 *                                                                  *
 *  Beispiel:  matrix A(6,2), B(2,6);                               *
 *                                                                  *
 *             B = A.T();  // B(i,j) := A(j,i) fuer alle i,j        *
 ********************************************************************
 */

matrix matrix::T()
{
  matrix C(n,m);   // Ergebnismatrix

  /* In der folgenden Schleife wird dem Element (j,i) der Ergebnis-
     matrix der Wert des Elements (i,j) der zu transponierenden
     Matrix zugewiesen
     */

  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
      C(j,i) = (*this)(i,j);

  return C;       // Rueckgabe der transponierten Matrix
}

/*
 ********************************************************************
 *  Methode : matrix  exp(int e)                                    *
 *                                                                  *
 *  Aufgabe : Mittels der Methode exp() wird das Potenzieren einer  *
 *            Matrix zur e-ten Potenz ermoeglicht. Die Potenz e     *
 *            muss ganzzahlig und die Matrix quadratisch sein.      *
 *            Die Potenzierung einer Matrix A ist hier dann wie     *
 *            folgt definiert:                                      *
 *            1.)  A.exp(0) = I,                                    *
 *                 wobei I die Einheitsmatrix entsprechender        *
 *                 Dimension ist                                    *
 *            2.)  Fuer e > 0 ist rekursiv                          *
 *                 A.exp(e) = A * A.exp(e-1)                        *
 *            3.)  A.exp (-1) = inv (A),                            *
 *                 wobei inv (A) die Inverse zu A ist               *
 *                 Die Inverse zu A wird durch Gausselimination mit *
 *                 Pivotsuche berechnet.                            *
 *            4.)  Es sei B=inv(A). Fuer e < -1 ist dann            *
 *                 A.exp (e) = B.exp (abs(e)),                      *
```

```
 *                wobei abs(e) den Absolutbetrag von e bezeichnet  *
 *           Falls die Inverse zu einer Matrix nicht existiert,    *
 *           jedoch eine negative Potenz berechnet werden soll,    *
 *           bricht das Programm mit einer Fehlermeldung ab.       *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, Potenz der Matrix wie oben definiert *
 *                                                                 *
 *  Beispiel:  matrix A(2,2), B(2,6), C(2,2);                      *
 *                                                                 *
 *            C = A.exp(2);  // In Ordnung: C:= A*A                *
 *                                                                 *
 *            C = B.exp(4);  // Fehler: Programm bricht mit der    *
 *                           // Meldung ab, dass man nur quadra-   *
 *                           // tische Matrizen potenzieren kann   *
 *******************************************************************
 */

matrix matrix::exp(int e)
{
  if(m != n)      // Teste, ob die zu potenzierende Matrix quadratisch ist
                  // Falls nicht ...
    {
                  // Abbruch mit entsprechender Fehlermeldung
      cerr << isa() << ": Man kann nur quadratische Matrizen ";
      cerr << (( e < 0) ? "invertieren " : "potenzieren ");
      cerr << endl;
      cerr << "Objekt: " << m << " x " << n << " " << isa() << endl;
      exit(-1);
    }

  // Ab hier wird nun die Berechnung durchgefuehrt.

  matrix C(m,m);       // Ergebnismatrix
  int i, j, k;         // Schleifenvariablen

  if (e == 0)          // Ist der Exponent gleich 0,
    {
      C.ident_mat();   // so gebe die Einheitsmatrix zurueck
      return C;
    }

  if (e > 0)                // Bei positivem Exponenten ...
    {
      C = (*this);          // berechne das Matrizenprodukt
      for (i = 1; i < e; i++)
        C *= (*this);
      return C;             // und liefere das Ergebnis als Rueckgabe-
                            // wert.
```

```
  }

// Ab hier folgt nun die Behandlung des Falles e < 0

int *p, r, hi;
TP Max, hr, *hv;

C = (*this);

// Wir invertieren nun C nach dem bekannten Verfahren mit
// Pivotsuche.

p = new int[m+1];

for(j = 1; j <= n; j++)
  p[j] = j;

for(j = 1; j <= n; j++)
  {
    Max = abs(C(j,j));
    r = j;

    for (i = j + 1; i <= n; i++)
      {
        if(abs(C(i,j)) > Max)
          {
            Max = abs(C(i,j));
            r = i;
          }
      }
    if (Max == 0.0)
      {
        cerr << isa() <<": Die zu invertierende ";
        cerr << m << " x " << n << " Matrix ist singulaer";
        cerr << endl;
        exit(-1);
      }
    if (r > j)
      {
        for (k = 1; k <= n; k++)
          {
            hr = C(j,k);
            C(j,k) = C(r,k);
            C(r,k) = hr;
          }
        hi = p[j];
        p[j] = p[r];
        p[r] = hi;
```

```
        }
      hr = 1.0 / C(j,j);
      for (i = 1; i <= n; i++)
        C(i,j) = hr * C(i,j);
      C(j,j) = hr;

      for (k = 1; k <= n; k++)
        if (k != j)
          {
            for (i = 1; i <= n; i++)
              if (i != j)
                {
                  C(i,k) -= C(i,j) * C(j,k);
                }
            C(j,k) = -hr * C(j,k);
          }
    }
  hv = new TP[m+1];

  for (i = 1; i<= n; i++)
    {
      for (k = 1; k <= n; k++)
        hv[p[k]] = C(i,k);
      for (k = 1; k <= n; k++)
        C(i,k) = hv[k];
    }
  delete p;
  delete hv;

  /* Die Matrix C enthaelt nun die Inverse. Wir multiplizieren
     sie nun so oft mit sich selbst, wie das der Exponent er-
     fordert
     */

  e = -e;

  for (i = 2; i <= e; i++)
    C *= C;

  return C;
}

/*
 ******************************************************************
 *  Methode : matrix  Cholesky()                                  *
 *                                                                *
 *  Aufgabe : Diese Methode implementiert die Cholesky-Zerlegung  *
 *            einer (quadratischen, positiv definiten) Matrix.    *
```

```
 *           Es wird geprueft, ob die zu zerlegende Matrix qua-    *
 *           dratisch ist. Falls nicht, wird das Programm mit      *
 *           einer Fehlermeldung abgebrochen. Falls sich bei der   *
 *           Berechnung der Zerlegung herausstellt, dass die Ma-   *
 *           trix nicht positiv definit ist, wird ebenfalls eine   *
 *           entsprechende Fehlermeldung ausgegeben und der Pro-   *
 *           grammlauf beendet.                                    *
 *                                                                 *
 *  Ergebnis: vom Typ matrix, untere Dreiecksmatrix,               *
 *           Cholesky-Zerlegung der Matrix.                        *
 *                                                                 *
 *  Beispiel:  matrix A(2,2), B(2,2), L(2,2), Z(7,2);              *
 *                                                                 *
 *            A(1,1) = 3.0;    A(1,2) = 1.0;                       *
 *            A(2,1) = 1.0;    A(2,2) = 4.0;                       *
 *                                                                 *
 *            B = 0.0;           // B ist die Nullmatrix           *
 *            L = A.Cholesky();  // Zerlegungsberechnung           *
 *                               // A = L * L.T()                  *
 *            L = Z.Cholesky();  // Fehlermeldung, weil Z nicht    *
 *                               // quadratisch ist.               *
 *            L = B.Cholesky();  // Fehlermeldung, weil B nicht    *
 *                               // positiv definit ist.           *
 *******************************************************************
 */

matrix matrix::Cholesky()
{
  int i,j,k;         //  Schleifenvariablen
  if (n != m) {      //  Teste, ob die Matrix quadratisch ist.
                     //  Falls nicht, dann Fehlermeldung und
                     //  Programmende

    cerr << isa();
    cerr << ": Die Cholesky-Zerlegung ist nur fuer quadratische";
    cerr << endl << "Matrizen berechenbar" << endl;
    cerr << "Objekt: " << m << " x " << n << " " << isa() << endl;
    exit (-1);
  }

  matrix L((*this));    // Ergebnismatrix, wird mit der zu zerlegenden
                        // Matrix initialisiert.

  // Die folgenden Schleifen berechnen nun die uebliche
  // Cholesky-Zerlegung.

  for (k=1; k<=n; k++) {
    if (L(k,k) <= 0.0) {
```

```
      cerr << isa() << ": Die " << m << " x " << n << " ";
      cerr << isa() << " ist nicht positiv definit." << endl;
      cerr << "Die Cholesky-Zerlegung kann nicht berechnet werden" << endl;
      exit (-1);
    }
    L(k,k) = sqrt (L(k,k));
    for (i=k+1; i<=n; i++) {
      L(i,k) = L(i,k)/L(k,k);
      for (j=k+1; j <=  i; j++)
        L(i,j) -= L(i,k)*L(j,k);
    }
  }

  // Fuelle mit 0.0 oberhalb der Diagnonale auf

  for (i=1; i<m; i++)
    for (k=i+1; k<=m; k++)
      L(i,k) = 0.0;

  return L;             // Rueckgabe der Cholesky-Zerlegung
}

/*
 *******************************************************************
 *  Methode : ~matrix(), DESTRUKTOR, VIRTUELL                      *
 *                                                                 *
 *  Aufgabe : Der Destruktor einer Klasse wird immer dann auto-    *
 *            matisch aufgerufen, wenn ein Objekt geloescht werden  *
 *            soll, z.B. wenn die lokale Variable ihre Gueltigkeit  *
 *            verloren hat. Der Destruktor der Matrix-Klasse gibt   *
 *            den Speicherplatz fuer die Matrix frei. Der Benutzer  *
 *            muss sich daher NICHT explizit um die Speicherverwal- *
 *            tung kuemmern, da der Aufruf des Destruktors, wie oben*
 *            erwaehnt, automatisch erfolgt.                        *
 *******************************************************************
 */

matrix::~matrix()
{
  delete a;      // Gebe Speicherplatz fuer das Array frei
}

/*
 *******************************************************************
 *  Operator:  << (ostream& out, matrix &M)                         *
 *                                                                 *
 *  Aufgabe : Der in C++ uebliche Operator << fuer die Ausgabe von  *
 *            Daten wird hier ueberladen, d.h. so umfunktioniert,   *
```

```
 *           dass er seine Funktionalitaet auch fuer die Matrizen  *
 *           behaelt. Damit bleiben alle Moeglichkeiten offen, die *
 *           dieser Operator bietet: Dateiausgabe, Bildschirmaus-  *
 *           ausgabe etc. Die Eintraege der Matrix werden wie      *
 *           folgt auf den Ausgabe-Stream out ausgegeben:          *
 *                                                                 *
 *           M(1,1) , M(1,2) , ... , M(1,n)                        *
 *           M(2,1) , M(2,2) , ... , M(2,n)                        *
 *           ...                    ...                            *
 *           M(m,1) , ...          , M(m,n)                        *
 *                                                                 *
 *           Die Elemente in einer Zeile werden dabei durch ein    *
 *           Komma getrennt.                                       *
 *                                                                 *
 *  Ergebnis: Referenz auf den Ausgabe-Stream                      *
 *                                                                 *
 *  Beispiel: matrix A(1,2);                                       *
 *                                                                 *
 *           A(1,1) = 5.0;    A(1,2) = 1.0;                        *
 *           cout << A;     // cout ist die Standard-Ausgabe, d.h. *
 *                          // normalerweise der Bildschirm. Hier  *
 *                          // sieht die Ausgabe wie folgt aus:   *
 *                          //   5.0   ,   1.0                     *
 *           cout << endl;  // Zeilenvorschub                      *
 *           cout << A.T(); // erzeugt die Ausgabe:                *
 *                          //   5.0                               *
 *                          //   1.0                               *
 *******************************************************************
 */

ostream& operator << (ostream& out, matrix& M)
{

  // Gebe die Elemente von M zeilenweise auf dem Ausgabe-Stream out aus

  for (int i=1; i<=M.m; i++)
    {
      for (int j=1; j<=M.n; j++) {
        out << setw (17) << setprecision (10) << M(i,j);
        if (j != M.n)
          out << ",";
      }
      if (i != M.m)
        out << endl;
    }

  return out;           // Rueckgabewert ist die Referenz auf den
                        // Ausgabe-Stream, wie bei dem Operator << ueblich.
```

```
}

/*
 *****************************************************************
 *  Operator:  >> (istream& in, matrix &M)                       *
 *                                                               *
 *  Aufgabe : Analog zum Ausgabe-Operator << wird hier der       *
 *            Eingabe-Operator >> fuer die Matrizen ueberladen.  *
 *            Wiederum bleibt die volle Funktionalitaet des Ein- *
 *            gabe-Operators vollkommen erhalten. Die Eingabe von *
 *            Matrizendaten aus einer Datei ist daher genauso    *
 *            moeglich, wie die Eingabe ueber die Tastatur. Die  *
 *            Eingabedaten werden auf dem Eingabe-Stream in der  *
 *            folgenden Reihenfolge erwartet:                    *
 *                                                               *
 *            M(1,1) , M(1,2) , ... , M(1,n)                     *
 *            M(2,1) , M(2,2) , ... , M(2,n)                     *
 *            ...                     ...                        *
 *            M(m,1) , ...          , M(m,n)                     *
 *                                                               *
 *            Dabei koennen zwischen den Elementen die Trenn-    *
 *            zeichen Komma, Zeilenvorschub, Tabulator und Leer- *
 *            zeichen stehen. Es ist nicht noetig, die Eingabe   *
 *            durch Zeilenvorschuebe nach einer kompletten Zeile *
 *            umzubrechen.                                       *
 *                                                               *
 *  Ergebnis: Referenz auf den Eingabe-Stream                    *
 *                                                               *
 *  Beispiel: Wir nehmen an, dass die Datei  daten.lp  wie folgt *
 *            aussieht:                                          *
 *                                                               *
 *            1.2 , 3.4                                          *
 *            4.2                                                *
 *            1.0                                                *
 *                                                               *
 *            Durch                                              *
 *                                                               *
 *            matrix A(2,2);                                     *
 *            ifstream input("daten.lp");                        *
 *            input >> A;                                        *
 *                                                               *
 *            werden die Daten aus der Datei in die Matrix A ein- *
 *            gelesen. Analog werden durch                       *
 *                                                               *
 *            cin >> A;                                          *
 *                                                               *
 *            die Eintraege von A ueber die Tastatur eingelesen. *
 *            cin ist die Standardeingabe in C++. Es werden solange *
```

```
 *           Daten gelesen, bis A "gefuellt" ist.                *
 *****************************************************************
 */

istream& operator >> (istream& in, matrix& M)
{
  double t;   // Hilfsvariable, die die naechste Eingabe vom Eingabe-
              // Stream in aufnimmt.

  // Mit der folgenden Schleife werden die Daten fuer M zeilenweise
  // aus dem Eingabe-Stream zunaechst in t eingelesen und dann in
  // M eingetragen.

  for (int i=1; i<=M.m; i++)
    {
      for (int j=1; j<=M.n; j++)
        {
          in >> ws_komma >> t;
          if (!in) {
            cerr << M.isa() << ": Die Eingabe wurde beendet, bevor alle";
            cerr << " Daten gelesen werden konnten ";
            cerr << endl;
            exit (-1);
          }
          M(i,j) = t;
        }
    }
  return in;   // Rueckgabewert ist eine Referenz auf den Eingabe-
               // Stream, wie in C++ ueblich.
}

/*
 *****************************************************************
 *  Klasse:  vector                                              *
 *                                                               *
 *          Diese Klasse ist von der Klasse matrix abgeleitet und *
 *          erbt daher alle Attribute und Methoden.              *
 *          Ein Vektor wird als Spaltenvektor angesehen, d.h. ein *
 *          Vektor mit 5 Komponenten ist eine 5x1-Matrix.        *
 *                                                               *
 *  Attribut  | Typ  | Beschreibung                              *
 *  ------------------------------------------------------------ *
 *            |      |                                           *
 *  m         | int  | Zeilenanzahl des Vektors, d.h. Dimension  *
 *  n         | int  | Spaltenanzahl, diese ist immer gleich 1   *
 *  a         | *TP  | Zeiger auf das  Array, welches die Eintraege *
 *            |      | des Vektors enthaelt                      *
 *                                                               *
```

```
 *  Alle Attribute sind protected.                               *
 *****************************************************************
 */

class vector : public matrix
{
 public:
  vector (int);
  vector (matrix&);

  char *isa();
  inline int GetDim();
  vector operator = (matrix&);
  vector operator = (vector&);
  vector operator = (TP);
  int operator == (const vector&);
  int operator == (TP);
  friend int operator == (TP x,vector& y) { return y == x; };
  int operator != (const vector&);
  int operator != (TP);
  friend int operator != (TP x, vector& V) { return V != x; };
  int operator <  (const vector&);
  int operator <  (TP);
  friend int operator < (TP x,vector& y) { return  y > x;  };
  int operator <= (const vector&);
  int operator <= (TP);
  friend int operator <= (TP x,vector& y) { return y >= x; };
  int operator > (const vector&);
  int operator > (TP);
  friend int operator > (TP x,vector& y) { return y < x;   };
  int operator >= (const vector&);
  int operator >= (TP);
  friend int operator >= (TP x,vector& V) { return V <= x; };
  TP& operator () (int i) { return matrix::operator() (i,1); };
  TP friend sp       (vector&, vector&);
  TP friend norm     (vector&);
  TP friend max_norm (vector&);
};

/*
 *****************************************************************
 *  Methode : vector (int i), KONSTRUKTOR                        *
 *                                                               *
 *  Aufgabe : Wird automatisch aufgerufen, wenn z.B. durch       *
 *             vector x(3)      oder                             *
 *             vector x(j)                                       *
 *            ein neuer Vektor erzeugt wird.  Der Konstruktor ruft *
 *            den Basisklassenkonstruktor von matrix mit i als der *
```

```
 *           Zeilenanzahl und 1 als der Spaltenanzahl auf         *
 *                                                                *
 *  Ergebnis: keines                                              *
 *                                                                *
 *  Beispiel:  vector x(3);                                       *
 *             erzeugt einen Spaltenvektor x mit 3 Eintraegen     *
 ******************************************************************
 */

vector::vector (int i) : matrix (i,1)
{
  // Leerer Rumpf, denn alle Arbeiten erledigt der Konstruktor
  // der Basisklasse
}

/*
 ******************************************************************
 *  Methode : vector (matrix& M), KONSTRUKTOR                     *
 *                                                                *
 *  Aufgabe : Dieser Konstruktor dient vor allem zur automatischen *
 *           Konversion von Matrizen nach Vektoren. Erst dadurch   *
 *           bleiben alle Methoden der Basisklasse WIRKLICH weiter *
 *           benutzbar.                                           *
 *                                                                *
 *  Ergebnis: keines                                              *
 ******************************************************************
 */

vector::vector (matrix& M) : matrix (M.GetRows(), 1)
{
  /* Durch den Zusatz  ... : matrix (M.m, 1) wird der Basisklassen-
     konstruktor aufgerufen. Er kuemmert sich um Speicherplatz.
     */

  if (M.GetCols() != 1) {    // Falls die Matrix mehr als eine Spalte besitzt,
                       // geben wir eine Fehlermeldung aus, denn dann kann
                       // sie nicht in einen Vektor konvertiert werden.
       cerr << isa() << ": Konversion der " << M.GetRows() << " x ";
       cerr << M.GetCols();
       cerr << " " << M.isa() << " in einen Vektor ist unmoeglich!" << endl;
       exit (-1);
    }
  else  // Ansonsten kopieren wir elementweise aus M in den Vektor
    {
      for (int i=1; i<=m; i++)
        a[i-1] = M(i,1);
    }
}
```

```
/*
 *****************************************************************
 *  Methode : isa(), VIRTUAL                                     *
 *                                                               *
 *  Aufgabe : Ueberschreibt die virtuelle Methode der Basisklasse *
 *            und bewirkt, dass aus allen Fehlerroutinen, ins-    *
 *            besondere der der Basisklasse, bei Fehlern mit      *
 *            Vektoren nicht "Matrix" sondern an der richtigen    *
 *            Stelle "Vektor" ausgegeben wird.                    *
 *                                                               *
 *  Ergebnis: Zeiger auf Zeichenkette mit Objekttyp, "Vektor"     *
 *****************************************************************
 */

char* vector::isa ()
{
  return "Vektor";
}

/*
 *****************************************************************
 *  Methode : int GetDim(), INLINE                               *
 *                                                               *
 *  Aufgabe : Anlog zu den Methoden GetRows() und GetCols() der   *
 *            Matrix-Klasse ist die Methode GetDim() dazu da,     *
 *            um von aussen einen  Lesezugriff auf die Dimension  *
 *            eines Vektors moeglich zu machen.                   *
 *                                                               *
 *  Ergebis : vom Typ int, Anzahl der Zeilen des Vektors          *
 *                                                               *
 *  Beispiel: matrix A(4,5);                                     *
 *            int p;                                             *
 *                                                               *
 *            p = A.GetRows();        // p = 4;                  *
 *****************************************************************
 */

inline int vector::GetDim()
{
  return m;
}

/*
 *****************************************************************
 *  Methode : vector operator = (const matrix& M)                *
 *                                                               *
 *  Aufgabe : Der Operator = implementionert die Zuweisung einer  *
```

```
 *           Matrix an einen Vektor. Die umgekehrte Zuweisung, d.h.*
 *           die Zuweisung eines Vektoren an eine Matrix, muss     *
 *           NICHT extra implementiert werden, da jeder Vektor     *
 *           durch die Erbreihenfolge ebenfalls eine Matrix ist.   *
 *           Der Operator prueft, ob die Matrix eine passende      *
 *           Dimension hat. Falls nicht, wird mit einer Fehler-    *
 *           meldung abgebrochen.                                  *
 *                                                                 *
 *  Ergebnis: vom Typ vector, Wert der Zuweisung                   *
 *                                                                 *
 *  Beispiel:  matrix A(2,1), B(2,3);                              *
 *             vector x(2);                                        *
 *                                                                 *
 *             x = A;        // In Ordnung: x(i) := A(i,1) fuer    *
 *                           // alle i                             *
 *                                                                 *
 *             x = B;        // Fehlermeldung: inkompatible Di-    *
 *                           // mensionen                          *
 *******************************************************************
 */

vector vector::operator = (matrix& M)
{
 matrix::SizeCheck(M);        // Teste auf die Vertraeglichkeit der
                              // Dimensionen

 /* Falls die Dimensionen kompatibel sind, d.h. M ist eine
    m x 1 Matrix, dann kopiere elementweise von M in den
    Vektor
    */

   for (int i=1; i<=m; i++)
     a[i-1] = M(i,1);

   return (*this);            // Rueckgabe des Wertes der Zuweisung
 }


/*
 *******************************************************************
 *                                                                 *
 *  Methode : vector operator = (const vector& y)                  *
 *                                                                 *
 *  Aufgabe : In dieser Form implementionert der Op. die Zuweisung *
 *           zwischen zwei Vektoren. Es findet wieder ein Test     *
 *           statt, ob die beiden Vektoren die gleiche Dimension   *
 *           besitzen. Falls nicht, wird, wie ueblich, eine        *
 *           Fehlermeldung fuer den Benutzer ausgegeben und das    *
```

```
 *           Der Operator prueft, ob die Matrix eine passende      *
 *           Dimension hat. Falls nicht, wird mit einer Fehler-    *
 *           meldung mit Informationen fuer den Benutzer abge-     *
 *           brochen.                                              *
 *                                                                 *
 *  Ergebnis: vom Typ vector, Wert der Zuweisung                   *
 *                                                                 *
 *  Beispiel:  vector x(2), y(2), z(5);                            *
 *                                                                 *
 *            x = y;        // In Ordnung: x := y                  *
 *                                                                 *
 *            x = z;        // Fehlermeldung: inkompatible Di-     *
 *                          // mensionen                           *
 *******************************************************************
 */

vector vector::operator = (vector& y)
{
 matrix::SizeCheck(y);       // Der uebliche Test auf die
                             // passenden Dimensionen

  // Bei passenden Dimensionen wird elementweise kopiert

   for (int i=0; i<m; i++)
     a[i] = y.a[i];

   return (*this);           // Rueckgabe des Wertes Zuweisung
}

/*
 *******************************************************************
 *  Methode : vector operator = (TP x)                             *
 *                                                                 *
 *  Aufgabe : Letztendlich wird noch die Zuweisung eines Skalars an *
 *           einen Vektor implementiert. Wie schon bei den Ma-     *
 *           trizen bedeutet diese Zuweisung, dass jeder Eintrag   *
 *           des Vektors auf den Wert des Skalars gesetzt wird.    *
 *                                                                 *
 *  Ergebnis: vom Typ vector, Wert der Zuweisung                   *
 *                                                                 *
 *  Beispiel:  vector x(2);                                        *
 *                                                                 *
 *            x = 0.5;      // In Ordnung: x(i) := 0.5 fuer alle i *
 *******************************************************************
 */

vector vector::operator = (TP x)
{
```

```
  // Ueberschreibe die Elemente des Vektors mit dem Wert x des Skalars

  for (int i=0; i<m; i++)
    a[i] = x;

  return (*this);         // Gebe den aktuellen Vektor zurueck, damit
                          // Zuweisungen der Art  x = y = 1.6 moeglich
                          // sind.
}

/*
 ******************************************************************
 *  Methode : int operator == (const vector& y)                   *
 *                                                                *
 *  Aufgabe : Test auf Gleichheit zweier Vektoren.                *
 *                                                                *
 *            ACHTUNG: Zum Test auf die Gleichheit zweier Elemente *
 *                     wird das Macro EQUAL benutzt, das standard- *
 *                     maessig als                                *
 *                       #define EQUAL(a,b)  ( a == b )           *
 *                     definiert ist. Dieses Macro kann ueber-    *
 *                     schrieben werden, um auch mehr "numerische" *
 *                     Vergleiche zu ermoeglichen. Dazu muss dann  *
 *                     VOR dem Einbinden der Datei matrix.h das   *
 *                     Macro entsprechend definiert werden. Moechte *
 *                     man z.B. zwei Elemente (vom Typ double) als *
 *                     gleich bezeichnen, wenn sie sich um maximal *
 *                     1e-5 unterscheiden, so sollte man das Macro *
 *                     als                                        *
 *                       #define EQUAL(a,b) ( fabs(a-b) <= 1e-5 )  *
 *                     definieren.                                *
 *                                                                *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.             *
 *                  1, falls (*this)(i) = y(i) fuer alle i        *
 *                     im Sinne des Macro EQUAL (s.o.)            *
 *                  0, sonst                                      *
 *                                                                *
 *            Dabei ist (*this) der aktuelle Vektor.              *
 *            Das Ergebnis ist ebenfalls 0, falls x und y unter-  *
 *            schiedliche Dimensionen besitzen.                   *
 *                                                                *
 *  Beispiel: vector x(2), y(2);                                  *
 *                                                                *
 *            if ( x == y )                                       *
 *               cout << "Beide Vektoren sind gleich!" << endl;   *
 ******************************************************************
 */
```

```
int vector::operator == (const vector& y)
{
  if (m != y.m)      // Falls der Vergleichsvektor unterschiedliche
                     // Dimension besitzt,
    return 0;        // dann ist das Ergebnis 0

  // Bei gleicher Dimension vergleiche Komponentenweise

  for (int i = 0; i < m; i++)
    if ( !(EQUAL(a[i], y.a[i])) )  // Bei einem Unterschied
      return 0;                     // ist das Ergebnis 0

  return 1;                // Im "Erfolgsfall" gebe 1 als Erbgebnis
                           // zurueck
}

/*
 *****************************************************************
 *  Methode : int operator == (TP x)                             *
 *                                                               *
 *  Aufgabe : Teste, ob alle Eintraege eines Vektors einen be-   *
 *            stimmten Wert haben.                               *
 *                                                               *
 *            ACHTUNG: Zum Vergleich eines Eintrages mit dem Skalar *
 *                     wird wieder das Macro EQUAL s.o. benutzt.  *
 *                                                               *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.            *
 *                  1, falls (*this)(i) = x fuer alle i          *
 *                     im Sinne des Macro EQUAL (s.o.)           *
 *                  0, sonst                                     *
 *            Dabei ist (*this) der aktuelle Vektor.             *
 *                                                               *
 *  Beispiel: vector x(2);                                       *
 *                                                               *
 *            if ( x ==  0.5 )                                   *
 *               cout << "Alle Eintraege in x sind 0.5" << endl; *
 *****************************************************************
 */

int vector::operator == (TP x)
{
  // Vergleiche jedes Element mit dem Wert x des Skalars

  for(int i = 0; i < m; i++)
    if ( !(EQUAL(a[i],x)) )    // Bei einem Unterschied
      return 0;                // ist das Ergebnis 0

  return 1;                    // Ansonsten hat es den Wert 1
```

```
}

/*
 ******************************************************************
 *  Methode : int operator != (const vector& y)                   *
 *                                                                *
 *  Aufgabe : Test auf Ungleichheit zweier Vektoren.              *
 *                                                                *
 *           ACHTUNG: Zum Test, ob zwei Elemente ungleich sind,   *
 *                    wird das Macro NOT_EQUAL benutzt, das per   *
 *                    Voreinstellung als                          *
 *                      #define NOT_EQUAL(a,b)  ( a != b )        *
 *                    erklaert ist. Dieses Macro kann wieder ueber-*
 *                    schrieben werden.                           *
 *                                                                *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.             *
 *                 1, falls (*this)(i) != y(i) fuer alle i        *
 *                    im Sinne des Macro NOT_EQUAL (s.o.)         *
 *                 0, sonst                                       *
 *                                                                *
 *           Dabei ist (*this) der aktuelle Vektor.               *
 *                                                                *
 *           ACHTUNG:                                             *
 *           Das Ergebnis ist ebenfalls 1, falls x und y unter-   *
 *           schiedliche Dimensionen besitzen.                    *
 *                                                                *
 *  Beispiel: vector x(2), y(2);                                  *
 *                                                                *
 *           if ( x != y )                                        *
 *              cout << "Die Vektoren sind unterschiedlich";      *
 ******************************************************************
 */

int vector::operator != (const vector& y)
{
  if(m != y.m)       // Falls der Vergleichsvektor unterschiedliche
                     // Dimension besitzt,
    return 1;        // dann ist das Ergebnis 1

  // Bei gleicher Dimension vergleiche komponentenweise

  for(int i = 0; i < m; i++)
    if ( !(NOT_EQUAL(a[i], y.a[i])) )
      return 0;

  return 1;
}
```

```
/*
 ****************************************************************
 *  Methode : int operator != (TP x)                            *
 *                                                              *
 *  Aufgabe : Teste, ob alle Eintraege eines Vektors ungleich einem *
 *            bestimmten Wert sind.                             *
 *                                                              *
 *            ACHTUNG: Zum Vergleich eines Eintrages mit dem Skalar *
 *                     wird wieder das Macro NOT_EQUAL (s.o.)   *
 *                     benutzt.                                 *
 *                                                              *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.           *
 *                  1, falls (*this)(i) = x fuer alle i         *
 *                     im Sinne des Macro EQUAL (s.o.)          *
 *                  0, sonst                                    *
 *            Dabei ist (*this) der aktuelle Vektor.            *
 *                                                              *
 *  Beispiel: vector x(2);                                      *
 *                                                              *
 *            if ( x ==  0.0 )                                  *
 *               cout<<"Die Eintraege in x sind von 0 verschieden"; *
 ****************************************************************
 */

int vector::operator != (TP x)
{
  // Vergleiche jedes Element mit dem Wert x des Skalars

  for(int i = 0; i < m; i++)
    if ( !(NOT_EQUAL(a[i],x)) )
      return 0;

  return 1;                    // Ansonsten hat es den Wert 1
}

/*
 ****************************************************************
 *  Methode : int operator <  (const vector& y)                 *
 *                                                              *
 *  Aufgabe : Testet, ob die Elemente des aktuellen Vektors strikt  *
 *            elementweise kleiner sind als die des Vektors y   *
 *                                                              *
 *            ACHTUNG: Zum Vergleich von zwei Eintragen der beiden  *
 *                     Vektoren wird das Macro LESS benutzt. Es ist *
 *                     standardmaessig als                      *
 *                       #define LESS(a,b)  ( a < b )           *
 *                     definiert. Dieses Macro kann analog zum  *
 *                     Macro EQUAL (s.o.) ueberschrieben werden, um *
```

```
 *                     die Vergleiche dem jeweiligen Geschmack an-  *
 *                     zupassen.                                    *
 *                                                                  *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.               *
 *                  1, falls (*this)(i) < y(i) fuer alle i          *
 *                     im Sinne des Macro LESS                      *
 *                  0, sonst                                        *
 *            Dabei ist (*this) der aktuelle Vektor.                *
 *            Das Ergebnis ist ebenfalls 0, falls x und y unter-    *
 *            schiedliche Dimensionen besitzen.                     *
 *                                                                  *
 *  Beispiel: vector x(2), y(2);                                    *
 *                                                                  *
 *            if ( x < y )                                          *
 *               cout << "x ist komp.weise kleiner als y!" << endl; *
 ********************************************************************
 */

int vector::operator <  (const vector& y)
{
  if(m != y.m)          // Bei unterschiedlichen Dimensionen
    return 0;           // hat der Vergleich den Wert 0

  // Ansonsten wird komponentenweise verglichen

  for(int i = 0; i < m; i++)
    if( ! (LESS(a[i], y.a[i])) ) // Falls ein Eintrag nicht kleiner ist,
      return 0;                  // hat der Vergleich den Wert 0

  return 1;            // Erfolgsfall, gebe 1 zurueck
}

/*
 ********************************************************************
 *  Methode : int operator <  (TP x)                                *
 *                                                                  *
 *  Aufgabe : Testet, ob alle Eintraege eines Vektors kleiner als   *
 *            ein Skalar sind.                                      *
 *                                                                  *
 *            ACHTUNG: Zum Vergleich von zwei Eintragen der beiden  *
 *                     Vektoren wird wieder das Macro LESS (s.o.)   *
 *                     benutzt.                                     *
 *                                                                  *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.               *
 *                  1, falls (*this)(i) < x fuer alle i             *
 *                     im Sinne des Macro LESS                      *
 *                  0, sonst                                        *
 *            Dabei ist (*this) der aktuelle Vektor.                *
```

```
 *                                                                  *
 *  Beispiel: vector x(2);                                          *
 *                                                                  *
 *            if ( x < 0 )                                          *
 *               cout << "Alle Eintraege in x sind negativ" << endl;*
 ********************************************************************
 */

int vector::operator <  (TP x)
{
  // Elementweiser Vergleich

  for (int i = 0; i < m; i++)
    if( !(LESS(a[i], x)) )
      return 0;

  return 1;
}

/*
 ********************************************************************
 *  Methode : int operator <= (const vector& y)                     *
 *                                                                  *
 *  Aufgabe : Testet, ob die Eintraege im aktuellen Vektor kompo-   *
 *            nentenweise kleiner oder gleich denen im Vektor y     *
 *            sind.                                                 *
 *                                                                  *
 *            ACHTUNG: Zum Test, ob ein Element kleiner oder gleich *
 *                     einem bestimmten Wert ist, wird das Macro    *
 *                     LESS_OR_EQUAL benutzt, dass per Vorein-      *
 *                     stellung als                                 *
 *                       #define LESS_OR_EQUAL(a,b)  ( a <= b )     *
 *                     definiert ist. Dieses Macro kann wieder wie  *
 *                     bereits EQUAL oder LESS ueberschrieben wer-  *
 *                     den, um die Anpassung der Vergleiche an den  *
 *                     Verwendungszweck anzupassen.                 *
 *                                                                  *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.               *
 *                  1, falls (*this)(i) <= y(i) fuer alle i         *
 *                     im Sinne des Macros LESS_OR_EQUAL            *
 *                  0, sonst                                        *
 *                                                                  *
 *            Dabei ist (*this) der aktuelle Vektor.                *
 *            Das Ergebnis ist ebenfalls 0, falls x und y unter-    *
 *            schiedliche Dimensionen besitzen.                     *
 *                                                                  *
 *  Beispiel: vector x(2), y(2);                                    *
 *                                                                  *
```

```
 *           if ( x <= y)                                       *
 *              cout << "x ist komponentenweise <= y" << endl;     *
 ******************************************************************
 */

int vector::operator <= (const vector& y)
{
  if(m != y.m)      // Wenn y eine andere Dimension als der
                    // aktuelle Vektor besitzt,
    return 0;       // ist der Wert des Vergleiches 0

  // Bei gleichen Dimensionen wird wieder (wie ueblich)
  // komponentenweise verglichen

  for(int i = 0; i < m; i++)
    if( !(LESS_OR_EQUAL(a[i], y.a[i])) )
      return 0;
  return 1;
}

/*
 ******************************************************************
 *  Methode : int operator <= (TP x)                              *
 *                                                                *
 *  Aufgabe : Wie der letzte Operator. Nur Vergleich jedes Elements *
 *           des aktuellen Vektors mit dem Wert x des Skalars.    *
 *                                                                *
 *           ACHTUNG: Zum Test, ob ein Element kleiner oder gleich *
 *                    x ist, wird wieder das Macro LESS_OR_EQUAL   *
 *                    (s.o.) verwendet.                           *
 *                                                                *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.             *
 *                 1, falls (*this)(i) <= x fuer alle i            *
 *                    im Sinne des Macros LESS_OR_EQUAL            *
 *                 0, sonst                                       *
 *                                                                *
 *           Dabei ist (*this) der aktuelle Vektor.               *
 *                                                                *
 *  Beispiel: vector x(2), y(2);                                  *
 *                                                                *
 *           if ( x <= y)                                         *
 *              cout << "x ist nicht positiv!" << endl;           *
 ******************************************************************
 */

int vector::operator <= (TP x)
{
  // Elementweiser Vergleich
```

```
  for(int i = 0; i < m; i++)
    if( !(LESS_OR_EQUAL(a[i], x)) )
      return 0;
  return 1;
}

/*
 *******************************************************************
 *  Methode : int operator >  (const vector& y)                    *
 *                                                                 *
 *  Aufgabe : Elementweiser "groesser"-Vergleich zwischen zwei Vek- *
 *            toren.                                               *
 *                                                                 *
 *            ACHTUNG: Zum Vergleich zweier Elemente wird das Macro *
 *                     GREATER verwendet, das per default als      *
 *                       #define GREATER(a,b)   (a > b)            *
 *                     definiert ist. Wie schon EQUAL, LESS und    *
 *                     LESS_OR_EQUAL kann man durch Umdefinieren   *
 *                     dieses Macros den Vergleich seinem Geschmack *
 *                     anpassen.                                   *
 *                                                                 *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.              *
 *                  1, falls (*this)(i) > y(i) fuer alle i         *
 *                     im Sinne des Macros GREATER                 *
 *                  0, sonst                                       *
 *                                                                 *
 *            Dabei ist (*this) der aktuelle Vektor.               *
 *            Das Ergebnis ist ebenfalls 0, falls x und y unter-   *
 *            schiedliche Dimensionen besitzen.                    *
 *                                                                 *
 *  Beispiel: vector x(2), y(2);                                   *
 *                                                                 *
 *            if ( x > y )                                         *
 *               cout << "x ist komp.weise groesser als y" << endl; *
 *******************************************************************
 */

int vector::operator >  (const vector& y)
{
  if(m != y.m)          // Bei unterschiedlichen Dimensionen
    return 0;           // ist das Ergebnis 0

  for(int i = 0; i < m; i++)
    if( !(GREATER(a[i], y.a[i])) )
      return 0;
  return 1;
}
```

```
/*
 ******************************************************************
 *  Methode : int operator  > (TP x)                              *
 *                                                                *
 *  Aufgabe : Wie letzter Operator, nur Vergleich mit einem Skalar  *
 *                                                                *
 *            ACHTUNG: Zum Vergleich wird wieder das Macro GREATER  *
 *                     (s.o.) benutzt.                            *
 *                                                                *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.             *
 *                  1, falls (*this)(i) > x fuer alle i           *
 *                     im Sinne des Macros GREATER                *
 *                  0, sonst                                      *
 *                                                                *
 *            Dabei ist (*this) der aktuelle Vektor.              *
 *                                                                *
 *  Beispiel: vector x(2);                                        *
 *                                                                *
 *            if ( x > 0.0 )                                      *
 *               cout << "x ist echt positiv" << endl;            *
 ******************************************************************
 */

int vector::operator >  (TP x)
{
  for(int i = 0; i < m; i++)
    if( !(GREATER(a[i],x)) )
      return 0;
  return 1;
}

/*
 ******************************************************************
 *  Methode : int operator >= (const vector& y)                   *
 *                                                                *
 *  Aufgabe : "Groesser oder gleich"-Test zweier Vektoren.        *
 *                                                                *
 *            ACHTUNG: Zum Vergleich zweier Elemente wird das Macro *
 *                     GREATER_OR_EQUAL benutzt, dass standard-   *
 *                     maessig als                                *
 *                       #define GREATER_OR_EQUAL(a,b)  ( a >= b )  *
 *                     definiert ist. Dieses Macro kann wieder    *
 *                     ueberschrieben werden.                     *
 *                                                                *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.             *
 *                  1, falls (*this)(i) >= y(i) fuer alle i       *
 *                     im Sinne des Macros GREATER_OR_EQUAL       *
```

```
 *                 0, sonst                                    *
 *                                                             *
 *           Dabei ist (*this) der aktuelle Vektor.            *
 *           Das Ergebnis ist ebenfalls 0, falls x und y unter-    *
 *           schiedliche Dimensionen besitzen.                 *
 *                                                             *
 *  Beispiel: vector x(2), y(2);                               *
 *                                                             *
 *           if ( x >= y)                                      *
 *              cout << "x ist komp.weise groeser oder gleich y";  *
 ***************************************************************
 */

int vector::operator >= (const vector& y)
{
  if(m != y.m)
    return 0;

  for(int i = 0; i < m; i++)
    if( !(GREATER_OR_EQUAL(a[i], y.a[i])) )
      return 0;
  return 1;
}

/*
 ***************************************************************
 *  Methode : int operator >= (TP x)                           *
 *                                                             *
 *  Aufgabe : Wie letzter Operator, nur Vergleich mit Skalar   *
 *                                                             *
 *           ACHTUNG: Zum Test wird wieder das Macro           *
 *                    GREATER_OR_EQUAL verwendet.              *
 *                                                             *
 *  Ergebnis: vom Typ int, Wert des Vergleiches, d.h.          *
 *                 1, falls (*this)(i) >= x fuer alle i        *
 *                    im Sinne des Macros GREATER_OR_EQUAL     *
 *                 0, sonst                                    *
 *                                                             *
 *           Dabei ist (*this) der aktuelle Vektor.            *
 *                                                             *
 *  Beispiel: vector x(2);                                     *
 *                                                             *
 *           if ( x >= 0.0 )                                   *
 *              cout << "x ist nichtnegativ" << endl;          *
 ***************************************************************
 */

int vector::operator >= (TP x)
```

```
{
  for(int i = 0; i < m; i++)
    if( !(GREATER_OR_EQUAL(a[i], x)) )
      return 0;
  return 1;
}

/*
 *******************************************************************
 *  Funktion: TP sp (vector& x, vector& y)                         *
 *                                                                 *
 *  Aufgabe : Berechnet das Skalarprodukt der Vektoren x und y.    *
 *            Es wird geprueft, ob beide Vektoren die selbe Dimen- *
 *            sion besitzen. Falls nicht, wird mit einer ent-      *
 *            sprechenden Fehlermeldung abgebrochen.               *
 *                                                                 *
 *  Ergebnis: vom Typ TP, Skalarprodukt von x und y                *
 *                                                                 *
 *  Beispiel: vector x(2), y(2)                                    *
 *                                                                 *
 *            cout << "Skalarprodukt von x und y = " << sp(x,y);   *
 *******************************************************************
 */

TP sp (vector& x, vector& y)
{
  if (x.m != y.m)          // Bei unterschiedlichen Dimensionen
    {
      cerr << x.isa() << ": ";
      cerr << "Inkompatible Dimensionen fuer das Skalarprodukt";
      cerr << endl;
      cerr << "1. Operand: " << x.m << " x " << x.n << " " << x.isa();
      cerr << endl;
      cerr << "2. Operand: " << y.m << " x " << y.n << " " << y.isa();
      cerr << endl;
      exit (-1);
    }

  TP r = 0.0;

  for (int i=1; i<= x.GetDim(); i++)
    r += x(i)*y(i);

  return r;
}

/*
 *******************************************************************
```

```
 *  Funktion: TP norm (vector& x)                                  *
 *                                                                 *
 *  Aufgabe : Berechnet die Euklidische Norm eines Vektors.        *
 *            Es gilt norm (x)  = sqrt( sp(x,x) );                 *
 *                                                                 *
 *  Ergebnis: vom Typ TP, Euklidische Norm des Vektors.            *
 *                                                                 *
 *  Beispiel: vector x(2);                                         *
 *                                                                 *
 *            x(1) = 3.0;   x(2) = 4.0;                            *
 *                                                                 *
 *            cout << "Norm: " << norm(x);  // Norm ist 5.0        *
 *******************************************************************
 */

TP norm (vector& x)
{
  return sqrt( sp(x,x) );
}

/*
 *******************************************************************
 *  Funktion: TP max_norm (vector& x)                              *
 *                                                                 *
 *  Aufgabe : Berechnet die Maximums-Norm eines Vektors.           *
 *            Es gilt max_norm (x)  = max { |x(i)| : i=1, ...}     *
 *                                                                 *
 *  Ergebnis: vom Typ TP, Maximums-Norm des Vektors.               *
 *                                                                 *
 *  Beispiel: vector x(2);                                         *
 *                                                                 *
 *            x(1) = 3.0;   x(2) = 4.0;                            *
 *                                                                 *
 *            cout << "Max-Norm: " << norm(x);  // Max-Norm ist 4.0 *
 *******************************************************************
 */

TP max_norm (vector& x)
{
  TP ergebnis = 0.0;
  TP hilfsvar;

  for (int i=1; i<=x.GetDim(); i++) {
    hilfsvar = fabs(x(i));
    if (hilfsvar > ergebnis)
      ergebnis = hilfsvar;
  }
```

```
  return ergebnis;
}

/*
 *****************************************************************
 *  Funktion: vector LT_Solve (matrix& L, vector& b)              *
 *                                                                *
 *  Aufgabe : Loest ein Gleichungssystem mit rechter Seite b und  *
 *            Matrix L. Dabei wird L als untere Dreiecksmatrix     *
 *            angenommen (und dementsprechend nur das untere Drei- *
 *            eck von L betrachtet.                                *
 *            Falls das Gleichungssystem   L*x = b  nicht loesbar  *
 *            ist, wird mit einer entsprechenden Fehlermeldung     *
 *            abgebrochen.                                         *
 *                                                                *
 *  Ergebnis: vom Typ vector, Loesung von L*x = b                 *
 *                                                                *
 *****************************************************************
 */

vector LT_Solve(matrix& L,vector& b)
{
  TP sum;
  vector r(L.GetCols());

  if (L.GetRows() != L.GetCols()) {
   cerr << L.isa() << ": Zum Loesen eines gestaffelten Sytems ist ";
   cerr << " eine quadratische Matrix erforderlich." << endl;
   cerr << "Objekt: " << L.GetCols() << " x " << L.GetRows() << " ";
   cerr << L.isa() << endl;
   exit (-1);
  }

  for (int i=1; i<=L.GetCols(); i++) {
      sum = b(i);
      for (int j=1; j <=i-1; j++)
        sum -= L(i,j)*r(j);

      if (L(i,i) == 0.0) {
        cerr << L.isa() << ": Die untere Dreiecksmatrix des ";
        cerr << "gestaffelten Systems ist singulaer" << endl;
        exit (-1);
      }
      r(i) = sum/L(i,i);
    }

  return r;
}
```

```
/*
 *****************************************************************
 *  Funktion: vector UT_Solve (matrix& U, vector& b)             *
 *                                                               *
 *  Aufgabe : Wie LT_Solve, nur wird die Matrix als obere Dreiecks- *
 *            matrix angenommen.                                 *
 *                                                               *
 *  Ergebnis: vom Typ vector, Loesung von U*x = b                *
 *                                                               *
 *****************************************************************
 */

vector UT_Solve(matrix& U,vector& b)
{
  TP sum;
  vector r(U.GetCols());

  if (U.GetRows() != U.GetCols()) {
   cerr << U.isa() << ": Zum Loesen eines gestaffelten Sytems ist ";
   cerr << " eine quadratische Matrix erforderlich." << endl;
   cerr << "Objekt: " << U.GetCols() << " x " << U.GetRows() << " ";
   cerr << U.isa() << endl;
   exit (-1);
  }

  for (int i=U.GetCols(); i>=1; i--) {
    sum = b(i);
      for (int j = i+1 ; j <= U.GetCols(); j++)
        sum -= U(i,j)*r(j);

    if (U(i,i) == 0.0) {
      cerr << U.isa() << ": Die obere Dreiecksmatrix des ";
      cerr << "gestaffelten Systems ist singulaer" << endl;
      exit (-1);
    }
  r(i) = sum/U(i,i);
  }

  return r;
}

/*
 *****************************************************************
 *  Funktion: ws_komma (istream& in)                             *
 *                                                               *
 *  Aufgabe : ws_komma ist ein istream-Manipulator. Er besitzt die  *
 *            Aufgabe, eine beliebige Anzahl von Whitespacezeichen  *
```

```
 *           und Kommas zu ueberlesen.                            *
 *                                                                *
 *  Ergebnis: vom Typ istream, Status des Eingabestream in        *
 *                                                                *
 ******************************************************************
 */

istream& ws_komma (istream& in)
{
  char c;
  int ende;

  ende = 0;

  while (!ende && in.get(c)) {
    if (c!=EOF) {
      if (!isascii(c)) {
        cerr << "Ungueltiges Zeichen bei der Eingabe.";
        cerr << " Zeichencode: " << c << endl;
        exit (-1);
      }
      if (!isspace(c) && c!= ',') {
        in.putback(c);
        ende = 1;
      }
    }
  }
  return in;
}

/*
 ******************************************************************
 *  Funktion: no_eof (istream& in)                                *
 *                                                                *
 *  Aufgabe : no_eof ist ebenfalls ein istream-Manipulator. Er    *
 *            testet auf das Dateiende und bricht das Programm mit *
 *            einer Fehlermeldung ab, falls versucht wird, ueber  *
 *            ueber das Dateiende hinaus zu lesen.                *
 *                                                                *
 *  Ergebnis: vom Typ istream, Status des Eingabestream in        *
 *                                                                *
 ******************************************************************
 */

istream& no_eof (istream &in)
{
  char c;
```

```
  if (!in.get(c) || c == EOF) {
    cerr << "Fehler: Vorzeitiges Ende beim Lesen des Eingabestreams";
    cerr << endl << endl;
    exit (-1);
  }
  else
    in.putback(c);
  return in;
}

#endif
```

B.2 Der Parser für die Konfigurationsdatei

B.2.1 Lexikalische Analyse: lexer.lex

```
%{
/*
   **********************************************************************
   **********************************************************************
   **    Diplomarbeit von   Sven Oliver Krumke,                        **
   **                       geb. am 22.02.68                           **
   **                       in Frankfurt am Main                       **
   **    Betreuer         : Prof. Dr. J. Stoer                         **
   **                                                                  **
   **    Datei            : lexer.lex                                  **
   **                                                                  **
   **    erstellt am      : 24.11.93                                   **
   **    letzte Aenderung : 11.01.94                                   **
   **                                                                  **
   **    Aufgabe          : Datei, die die Spezifikationen fuer die    **
   **                       regulaeren Ausdruecke enthaelt, welche     **
   **                       fuer die lexikalische Analyse des          **
   **                       Konfigurationsfiles notwendig sind.        **
   **                       Das in dieser Datei verwendete Format ist **
   **                       das fuer LEX bzw. FLEX uebliche Format.    **
   **********************************************************************
   **********************************************************************
*/

%}

/*
   Es folgen die Definitionen der regulaeren Ausdruecke fuer
   die Grammatik zum Parsen des Konfigurationsfiles
```

```
  */

COMMENT   #(.)*

EQUAL     =
NEWLINE   \n

DIGIT     [0-9]
CHAR      [a-zA-Z]
WHITE     [ \t]+

E1FLOAT   {DIGIT}+"."{DIGIT}*([Ee]([+\-])?{DIGIT}*)?
E2FLOAT   {DIGIT}[Ee]([+\-])?{DIGIT}*
INT       {DIGIT}+
FLOAT     {E1FLOAT}|{E2FLOAT}|{INT}

KEINE     keine

LP        lp
QP        qp
AUTO      automatisch

EPS       eps_newton
GAMMA     gamma

COMPL     term_komplementaritaet
PRIMINF   term_prim_unzulaessigkeit
DUALINF   term_dual_unzulaessigkeit

K         barriere_parameter
K_MAX     maximale_barriere

INC       barriere_inkrement
FACTOR    barriere_faktor

PROBLEM   problem

M         m

BARRIER   barriere_methode
KONST     konstant
VAR       variabel
UNZ       unzulaessig


%%

{WHITE}         /* "Whitespace"-Zeichen werden ueberlesen       */
```

```
{COMMENT}       { return LL_COMMENT; }    /* Kommentar erkannt  */

{EQUAL}         { return LL_EQUAL;   }    /* Gleichheitszeichen */

{NEWLINE}       { line++;            }    /* Zeilenumbruch      */

   /*
   Wird eines der Schluesselwoerter erkannt, so wird das ent-
   sprechende TOKEN dem Parser geliefert.
   */

{KEINE}         { return LL_KEINE;     }
{AUTO}          { return LL_AUTO;      }

{QP}            { return LL_QP;        }
{LP}            { return LL_LP;        }

{COMPL}         { return LL_COMPL;     }
{PRIMINF}       { return LL_PRIMINF;   }
{DUALINF}       { return LL_DUALINF;   }


{EPS}           { return LL_EPS;       }
{GAMMA}         { return LL_GAMMA;     }

{K}             { return LL_BARRIER;   }
{K_MAX}         { return LL_K_MAX;     }
{INC}           { return LL_INC;       }
{FACTOR}        { return LL_FACTOR;    }
{PROBLEM}       { return LL_PROBLEM;   }

{M}             { return LL_M;         }

{BARRIER}       { return LL_METHODE;   }
{KONST}         { return LL_CONST;     }
{VAR}           { return LL_VAR;       }
{UNZ}           { return LL_UNZ;       }

  /*
   Wird eine Fliesskommazahl oder eine ganze Zahl erkannt,
   so wird mit Hilfe der Funktion atof() der Standardbibliothek
   ihr numerischer Wert bestimmt und dem Parser geliefert.
   */

{FLOAT}         { yylval = atof(yytext); return LL_FLOAT;  }

  /*
```

```
   Wird keiner der obigen regulaeren Ausdruecke erkannt, so liegt
   ein Syntax-Fehler vor.
   Das Lesen der Konfigurationsdatei modbfm.rc wird mit einer
   entsprechenden Fehlermeldung abgebrochen.
   */

.               {
                  fprintf (stderr, "modbfm: ");
                  fprintf (stderr, "Syntaxfehler in Zeile ");
                  fprintf (stderr, "%d von modbfm.rc", line);
                  fprintf (stderr, "\n");
                  exit (-1);
                 }
%%
```

B.2.2 Syntaktische Analyse: parser.y

```
%{
/*
   *********************************************************************
   *********************************************************************
   **    Diplomarbeit von   Sven Oliver Krumke,                       **
   **                       geb. am 22.02.68                          **
   **                       in Frankfurt am Main                      **
   **    Betreuer         : Prof. Dr. J. Stoer                        **
   **                                                                 **
   **    Datei            : parser.y                                  **
   **                                                                 **
   **    erstellt am      : 24.11.93                                  **
   **    letzte Aenderung : 11.01.94                                  **
   **                                                                 **
   **    Aufgabe          : Datei, die die attributierte Grammatik    **
   **                         fuer den Konfigurationsfile-Parser      **
   **                         enthaelt.                               **
   **                       Die simple Grammatik ist eine LALR(1)-    **
   **                       Grammatik im Eingabeformat fuer yacc bzw. **
   **                       bison.                                    **
   *********************************************************************
   *********************************************************************
*/

#include "parser.h"
#include "main.h"
#include "parser.tab.h"
#include "lex.yy.c"

/*
```

```
   Definition der Variablen line, in der die Zeilen der Konfigurations-
   datei gezaehlt werden, damit bei Syntaxfehlern die Zeilennummer
   ausgegeben werden kann
   */

int line;

%}

/*
   Es folgt die Deklaration der einzelnen Schluesselwoerter (Token),
   die von der lexikalischen Analyse geliefert werden.
   */

%token LL_COMMENT

%token LL_EQUAL

%token LL_AUTO LL_KEINE

%token LL_QP LL_LP

%token LL_COMPL LL_PRIMINF LL_DUALINF

%token LL_EPS LL_GAMMA

%token LL_BARRIER LL_K_MAX LL_INC LL_FACTOR LL_M

%token LL_METHODE LL_CONST LL_VAR LL_UNZ

%token LL_FLOAT

%token LL_PROBLEM

/*
   Das Startsymbol dieser Einfachst-Grammatik ist das Symbol YY_FILE
   */

%start YY_FILE

%%

/*
   Hier beginnt die attributierte LALR(1)-Grammatik zum Parsen der
   Konfigurationsdatei
   */

YY_FILE       : YY_STMNT_LIST;
```

```
YY_STMNT_LIST : YY_STMNT YY_STMNT_LIST
              | YY_STMNT;

YY_STMNT      : LL_COMMENT                     /* Kommentar parsen */

              /* Anweisung:    eps_newton = {FLOAT}, z.B.
                               eps_newton = 0.5               */

              | LL_EPS LL_EQUAL LL_FLOAT      { eps_newton      = $3;}

              /* Anweisung:    gamma = {FLOAT}, z.B.
                               gamma = 1.0e5               */

              | LL_GAMMA   LL_EQUAL LL_FLOAT  { gamma_mbf       = $3;}

              /* Anweisung:    term_komplementaritaet = {FLOAT} */

              | LL_COMPL  LL_EQUAL LL_FLOAT   { t_komplement    = $3;}

              /* Anweisung:    term_prim_unzulaessigkeit = {FLOAT} */

              | LL_PRIMINF LL_EQUAL LL_FLOAT  { t_prim_unz      = $3;}

              /* Anweisung:    term_dual_unzulaessigkeit = {FLOAT} */

              | LL_DUALINF LL_EQUAL LL_FLOAT  { t_dual_unz      = $3;}

              /* Anweisung:    barriere_parameter = {FLOAT}        */

              | LL_BARRIER LL_EQUAL LL_FLOAT  { k_start         = $3;}

              /* Anweisung:    maximale_barriere = {FLOAT}         */

              | LL_K_MAX LL_EQUAL LL_FLOAT    { k_max           = $3;}

              | LL_K_MAX LL_EQUAL LL_KEINE    { k_max           = -1.0;}

              /* Anweisung:    barriere_methode = konstant
                               barriere_methode = variabel
                               barriere_methode = unzulaessig      */

              | LL_METHODE LL_EQUAL LL_CONST  { b_methode     = KNST;}
              | LL_METHODE LL_EQUAL LL_VAR    { b_methode     = VAR; }
              | LL_METHODE LL_EQUAL LL_UNZ    { b_methode     = UNZ; }

              /* Anweisung:   barriere_inkrement = {FLOAT}         */
```

```
            | LL_INC LL_EQUAL LL_FLOAT      { k_inkrement   = $3;  }

            /* Anweisung:   barriere_faktor = {FLOAT}            */

            | LL_FACTOR LL_EQUAL LL_FLOAT   { k_faktor      = $3;  }

            /* Anweisung:   standard_problem = QP
                            bzw.
                            standard_problem = LP                */

            | LL_PROBLEM LL_EQUAL LL_QP     { prog_standard = QP;  }
            | LL_PROBLEM LL_EQUAL LL_LP     { prog_standard = LP;  }
            | LL_PROBLEM LL_EQUAL LL_AUTO   { prog_standard = AUTO;}

            /* Anweisung:   M = {FLOAT}                          */

            | LL_M LL_EQUAL LL_FLOAT        { M = $3;             }
%%

int yyerror (char *s)
{
  fprintf (stderr, "modbfm: ");
  fprintf (stderr, "Syntaxfehler in Zeile ");
  fprintf (stderr, "%d von modbfm.rc", line);
  fprintf (stderr, "\n");
  exit (-1);
}
```

B.3 Das Hauptmodul: `main.c`

```
/*
   *******************************************************************
   *******************************************************************
   **    Diplomarbeit von   Sven Oliver Krumke,                     **
   **                       geb. am 22.02.68                        **
   **                       in Frankfurt am Main                    **
   **    Betreuer         : Prof. Dr. J. Stoer                      **
   **                                                               **
   **    Datei            : main.c                                  **
   **                                                               **
   **    erstellt am      : 24.11.93                                **
   **    letzte Aenderung : 21.02.94                                **
   **                                                               **
   **    Aufgabe          : Enthaelt die Implementatierung der      **
   **                       Modifizierten Barrieremethode, (vgl.    **
```

```
   **                      R. Polyak: Modified Barrierfunctions,     **
   **                      Math. Progr. 54, 1992, pp. 177-222)       **
   **                      fuer konvexe quadratische (und lineare)   **
   **                      Programme der Form                        **
   **                                                                **
   **                           1  T       T                         **
   **                      min  - x Q x + c x                        **
   **                           2                                    **
   **                                                                **
   **                        x : A x > b                             **
   **                                =                               **
   ******************************************************************
   ******************************************************************
*/

/*
   Als erstes werden die benoetigten Funktionsdeklarationen
   fuer die Funktionen aus der Standardbibliothek eingebunden
   */

#include <stdio.h>          // freopen()
#include <iostream.h>       // cin, cout, cerr
#include <fstream.h>        // ifstream, ofstream
#include <string.h>         // strlen()

/*
   Nun werden die Header Files mit den globalen Deklarationen
   von Variablen und Typen eingebunden.
   */


#include "matrix.h"      // Vektor- und Matrix-Klasse samt
                         // zugehoerigen Operatoren
#include "main.h"        // Externe Variablen und
                         // Funktionsdeklarationen des Hauptmoduls
#include "parser.h"      // Variablen des Parsers


/*
   Prototypen der Funktionen des Hauptmoduls
   */


ETYPE  F     (matrix&, vector&,
              matrix&, vector&,
              vector&, vector&, ETYPE);

vector dF    (matrix&, vector&,
```

```
              matrix&, vector&,
              vector&, vector&, ETYPE);

matrix HF    (matrix&,
              matrix&, vector&,
              vector&, vector&, ETYPE);

void         falsche_option (char *);

void tabelle (ETYPE, ETYPE, ETYPE, ETYPE, ETYPE );

/*
   Definition der globalen Variablen, die ihre Werte durch
   die Eintragungen im File
            modbfm.rc
   erhalten koennen.
   */

int   prog_standard;    // Variable, die angibt, ob per Voreinstellung mit
                        // einem quadratischen oder linearen Programm
                        // gearbeitet werden soll.
                        // prog_standard == AUTO : Dateiendung entscheidet
                        //                        ueber Programmtyp
                        // prog_standard == LP   : Immer lineare Programme
                        // prog_standard == QP   : Immer quadratische Progr.

ETYPE k_start;          // Start-Barriereparameter
ETYPE k_max;            // Maximaler Barriereparameter

ETYPE eps_newton;       // Genauigkeit bei der Newton-Iteration
                        // zur Loesung der Probleme
                        //       min F(x,y,k)

int   b_methode;        // Variable, die bestimmt, mit welcher
                        // Version der Modifizierten Barrieremethode
                        // gearbeitet werden soll.
                        // b_methode == KNST  : konstanter B-Parameter
                        // b_methode == VAR   : Barriereanpassung
                        // b_methode == UNZ   : Start mit unz. Vektor

ETYPE k_inkrement;      // Inkrement bei wechselnden
                        // Barriereparametern (nur bei VAR oder UNZ)

ETYPE k_faktor;         // Faktor zur Aenderung des Barriereparameters
                        // (nur bei VAR oder UNZ)

ETYPE gamma_mbf;        // Testparameter bei wechselnden
                        // Barriereparametern und Faktor zur Erhoehung
```

```
                        // der Newton-Genauigkeit

ETYPE t_komplement;     // Abbruchschranke fuer die Komplementaritaet
                        //   m
                        //  ---        T
                        //  \   y   | a  x - b  |
                        //  /    j     j      j
                        //  ---
                        //  j=1

ETYPE t_prim_unz;       // Abbruchschranke fuer die primaere
                        // Unzulaessigkeit
                        //                 T
                        //  max ( max (0, a  x - b ) )
                        //   j             j      j

ETYPE t_dual_unz;       // Abbruchschranke fuer die duale
                        // Unzulaessigkeit
                        //
                        //  || dF(x,y,k) ||oo

ETYPE M;                // Parameter bei der Transformation auf (QP)^(M)



/*
   Definition der restlichen globalen Variablen
   */

int   quadrat;          // Variable, die angibt, ob momentan gerade
                        // mit eine quadratischen Programm gearbeitet
                        // wird.

int n;                  // Anzahl der Variablen
int m;                  // Anzahl der Restriktionen

int funkt;              // Anzahl der Funktionsauswertungen
int grad;               // Anzahl der Gradientenauswertungen
int hesse;              // Anzahl der Auswertungen der Hessematrix


/* ---------------------------------------------------------------- */


/*

 ******************************************************************
 *                                                                *
```

```
 *  Funktion: ETYPE F(matrix& Q, vector&c,                         *
 *                    matrix& A, vector& b,                        *
 *                    vector&x, vector&lambda, ETYPE k)            *
 *                                                                 *
 *  Aufgabe : Berechnet den Funktionswert der Modifizierten        *
 *            Barrierefunktion                                     *
 *                                   m                             *
 *               1  T       T    1  ---              T             *
 *    F(x,y,k) = - x Q x + c x - -  \  lambda  log [k (a x - b ) +1]*
 *               2               k  /       j          j      j    *
 *                                  ---                            *
 *                                 j = 1                           *
 *                                                                 *
 *  Ergebnis: Vom Typ ETYPE, Funktionswert F(x,lambda,k)           *
 *                                                                 *
 *******************************************************************

 */

ETYPE F(matrix& Q, vector& c,
        matrix& A, vector& b,
        vector& x, vector& lambda,
        ETYPE k)
{
  ETYPE ergebnis; // Hilfsvariable fuer das Ergebnis
  vector r(m);    // Hilfsvektor

  /*
     Wir erhoehen den Wert der Variablen funkt um 1,
     damit wir in der Statistik eine korrekte Anzahl
     von Funktionsauswertungen erhalten.
     */

  ++funkt;

  r      = A*x-b;

  if (quadrat)    // Wenn ein quadratisches Programm behandelt wird,
                  // berechne    1  T       T
                  //             - x Q x + c x
                  //             2
    ergebnis = 0.5 * sp(x, Q*x) + sp(c,x);
  else
                  // Ansonsten nur     T
                  //                  c x
                  //
    ergebnis = sp(c,x);
```

```
  /*
     In der folgenden Schleife werden Strafterme abgezogen
     */

  for (int i=1; i<=m; i++)
    ergebnis -= (lambda(i) / k) * log(k*r(i)+1.0);

  /*
     Nun wird nur noch das Ergebnis zurueckgegeben
     */

  return ergebnis;
}

/*

 *******************************************************************
 *                                                                 *
 *  Funktion: vector dF(matrix& Q, vector&c,                       *
 *                      matrix& A, vector& b,                      *
 *                      vector&x, vector&lambda, ETYPE k)          *
 *                                                                 *
 *  Aufgabe : Berechnet den Gradienten der Modifizierten           *
 *            Barrierefunktion                                     *
 *                                   T                             *
 *            dF(x,y,k) = Q x + c - A r,                           *
 *                                                                 *
 *            wobei                                                *
 *                       lambda(i)                                 *
 *            r(i) =   --------------                              *
 *                       T                                         *
 *                     k(a x - b ) +1                              *
 *                        i     i                                  *
 *                                                                 *
 *                                                                 *
 *  Ergebnis: Vom Typ vector, Gradient von F(x,lambda,k)           *
 *                                                                 *
 *******************************************************************

 */

vector dF(matrix& Q, vector&c,
          matrix& A, vector& b,
          vector& x, vector& lambda,
          ETYPE k)
{
  vector r(m);
  vector ergebnis(n);
```

```
  vector aux(n);

  /*
     Fuer die Statistik: Inkrementiere die Variable grad
     */

  ++grad;

  r     = A*x-b;

  for (int i=1; i<=m; i++)
    r(i) = lambda(i)/(k*r(i)+1.0);

  if (quadrat)
    ergebnis = Q*x + c;
  else
    ergebnis = c;

  aux = A.T()*r;

  return ergebnis-aux;

}

/*

 ******************************************************************
 *                                                                *
 *  Funktion: matrix HF(matrix& Q,                                *
 *                      matrix& A, vector& b,                     *
 *                      vector&x, vector& lambda, ETYPE k)        *
 *                                                                *
 *  Aufgabe : Berechnet die Hessematrix der Modifizierten         *
 *            Barrierefunktion                                    *
 *                                   T                            *
 *            HF(x,y,k) = Q - k A D Y D A,                        *
 *                                                                *
 *            wobei                                               *
 *                                                                *
 *            Y = diag (lambda(1), ... , lambda(m))               *
 *                                                                *
 *            D = diag (d(1), ... , d(m))                         *
 *                                                                *
 *                               1                                *
 *            d(i) =   --------------                             *
 *                            T                                   *
 *                      k(a x - b ) +1                            *
 *                         i     i                                *
```

```
 *                                                                   *
 *                                                                   *
 *  Ergebnis: Vom Typ matrix, Hessematrix von F(x,lambda,k)          *
 *                                                                   *
 *********************************************************************

 */

matrix HF(matrix& Q,
          matrix& A, vector& b,
          vector& x, vector& lambda,
          ETYPE k)
{
  vector r(m);
  matrix Z(m,n);

  /*
     Fuer die Statistik: Inkrementiere die Variable hesse
     */

  ++hesse;

  r      = A*x-b;

  for (int i=1; i<=m; i++)
     for (int j=1; j<=n; j++)
       Z(i,j) = k*A(i,j)*lambda(i)/((k*r(i)+1.0)*(k*r(i)+1.0));

  if (quadrat)
    return (Q + A.T()*Z);
  else
    return A.T()*Z;
}

/*

 *********************************************************************
 *                                                                   *
 *  Funktion: void falsche_option (char *s)                          *
 *                                                                   *
 *  Aufgabe : Wird aufgerufen, wenn dem Programm eine falsche        *
 *            Kommandozeilenoption uebergeben wurde.                 *
 *            S ist ein Zeiger auf die Zeichenkette, welche diese    *
 *            Option enthaelt. Es wird eine Fehlermeldung aus-       *
 *            gegeben, die u.a. die falsche Option enthaelt und      *
 *            dann das laufende Programm abgebrochen.                *
 *                                                                   *
 *  Ergebnis: Vom Typ vector, Gradient von F(x,y,k)                  *
```

```
 *                                                                  *
 ********************************************************************

 */

void falsche_option (char *s)
{
  cerr << "modbfm: FEHLER     : Falscher ";
  cerr << "Kommandozeilenparameter:";
  cerr << s << endl;
  cerr << "modbfm: Das Programm wird abgebrochen!" << endl;
  cerr << endl;

  exit (-1);
}

/*

 ********************************************************************
 *                                                                  *
 *  Funktion: void tabelle (ETYPE pfkt, ETYPE prim_inf,             *
 *                          ETYPE dfkt, ETYPE dual_inf,             *
 *                          ETYPE kompl)                            *
 *                                                                  *
 *  Aufgabe : Druckt eine Tabelle mit Informationen ueber den       *
 *            laufenden Stand der Iteration aus. Dabei bedeuten     *
 *            pfkt     : Primaerer Funktionswert                    *
 *            prim_inf : Primaere Unzulaessigkeit                   *
 *            dfkt     : Dualer Funktionswert                       *
 *            dual_inf : Duale Unzulaessigkeit                      *
 *            compl    : Komplementaritaet                          *
 *                                                                  *
 *  Ergebnis: Keines                                                *
 *                                                                  *
 ********************************************************************

 */


void tabelle (ETYPE pfkt, ETYPE prim_unz, ETYPE dfkt, ETYPE dual_unz,
              ETYPE kompl )
{

  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n");
  fprintf (stdout, "  |                        | ");
  fprintf (stdout, "%-20.12s   | ", "Primaer      ");
```

```
  fprintf (stdout, "%-20.12s   |\n", "Dual         ");
  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n");
  fprintf (stdout, "  | Unzulaessigkeit        | ");
  fprintf (stdout, "%20.12g   | ", prim_unz);
  fprintf (stdout, "%20.12g   |\n", dual_unz);
  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n");
  fprintf (stdout, "  | Funktionswert          | ");
  fprintf (stdout, "%20.12g   | ", pfkt);
  fprintf (stdout, "%20.12g   |\n", dfkt);
  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n");
  fprintf (stdout, "  | Komplementaritaet      | ");
  fprintf (stdout, "%30.12g", kompl);
  fprintf (stdout, "     %10s   |\n", " ");
  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n");
  fprintf (stdout, "  | Rel. Dualitaetsluecke  | ");
  ETYPE rel_dual_gap;
  rel_dual_gap = fabs(pfkt - dfkt) / (max(1.0, fabs(pfkt)));
  fprintf (stdout, "%30.12g", rel_dual_gap);
  fprintf (stdout, "     %10s   |\n", " ");
  fprintf (stdout, "  ---------------------------");
  fprintf (stdout, "------------------------");
  fprintf (stdout, "------------------------\n\n");
}



/* ---------------------------------------------------------------- */



/*

 *********************************************************************
 *                                                                   *
 *  Funktion: main (int argc, char **argv)                           *
 *                                                                   *
 *  Aufgabe : Hauptteil des Programms.                               *
 *                                                                   *
 *  Ergebnis: Keines                                                 *
 *                                                                   *
```

```
 ******************************************************************

 */

main (int argc, char **argv)
{
  int i,j;                   // Schleifenvariablen
  char *konfig;              // Zeiger auf den Namen der
                             // Konfigurationsdatei
  char konf_standard[15];    // Zeichenkette mit dem Standardnamen
                             // modbfm.rc fuer die
                             // Konfigurationsdatei

  strcpy (konf_standard, "specfile.mbf"); // Standardnamen intialisieren

  /*
     Als erstes geben wir eine Meldung aus.
     */

  cout << endl << "modbfm v1.0 von Sven O. Krumke" << endl << endl;

  /*
     Zunaechst werden die Parameter des Verfahrens mit Standardwerten
     belegt.
     */

  k_start         = 100;     // Start-Barriereparameter

  k_max           = 10000;   // Maximaler Barriereparameter

  eps_newton      = 2e-5;    // Genauigkeit fuer die Newton-Iteration

  b_methode       = KNST;    // Standard: Konstanter Barriereparameter

  gamma_mbf       = 1.0;     // Newton-Genauigkeit konstant lassen

  k_inkrement     = 1.0;     // Bei Erhoehung des Barriereparameters
  k_faktor        = 2;       // (nur VAR oder UNZ ohne Einschr.)
                             // berechne den neuen Parameter als 2*k+1

  t_komplement    = 1e-7;    // Abbruchkriterien: Komplementaritaet
  t_prim_unz      = 1e-7,    //                   primaere Unzulaessigk.
  t_dual_unz      = 1e-7;    //                   duale Unzulaessigkeit

  prog_standard   = AUTO;    // Standard: Dateiendung .lp  => LP
                             //           Dateiendung .qp  => QP

  M               = 100.0;   // Standardparameter fuer die Transformation
```

```
                          // von (QP) auf (QP)^(M)

/*
   Nun definieren und initialisieren wir die Variablen, die
   durch Kommandozeilenoptionen gesetzt bzw. veraendert
   werden koennen
   */

int startvektor_auto = 1;     // Standard: Automatische Wahl
                              //           des Startvektors

int verbose          = 0;     // Standard: Kurze Meldungen

konfig = konf_standard;       // Konfigurationsdatei ist
                              // standardmaessig modbfm.rc

int ignoriere_konf   = 0;     // Standard: Werte die
                              // Konfigurationsdatei aus

int quiet            = 0;     // Standard: Ausgabe wesentlicher
                              // Zwischenergebnisse

/*
   Als naechstes werden die Kommandozeilenparameter, d.h.
   die Parameter, die dem Programm uebergeben wurden, ausgewertet.
   */

for (i=1; i<argc; i++) {        // Fuer alle Kommandozeilenopt. ...

  if (argv[i][0] != '-')        // Optionen muessen mit einem
    break;                      // '-' beginnen

  if ( strlen(argv[i]) > 2 ) {  // Es sind nur Optionen aus einem
                                // aus einem Zeichen zulaessig.
    falsche_option (argv[i]);   // Bei falscher Option => Fehler
  }
  else {
    /*
       Gelangt das Programm bis hierhin, so ist eine Option aus
       einem Zeichen mit vorangestelltem '-' erkannt worden.
       Wir testen nun, ob die Option zulaessig ist.
       Falls nicht, wird eine Fehlermeldung ausgegeben und das
       Programm abgebrochen. Ansonsten wird die Option ausgewertet.
       */

    switch (argv[i][1]) {

    case 'h' :
```

```
case '?' :
  cout << endl;
  cout << "                            MODBFM V1.0" << endl;
  cout << "                            ------------";
  cout << endl;
  cout << "Programm aus der Diplomarbeit von";
  cout << " Sven Oliver Krumke (Univ. Wuerzburg)" << endl;
  cout << "zur Loesung von linearen und ";
  cout << "konvexen quadratischen Programmen der Form " << endl;
  cout << endl;
  cout << "             1  T       T   " << endl;
  cout << "        min  - x Q x + c x  " << endl;
  cout << "             2              " << endl;
  cout << endl;
  cout << "         x : A x > b        " << endl;
  cout << "                 =          " << endl;
  cout << endl;
  cout << "mit Hilfe der Modifizierten Barrieremethode";
  cout << " (R. Polyak, 1992)." << endl;
  cout << endl;
  cout << "AUFRUF: modbfm [Optionen] Datenfile" << endl;
  cout << endl;
  cout << "Gueltige Optionen                       " << endl;
  cout << "-----------------" << endl;
  cout << "-h oder -?   : Dieser Hilfetext        " << endl;
  cout << "-l name      : Benutzen der Datei name anstelle von ";
  cout << "modbf.rc zur Konfiguration" << endl;
  cout << "-s           : Startvektor aus der Eingabedatei";
  cout << " lesen" << endl;
  cout << "-q           : Ignorieren der Konfigurationsdatei";
  cout << " modbfm.rc" << endl;
  cout << "-v           : Ausfuehrliche Ausgaben waehrend ";
  cout << "des Programmlaufs" << endl;
  cout << "-x           : Unterdrueckung aller Ausgaben bis auf ";
  cout << "das Endresultat" << endl;
  cout << endl;

  exit (0);
  break;

case 'l' :
  konfig = argv[i++];   // den Namen der Konfigurationsdatei
                        // holen und Zeiger auf die Optionen
                        // inkrementieren
  break;

case 's' :
  startvektor_auto = 0;
```

```
      break;

    case 'q' :
      ignoriere_konf   = 1;
      break;

    case 'v' :
      verbose = 1;
      break;

    case 'x' :
      verbose = 0;
      quiet   = 1;
      break;

      default :
      falsche_option (argv[i]);
      break;
    }
  }
}


/*
   Nun wird die Konfigurationsdatei modbfm.rc ausgewertet.
   Eintragungen in dieser Datei ueberschreiben die Standardwerte
   fuer die Parameter.
   */

if (ignoriere_konf) {    // Wenn die Konfigurationsdatei
                         // ignoriert werden soll.
  cout << "Die Konfigurationsdatei ";
  cout << konfig << " wird ignoriert";
  cout << endl << endl;
}
else {
  if (freopen (konfig, "r", stdin)) {
    /*
       Falls die Konfigurationsdatei erfolgreich zum Lesen
       geoeffnet werden konnte, wird die Standardeingabe
       auf die Datei "verbogen".
       */

    cout << "Auswerten der Konfigurationsdatei";
    cout << konfig;
    cout << " ... " << endl;

    line = 1;  // Zeilennummer fuer evtl. Fehlermeldungen
```

```
               // initialisieren

    yyparse(); // Die Konfigurationsdatei wird mit Hilfe des
               // Parsers ausgewertet

    cout << "Konfigurationsdatei erfolgreich ausgewertet";
    cout << endl << endl;

    fclose (stdin); // Ruecksetzen der Standardeingabe
  }
  else {
    /*
       Falls keine Konfigurationsdatei existiert oder sie nicht
       lesbar ist (z.B. der Benutzer keine Leserechte fuer die
       Datei besitzt) geben wir eine Meldung aus, die darueber
       informiert.
       */

    cout << "modbfm: INFORMATION: ";
    cout << "Keine oder nicht lesbare Konfigurationsdatei:";
    cout << konfig;
    cout << endl << endl;
  }
}

/*
   Jetzt wird der Name der Datei mit den Problemdaten
   aus der Kommandozeile gelesen
   */

if (i == argc) {       // Wenn kein Dateiname angegeben wurde,
                       // dann Abbruch mit Fehlermeldung
  cerr << "modbfm: FEHLER     : Es wurde kein Dateiname ";
  cout << "fuer die " << endl;
  cerr << "                     Datei mit den ";
  cerr << "Problemdaten angegeben" << endl;
  cerr << "modbfm: Das Programm wird abgebrochen!" << endl;
  cerr << endl;
  exit (-1);
}

switch (prog_standard) {

case LP :
  quadrat = 0;
  break;

case QP :
```

```
    quadrat = 1;
    break;

  case AUTO :
    int fehler = 0;
    if (strlen(argv[i]) < 3)
      fehler = 1;
    else {
      char *s = argv[i]+strlen(argv[i])-3;

      if (!strcmp(s, ".lp"))
        quadrat = 0;
      else
        if (!strcmp(s,".qp"))
          quadrat = 1;
        else
          fehler = 1;
    }

    if (fehler) {
      cerr << "modbfm: FEHLER     : Der Dateiname: ";
      cerr << argv[i] << " besitzt keine der Endungen " << endl;
      cerr << "                     .lp oder .qp, aber ";
      cerr << "\'standard_programm\'" << endl;
      cerr << "                     ist auf \'auto\' gesetzt";
      cerr << endl;
      cerr << "modbfm: Das Programm wird abgebrochen!" << endl;
      cerr << endl;
      exit (-1);
    }
  }

  ifstream input (argv[i]);   // Versuche, die Datei, deren Name
                              // in argv[i] steht, zum Lesen
                              // zu oeffnen

  if (! input )  {            // Wenn die Datei nicht geoeffnet
                              // werden kann, Fehler und Abbruch.
    cerr << "modbfm: FEHLER:      Die Eingabedatei: ";
    cerr << argv[i];
    cerr << " kann nicht zum Lesen" << endl;
    cerr << "                     geoeffnet werden";
    cerr << endl;
    cerr << "modbfm: Das Programm wird abgebrochen!" << endl;
    cerr << endl;
    exit (-1);
  }
```

```
/*
   Als naechstes werden die Problemdaten aus der Datei eingelesen.
   Es wird erwartet, dass die Daten in folgender Reihenfolge in der
   Datei stehen:

   Kommentarzeilen  (optional, beginnend mit '#'                  )
   n                (Anzahl der Variablen                         )
   m                (Anzahl der Nebenbedingungen                  )
   Q                (nur bei quadratischen Problemen              )
   c
   A                (Matrix der Nebenbedingungen                  )
   b                (rechte Seite der Nebenbedingungen            )
   x_bar            (Startvektor, nur bei Kommandozeilenoption -s
                     oder bei der UNZ-Variante                    )
   */

char zeichen;
int  ende;

ende = 0;

/* Zunaechst werden die Kommentarzeilen ueberlesen  */

while (!ende && input.get(zeichen))
  if (zeichen == '#') {
      while ( zeichen != EOF && zeichen != '\n' )
        input.get(zeichen);
    }
  else {
    input.putback (zeichen);
    ende = 1;
  }

input >> ws_komma >> n;
input >> ws_komma >> m;

/*
   Falls die Transformation auf (QP)^(M) durchgefuehrt werden
   soll, so sind n und m zu inkrementieren.
   */

int n_t = n;
int m_t = m;

if (b_methode != UNZ) {
  ++n_t;
  ++m_t;
}
```

```
/*
   Wir definieren nun die Variablen, die die Problemdaten
   aufnehmen sollen.
   */

matrix Q(n_t,n_t);      // Matrix der quadr. Form
vector c(n_t);          // Vektor in der Zielfunktion
matrix A(m_t,n_t);      // Nebenbedingungsmatrix
vector b(m_t);          // rechte Seite der Nebenbed.
vector x_bar(n_t);      // Startvektor

/*
   Wir definieren nun die restlichen Vektor- und Matrizen-
   variablen, die im Laufe der Iteration verwendet werden.
   */

vector x(n_t);          // Laufender primaerer Iterationsvektor
vector x_best(n_t);     // Bisher bester priaemere Iterationsvektor
vector lambda(m_t);     // Laufender dualer Iterationsvektor
vector lambda_neu(m_t); // Tentativer neuer dualer Iterationsvektor

matrix L(n_t, n_t);
vector r(m_t), r_zeta(m_t), r_u(m_t), r_bar(m_t);
vector zeta(n_t);
vector dd(n_t);
vector u(n_t);

 /*
    Lesen der Problemdaten. Hier ist LEIDER die Benutzung des >>
    Operators aus der Matrix-Bibliothek nicht moeglich, da u.U.
    nur ein Teil der Matrix gelesen werden soll (naemlich dann,
    wenn die Transformation auf (QP)^(M) gemacht wird).
*/

if (quadrat)
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      input >> ws_komma >> no_eof >> Q(i,j);


for (i=1; i<=n; i++)
  input >> ws_komma >> no_eof >> c(i);

for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    input >> ws_komma >> no_eof >> A(i,j);
```

```
for (i=1; i<=m; i++)
  input >> ws_komma >> no_eof >> b(i);

/*
   Wir pruefen noch, ob der Startvektor aus der Datei gelesen
   werden soll. Dies ist der Fall, wenn die Option -s beim
   Start des Programms angegeben wurde oder die Modifizierte
   Barrieremethode mit einem unzulaessigen Vektor gestartet
   werden soll. Bei Bedarf wird der Startvektor aus der
   Datei gelesen.
   */

if (!startvektor_auto || b_methode == UNZ)
  for (i=1; i<=n; i++) {
    input >> ws_komma >> no_eof >> x_bar(i);
  }
else {

  /*
     Falls kein Startvektor gelesen werden wurde, so
     definieren wir einen solchen durch x_bar = e
     */

  x_bar  = 1.0;
}

/*
   Falls die Transformation auf (QP)^(M) vollzogen werden soll,
   dann wird nun die (n+1) Komponente von x_bar so berechnet,
   dass x_bar ein passender Startvektor fuer die erste Newton-
   Iteration ist. Ferner werden die zusaetzlichen Zeilen und
   Spalten in den Vektoren und Matrizen entsprechend (QP)^(M)
   ergaenzt.
   */

if (b_methode != UNZ) {

  r            =  A*x_bar-b;
  ETYPE maxmin = 0.0;

  /* Bestimme die Unzulaessigkeit maxmin >= 0 von x = e */

  for (i=1; i<=m; i++)
    if (r(i) < 0.0) {
      if ( maxmin < fabs(r(i)) )
        maxmin = fabs(r(i));
     }
```

```
    /*
       Bestimme x_bar(n+1) so, dass x_bar als Startpunkt fuer die
       erste Newton-Iteration verwendbar ist.
       */
    x_bar(n+1) = maxmin + 2.0 / k_start;

    /* Ergaenze die Vektoren und Matrizen */

    for (i=1; i<=n+1; i++) {
      Q(n+1,i) = 0.0;
      Q(i,n+1) = 0.0;
    }

    c(n+1) = M;

    for (i=1; i<=n; i++)
      A(m+1,i) = 0.0;

    for (j=1; j<=m; j++)
      A(j,n+1) = 1.0;

    A(m+1, n+1) = 1.0;

    b(m+1) = 0.0;

    /*
       Erhoehe n und m, so dass wir ab sofort n wieder als die
       Anzahl der Variablen und m als die Anzahl der Nebenbed.
       einheitlich verwenden koennen.
       */

    n = n_t;
    m = m_t;
  }

  else {
    /*
       Falls die Variante mit unzlaessigem Startpunkt gewaehlt
       wurde, muessen wir den Anfangsbarriereparameter k_start
       so waehlen, dass x_bar im Inneren von S_{k_start} liegt.
       */

    r = A*x_bar - b;

    ETYPE maxmin = 0.0;
    for (i=1; i<=m; i++)
      if (r(i) < 0)
        if (fabs(r(i)) > maxmin)
```

```
        maxmin = fabs(r(i));

  /*
     Wenn der Startvektor x_bar unzulaessig fuer das Original-
     problem war (dies wird fast immer so sein), dann
     berechnen wir k_start geeignet. Ansonsten waehlen wir
     k_start wie in der Konfigurationsdatei angegeben, bzw.
     wie per Voreinstellung gegeben
     */

  if (maxmin > 0.0)
    k_start = 1.0 / (2.0*maxmin+0.5);

  if (!quiet)
    cout << "  Start-Barriere: " << k_start << endl << endl;
}


/*
   Hier erfolgt nun die Definition und Initialisierung von
   skalaren Variablen, die entweder fuer Statistik- oder
   sonstige Hilfszwecke benoetigt werden.
   */

int  nu         = 0;       // Nr. der laufenden (auesseren) Iteration
int  newton     = 0;       // Gesamtanzahl von Newtonschritten
int  akt_newton = 0;       // Newtonschritte im akt. Teilproblem
int  updates    = 0;       // Anzahl von Multiplikator-Updates
int  erhoehung  = 0;       // Anzahl der Barrierep. - Erhoehungen
int  newton_1   = 0;       // Anzahl der Newton-Schritte im 1.
                           // Problem
int  newton_2   = 0;       // Anzahl der Newton-Schritte im 2.
                           // Problem
int  newton_e   = 0;       // Anzahl der Newton-Schritte im letzten
                           // Problem
ETYPE gap_1;               // Rel. Dualit"atsluecke nach 1. Update

ETYPE  k        = k_start; // Laufender Barriereparameter

int   l         = 1;       // Benoetigt bei der MBFM mit Anpassung
ETYPE t;                   // Line-Search Parameter
ETYPE t_max;

ETYPE fu, val, prd, pfkt, dfkt;
ETYPE test_funktionswert;
ETYPE funkt_best;
ETYPE max_verletzung;
```

```
funkt          = 0;
grad           = 0;
hesse          = 0;

ETYPE prim_unz;
ETYPE dual_unz;
ETYPE kompl;
ETYPE test_mass;

r = A*x_bar-b;

if (    ( b_methode == VAR && ! (r > (1.0/k)) )
    || ( b_methode != VAR && ! (r >-(1.0/k)) )  )
  {
    cerr << "Falscher Startvektor ... ENDE" << endl;
    cout << "Startvektor :" << x_bar.T() << endl;
    cout << "A*x_bar-b   :" << r.T() << endl;
    exit (-1);
  }

x      = x_bar;
x_best = x_bar;

if (quadrat)
  funkt_best = 0.5*sp(x,Q*x) * sp(c,x);
else
  funkt_best = sp(c,x);

lambda = 1.0;

/*
   Es beginnt die Iteration des Verfahrens. Gegeben sind
   x^(nu-1), lambda^nu und k^nu (fuer nu=0 ist x^(nu-1)
   durch x^(nu-1) := x_bar gegeben).
   */

while (1) {

  /*
     Wir testen zunaechst, ob die Abbruchkriterien erfuellt
     sind.
     */

  r        = A*x-b;
  prim_unz = 0.0;
  kompl    = 0.0;
  for (i=1; i<=m; i++) {
    kompl += lambda(i)*fabs(r(i));
```

```
    if (-r(i)  > prim_unz)
      prim_unz = -r(i);
  }

  dual_unz = max_norm (dF(Q,c,A,b,x,lambda,k));

  if (quadrat)
    fu = sp(x,Q*x);
  else
    fu = 0.0;

  pfkt =  0.5*fu+sp(c,x);
  dfkt = -0.5*fu+sp(b,lambda);

  if (nu == 1 && akt_newton == 0)
    gap_1 = fabs(pfkt - dfkt) / (max(1.0, fabs(pfkt)));

  /*
     Ausgabe von Informationen, falls die Option -v
     angegeben wurde, d.h. ausfuehrliche Ausgaben erwuenscht
     sind.
     */

  if (verbose) {
    fprintf (stdout, "  Aeussere Iteration Nr : ");
    fprintf (stdout, "%5d\n\n", nu);
    fprintf (stdout, "  Newtonschritte aktuell: ");
    fprintf (stdout, "%5d\n", akt_newton);
    fprintf (stdout, "  Newtonschritte insges.: ");
    fprintf (stdout, "%5d\n\n", newton);
    fprintf (stdout, "  Multiplikator Updates : ");
    fprintf (stdout, "%5d\n\n", updates);
    fprintf (stdout, "  Barriereparameter     : ");
    fprintf (stdout, "%g\n", k);
    fprintf (stdout, "  Parameter-Erhoehungen : ");
    fprintf (stdout, "%5d\n\n", erhoehung);

    tabelle (pfkt, prim_unz, dfkt,
             dual_unz, kompl);
  }

  if (prim_unz     <= t_prim_unz &&
      dual_unz     <= t_dual_unz &&
      kompl        <= t_komplement)     {

    fprintf (stdout, "\n\nProgrammende nach ");
    fprintf (stdout, "%d aeusseren Iterationen\n\n", nu);
```

```
fprintf (stdout, "%30s\n", "Statistik");
fprintf (stdout, "%30s\n\n\n", "---------");

fprintf (stdout, "          Problemtyp              : ");
if (quadrat)
  fprintf (stdout, "Quadratisches Programm\n\n");
else
  fprintf (stdout, "Lineares Programm\n\n");

fprintf (stdout, "          Mod. Barrieremethode    :  ");
switch (b_methode) {
case KNST:
  fprintf (stdout, "Konstanter Barriereparameter\n");
  fprintf (stdout, "          Barriereparameter       :  ");
  fprintf (stdout, "%8g\n", k_start);
  --n;
  --m;
  break;

case VAR:
  fprintf (stdout, "Variabler Barriereparameter\n");
  fprintf (stdout, "          Start-Barriereparameter :  ");
  fprintf (stdout, "%8g\n", k_start);
  fprintf (stdout, "          End-Barriereparameter   :  ");
  fprintf (stdout, "%8g\n", k);
  fprintf (stdout, "          Parameter-Erhoehungen   :  ");
  fprintf (stdout, "%8d\n", erhoehung);
  --n;
  --m;
  break;

case UNZ:
  fprintf (stdout, "Start mit unzulaessigem Startpunkt\n");
  fprintf (stdout, "          Start-Barriereparameter :  ");
  fprintf (stdout, "%8g\n", k_start);
  fprintf (stdout, "          End-Barriereparameter   :  ");
  fprintf (stdout, "%8g\n", k);
  fprintf (stdout, "          Parameter-Erhoehungen   :  ");
  fprintf (stdout, "%8d\n", erhoehung);
  break;
}

fprintf (stdout, "          Variablen               : ");
fprintf (stdout, "%5d\n", n);
fprintf (stdout, "          Nebenbedingungen        : ");
fprintf (stdout, "%5d\n\n", m);

fprintf (stdout, "          Aeussere Iterationen    : ");
```

```
fprintf (stdout, "%5d\n", nu);
fprintf (stdout, "          Multiplikator Updates   : ");
fprintf (stdout, "%5d\n\n", updates);
fprintf (stdout, "          Newtonschritte gesamt   : ");
fprintf (stdout, "%5d\n", newton);
if (nu >= 1) {
  fprintf (stdout, "          ohne 1. Problem         : ");
  fprintf (stdout, "%5d\n", newton-newton_1);
}

fprintf (stdout, "          Newtonschr. 1. Problem  : ");
fprintf (stdout, "%5d\n", newton_1);
if (nu >= 1) {
  fprintf (stdout, "          Newtonschr. 2. Problem  : ");
  fprintf (stdout, "%5d\n", newton_2);
}
fprintf (stdout, "          Newtonschr. letztes Pr. : ");
fprintf (stdout, "%5d\n", newton_e);
fprintf (stdout, "          Durchschn. Newtonschr.  : ");
fprintf (stdout, "%5g\n", double(newton) / double(nu+1));
if (nu >=1) {
  fprintf (stdout, "          ohne 1. Problem         : ");
  fprintf (stdout, "%5g\n",
           double(newton-newton_1)/double(nu));
  fprintf (stdout, "          Newton-Qualitaet        : ");


  ETYPE rel_dual_gap;

  rel_dual_gap = fabs(pfkt - dfkt) / (max(1.0, fabs(pfkt)));
  fprintf (stdout, "%5g\n\n",
           pow(rel_dual_gap/gap_1,
               1.0/double(newton-newton_1)));
  fprintf (stdout, "          Gap nach 1. Update      : ");
  fprintf (stdout, "%5g\n\n", gap_1);
}

fprintf (stdout, "          Funktionsauswertungen   : ");
fprintf (stdout, "%5d\n", funkt);
fprintf (stdout, "          Gradientenauswertungen  : ");
fprintf (stdout, "%5d\n", grad);
fprintf (stdout, "          Hessematrixberechnungen : ");
fprintf (stdout, "%5d\n\n", hesse);

if (quadrat)
  fu = sp(x, Q*x);
else
  fu = 0.0;
```

```
    tabelle (0.5*fu+sp(c,x), prim_unz, -0.5*fu+sp(b,lambda),
             dual_unz, kompl);

    cout << endl << endl;
    cout << "x_opt      = " << x.T() << endl;
    cout << "lambda_opt = " << lambda.T() << endl;
    exit (0);
  }

  newton_e = akt_newton;

  /*
     Wir finden nun die Newtonrichtung z im laufenden
     Punkt x durch Loesen des linearen Gleichungssystems:

       HF(Q,A,b,x,lambda,k) z = -dF(Q,A,c,b,x,lambda,k)
  */

  L     = HF(Q,A,b,x,lambda,k).Cholesky();
  dd    = -dF(Q,c,A,b,x,lambda,k);
  u     = LT_Solve(L, dd );
  zeta  = UT_Solve(L.T(), u);

  if (verbose) {
    ETYPE residuum;
    ETYPE norm_zeta;

    norm_zeta = norm(zeta);

    fprintf (stdout, ">>Newton-Schritt gefunden\n");
    fprintf (stdout, "  Norm                          : %20g\n",
             norm_zeta);

    residuum = norm (dd - L*L.T()*zeta);
    fprintf (stdout, "  Abs. Fehler bei der Berechnung: %20g\n",
             residuum);
    fprintf (stdout, "  Rel. Fehler bei der Berechnung: %20g\n",
             residuum / norm_zeta);
  }

  /*
     Da nun der Newton-Schritt bekannt ist, wird nun eine
     Line-Search mit dem von Polyak vorgeschlagenen
     Algorithmus (Armijo-Goldstein, Algorithmus (6.1.1) in
     der Diplomarbeit) durchgefuehrt.
     */
```

```
val     = F(Q,c,A,b,x,lambda,k);
prd     = - sp( dd, zeta ) / 3.0;
r_zeta  = A*zeta;

t   = 1.0;

do {
  u  = x+(t*zeta);
  r_u = r+(t*r_zeta);

  if ( r_u > -1.0/k ) {    // Falls u = x+t*zeta in int S_k liegt,

    /* Die naechste Abfrage prueft, ob das Line-Search
       Abbruch-Kriterium:
                                           1                  T
       F(x+t*zeta,lambda,k)-F(x,lambda,k)<= - *t*dF(x,lambda,k) zeta (*)
                                           3
       erfuellt ist.
       */

    if (F(Q,c,A,b,u,lambda,k) - val <= t * prd)
      break;
  }

  /* Falls entweder x+t*u nicht in int S_k liegt oder
     die Ungleichung (*) nicht erfuellt ist, dann wird
     t halbiert.
     */

  t *= 0.5;

} while (1);

/*
   Falls ausfuehrliche Ausgaben per Option -v eingestellt
   sind, dann geben wir den resultierenden Line-Search
   Parameter t aus.
   */

if (verbose) {
  fprintf (stdout, "  Line-Search Parameter : ");
  fprintf (stdout, "%24.18g\n\n", t);
}

/*
   Ab hier gilt: (i)  u=x+t*zeta liegt in S_k
                 (ii) Die Ungleichung (*) ist erfuellt
                 */
```

```
x = u;

++akt_newton;   //  Erhoehe den Zaehler fuer die
                //  Newton-Schritte im aktuellen Problem,
++newton_e;     //  fuer die Newton-Schritte im letzten Problem
++newton;       //  und fuer die Newton-Schritte insgesamt

if ( t == 1.0  && norm(zeta) <= eps_newton )  {

  /*
     Jetzt erfolgt die Unterscheidung nach den drei Varianten
     der Modifizierten Barrieremethode
     */

  switch (b_methode) {

  case KNST:    // Konstanter Barriereparameter
    /*
       Fuehre den Multiplikator-Update durch
       */

    for (i=1; i<=m; i++)
      lambda(i) = lambda(i)/(k*r_u(i)+1.0);

    if (verbose)
      cout << ">>Multiplikator-Update durchgefuehrt" << endl;

    ++updates;
    eps_newton = eps_newton * gamma_mbf;
    if (nu == 0)
      newton_1 = akt_newton;
    if (nu == 1)
      newton_2 = akt_newton;
    akt_newton = 0;
    ++nu;

    break;

  case VAR:     // Modifizierte Barrieremethode mit Anpassung
    if (quadrat)
      test_funktionswert = 0.5*sp(x,Q*x) + sp(c,x);
    else
      test_funktionswert = sp(c,x);

    if (test_funktionswert < funkt_best) {
      x_best = x;
      funkt_best = test_funktionswert;
```

```
}

/*
 Berechne, wie der neue Multiplikator y_neu aussehen
 wuerde.
 */

for (i=1; i<=m; i++)
  lambda_neu(i) = lambda(i)/(k*r_u(i)+1.0);

/*
 Berechnung der dualen Unzulaessigkeit.
 */

dual_unz = max_norm (dF(Q,c,A,b,x,lambda_neu,k));

/*
   Berechnung der primaeren Unzulaessigkeit und
   der Komplementaritaet
   */

prim_unz = 0.0;
kompl    = 0.0;

for (i=1; i<=m; i++) {
  if (-r_u(i)  > prim_unz)
    prim_unz = -r_u(i);
  kompl += fabs(r_u(i))*lambda_neu(i);
}

/*
   Sind ausfuehrliche Ausgaben gewuenscht, so geben
   wir die berechneten Werte aus.
   */

if (verbose) {
  fprintf (stdout, ">>Test der Variablen Methode\n");
  fprintf (stdout, "  Primaere Unzulaessigkeit: ");
  fprintf (stdout, "%20.12g\n", prim_unz);
  fprintf (stdout, "  Duale    Unzulaessigkeit: ");
  fprintf (stdout, "%20.12g\n", dual_unz);
  fprintf (stdout, "  Komplementaritaet       : ");
  fprintf (stdout, "%20.12g\n\n", kompl);
}

test_mass = (dual_unz > prim_unz) ? dual_unz : prim_unz;
if (kompl > test_mass)
  test_mass = kompl;
```

```
    if (test_mass <= pow(gamma_mbf,l+1)) {

      if (verbose)
        cout << " >>Multiplikator Update wird durchgefuehrt" << endl;

      /*
         Wenn das Testmass genuegend klein ist,
         dann ...
         */

      lambda = lambda_neu;     // Mult. Update
      ++updates;
      ++l;
      eps_newton = eps_newton * gamma_mbf;
      if (nu == 0)
        newton_1 = akt_newton;
      if (nu == 1)
        newton_2 = akt_newton;
      akt_newton = 0;
      ++nu;
    }
    else {

      t_max = 1.0;

      r_bar = A*x_bar  - b;
      r_u   = A*(x_best-x_bar);

      for (i=1; i<=m; i++)
        if (r_u(i) < 0.0) {
           t = (1.0/k - r_bar(i))/r_u(i);
           if (t < t_max)
             t_max = t;
         }

      if (verbose)
        cout << "t_max = " << t_max << endl;

      x          = x_bar + t_max*(x_best-x_bar);
      lambda     = 1.0;
      l          = 1;
      eps_newton = eps_newton / k;
      if (nu == 0)
        newton_1 = akt_newton;
      if (nu == 1)
        newton_2 = akt_newton;
      akt_newton = 0;
```

```
      ++nu;

      if (!quiet) {
        cout << ">>Barriereparameter wird erhoeht: " << endl;
        cout << "  Alter Parameter : " << k << endl;
      }

      if (k_max < 0.0 || k*k_faktor + k_inkrement <= k_max) {
        k          = k*k_faktor + k_inkrement;
        ++erhoehung;
      }
      else {
        if (k != k_max)
          ++erhoehung;
        k          = k_max;
      }

      if (!quiet) {
        cout << "  Neuer Parameter : " << k << endl;
        cout << endl << endl;
      }
    }

    break;

  case UNZ:
    max_verletzung = 0.0;
    r_u = A*x - b;

    for (i=1; i<=m ; i++)
      lambda(i) = lambda(i) / (k*r_u(i) + 1.0);

    ++updates;

    for (i=1; i<=m; i++) {
      if (-r_u(i) > max_verletzung)
        max_verletzung = -r_u(i);
    }

    if (max_verletzung == 0.0) {
      if (!quiet) {
        cout << ">>Erhoehe den Parameter ohne Beschraenkung" << endl;
        cout << "  Alter Parameter : " << k << endl;
      }

      ETYPE k_min;

      k_min = k*k_faktor + k_inkrement;
```

```
    if (k_max <= 0.0) {
      k = k_min;
      ++erhoehung;
    }
    else {
      if (k_min > k_max)
        k_min = k_max;
      if (k != k_min) {
        ++erhoehung;
        k = k_min;
      }
    }

    if (!quiet) {
      cout << "  Neuer Parameter : " << k << endl;
      cout << endl << endl;
    }
  }
  else
    {
      if (!quiet) {
        cout << ">>Erhoehe den Parameter MIT Beschraenkung" << endl;
        cout << "  Alter Parameter : " << k << endl;
      }

      ETYPE k_min;

      k_min = 0.5 * (k+1.0 / max_verletzung);
      if (k_min > k * k_faktor + k_inkrement)
        k_min = k * k_faktor + k_inkrement;

      if (k_max <= 0.0) {
        k = k_min;
        ++erhoehung;
      }
      else {
        if (k_min > k_max)
          k_min = k_max;
        if ( k != k_min) {
          k = k_min;
          ++erhoehung;
        }
      }

      if (!quiet) {
        cout << "  Neuer Parameter : " << k << endl;
        cout << endl << endl;
```

```
            }
          }

        eps_newton = eps_newton * gamma_mbf;
        if (nu ==0)
          newton_1 = akt_newton;
        if (nu ==1)
          newton_2 = akt_newton;
        akt_newton = 0;
        ++nu;
        break;

      }
      if (quadrat)
        test_funktionswert = 0.5*sp(x,Q*x) + sp(c,x);
      else
        test_funktionswert = sp(c,x);

      if (test_funktionswert < funkt_best) {
        x_best     = x;
        funkt_best = test_funktionswert;

        if (verbose)  {
          cout << ">>Neuer Vektor x_best" << endl;
          cout<< x.T() << endl << endl;
        }
      }
    }
  }     /* Ende der Iteration */
}
```

Anhang C

Symbole und Notationen

Vektoren und Matrizen

e	Vektor aus lauter Einsen: $e = (1, \ldots, 1)^T$
$I^{r,r}$	$r \times r$ Einheitsmatrix
$0^{r,p}$	Nullmatrix mit r Zeilen und p Spalten
$\mathrm{diag}(x_1, \ldots, x_n)$	$n \times n$–Diagonalmatrix mit den Einträgen $x_1, \ldots, x_n$ auf der Hauptdiagonalen
$\Delta(x, k)$	Spezielle Diagonalmatrix: $\Delta(x,k) = \mathrm{diag}(k(a_1^T x - b_1) + 1, \ldots, k(a_m^T x - b_m) + 1)$

Zu einem Vektor bezeichnet der entsprechende Großbuchstabe die Diagonalmatrix aus den Einträgen des Vektors: Ist $x \in \mathbb{R}^n$ ein Vektor so sei $X = \mathrm{diag}(x_1, \ldots, x_n)$.

Für einen Vektor $\lambda = (\lambda_1, \ldots, \lambda_m)^T \in \mathbb{R}^m$ steht $\lambda_{(p)}$ für die Projektion von λ auf die ersten p Komponenten: $\lambda_{(p)} = (\lambda_1, \ldots, \lambda_p)^T$. Weiterhin bezeichnet $\lambda_{(m-p)}$ die Projektion von λ auf die $m - p$ letzten Koordinaten.

Vektor– und Matrixnormen

$\|.\|_1$	ℓ_1–Norm eines Vektors: $\|x\|_1 = \sum\limits_j \|x_j\|$
$\|.\|_2$	Euklidische Norm eines Vektors: $\|x\|_2 = \sqrt{x^T x}$
$\|.\|_\infty$	Maximumsnorm eines Vektors: $\|x\|_2 = \max\limits_j \|x_j\|$
$\mathrm{lub}_2(B)$	Natürliche Matrixnorm von B zur Vektornorm $\|.\|_2$: $\mathrm{lub}(B)_2 = \sup\limits_{x \neq 0} \frac{\|Bx\|_2}{\|x\|_2}$
$\mathrm{lub}_\infty(B)$	Natürliche Matrixnorm von B zur Vektornorm $\|.\|_\infty$: $\mathrm{lub}_\infty(B) = \sup\limits_{x \neq 0} \frac{\|Bx\|_\infty}{\|x\|_\infty}$

Mengen

$\mathbb{R}^n_+$	$\{x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n : x_j \geq 0, j = 1, \ldots, n\}$
$\mathbb{R}^n_{++}$	$\{x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n : x_j > 0, j = 1, \ldots, n\}$
$\bar{B}(\hat{x}, \delta)$	Abgeschlossene Kugel um $\hat{x}$ mit Radius δ: $\bar{B}(\hat{x}, \delta) = \{ x : \|x - \hat{x}\|_\infty \leq \delta\}$
$D(\lambda^*, k_0, \delta, \varepsilon)$	siehe Seite 31 und 34
lev_c	Niveaumenge der Modifizierten Barrierefunktion $\mathcal{F}$: $\mathrm{lev}_c := \{x \in S_k : \mathcal{F}(x, \lambda, k) \leq c\}$ (siehe Seite 27)
$\partial f(y)$	Subdifferential der (konvexen oder konkaven) Funktion f an der Stelle y (siehe Seite 73)

Funktionen

q	Quadratische Zielfunktion des konvexen quadratischen Programms (QP): $q(x) = \frac{1}{2}x^TQx + c^Tx$
η	Zielfunktion des zu (QP) dualen quadratischen Programms (QPD): $\eta(x, \lambda) = -\frac{1}{2}x^TQx + b^T\lambda$ (siehe auch Seite 9)
L	Klassische Lagrangefunktion zu (QP) (siehe Seite 8)
$L_{(P)}$	Klassische Lagrangefunktion zu (P) (siehe Seite 21 und 71)
log	Natürlicher Logarithmus
$\hat{\delta}$	Entropie–Funktion, Entropie–Distanz: $\hat{\delta}(s, t) = s \log\left(\frac{s}{t}\right) - s + t$ (siehe Seite 77)
F	Frisch–Funktion, klassische logarithmische Barrierefunktion (siehe Seite 11)
Φ	Erweiterte Lagrangefunktion (siehe Seite 15)
$\mathcal{D}$	Bregman–Distanz: $\mathcal{D}(\lambda, \mu) = \sum_{j=1}^{m} \hat{\delta}(\lambda_j, \mu_j)$ (siehe Seite 79)
$\mathcal{E}$	Exponential–Straffunktion (siehe Seite 16)
$\mathcal{F}$	Modifizierte Barrierefunktion (vgl. Seite 19)
$\mathcal{M}$	Barrierefunktion mit Shift: $\mathcal{M}(., k) = \mathcal{F}(., e, k)$ (siehe Seite 45)
d	Zielfunktion des (Lagrange–) dualen Programms (D) zu (QP): $d(\lambda) = \inf_{x \in \mathbb{R}^n} L(x, \lambda)$ (siehe Seite 69)
$d_{(P)}$	Zielfunktion des (Lagrange–) dualen Programms (DP) zu (P): $d(\lambda) = \inf_{x \in \mathbb{R}^n} L_{(P)}(x, \lambda)$ (siehe Seite 71)
dist_1	Abstand zwischen einem Punkt x und einer Menge M: $\mathrm{dist}_1(x, M) = \inf_{u \in M} \|x - u\|_1$

Ableitungen von Funktionen

$\nabla_x f(x')$	Gradient von f bezüglich des Vektors x an der Stelle x'. Der Gradient ist ein Spaltenvektor mit j-tem Element $\frac{\partial f(x')}{\partial x_j}$.
$D_{z,w} f(z', w', x')$	Ableitung der Funktion $f(z, w, x) = (f_1(z, w, x), \ldots, f_r(z, w, x))^T$ an der Stelle z', w', x' bezüglich der (Vektor–) Variablen $z \in \mathbb{R}^d, w \in \mathbb{R}^\ell$. $D_{z,w} f(z', w', x')$ ist eine Matrix (b_{ij}) mit $$b_{ij} = \begin{cases} \frac{\partial f_i}{\partial z_j}(z', w', x') & , \quad 1 \leq j \leq d \\ \frac{\partial f_i}{\partial w_{j-d}}(z', w', x') & , \quad d+1 \leq j \leq d+\ell. \end{cases}$$
$\nabla^2_{xx} f(x')$	Hessematrix von f bezüglich des Vektors x an der Stelle x'. Der Eintrag (j, k) in $\nabla_{xx} f(x')$ ist $\frac{\partial^2 f(x')}{\partial x_j x_k}$.

Literaturverzeichnis

[1] I. Adler, N. K. Karmarkar, M. G. C. Resende, G. Veiga, „An implementation of Karmarkar's algorithm for linear programming“, Mathematical Programming 44 (1989), 297–335

[2] D. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods* (Academic Press, New York, 1982)

[3] G. B. Dantzig, *Linear Programming and Extensions* (Princeton University Press, Princeton, NJ, 1963)

[4] S. C. Dewhurst, K. T. Stark, *Programming in C++* (Prentice Hall, Englewood Cliffs, NJ, 1989)

[5] M. A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual* (Addison–Wesley, NJ, 1990)

[6] I. Ekeland, R. Temam, *Convex Analysis and Variational Problems* (North–Holland Publishing Company, Amsterdam, 1976)

[7] A. V. Fiacco and G. P. McCormick, *Nonlinear Programming: Sequential Uncomstrained Minimization Techniques* (Wiley, New York, 1968)

[8] R. Fletcher, *Practical Methods of Optimization* (Second Edition, Wiley, 1991)

[9] K. R. Frisch, „The logarithmic potential method of convex programming“, Memorandum, University Institute of Economics, Oslo (Oslo, 1955)

[10] A. George, J. Liu, *Computer solution of large sparse positive definite systems* (Prentice Hall, Englewood Cliffs, NJ, 1981)

[11] P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin and M. H. Wright, „On projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method“, Mathematical Programming 36 (1986), 183–209

[12] C. C. Gonzaga, „Conical projection algorithms for linear programming“, Mathematical Programming 43 (1989), 151–173

[13] C. C. Gonzaga, „Large step path–following methods for linear programming, Part 1: Barrier function method“, SIAM J. Optimiziation, Vol. 1, 268–279

[14] R. Horst, *Nichtlineare Optimierung* (Carl Hanser Verlag, München Wien, 1979)

[15] M. Iri and H. Imai, „A Multiplicative Barrier Function Method for Linear Programming“, Algorithmica 1 (1986), 455–482

[16] R. H. F. Jackson, P. T. Boggs, S. G. Nash, S. Powell, „Guidelines for reporting results of computational experiments. Report of the ad hoc committee“, Mathematical Programming 49 (1991), 413–425

[17] F. Jarre, „On the convergence of the method of analytic centers when applied to convex quadratic programs“, Mathematical Programming 49 (1991), 341–358

[18] F. Jarre, *The method of analytic centers for smooth convex programs*, Dissertation zur Erlangung des naturwissenschaftlichen Doktorgrades der Universität Würzburg, 1989

[19] D. L. Jensen, R. Polyak, R. Schneur, „Numerical experience with modified barrier functions for linear programming“, Technical Report, IBM T.J. Watson Research Center, New York, 1992

[20] N. A. Karmarkar, „A new polynomial–time algorithm for Linear Programming“, Combinatorica 4 (1984), 373–395

[21] I. J. Lustig, R. E. Marsten, D. F. Shanno, „Interior point methods for linear programming: Computational state of the art“, Technical Report SOR 92–17, Princeton University, NY, 1992

[22] V. Klee, G. J. Minty, „How good is the simplex algorithm?“, O. Sisha, ed. Inequalities III (Academic Press, New York, 1972)

[23] W. Murray, „Analytical expressions for eigenvalues and eigenvectors of the Hessian matrices of barrier and penalty functions“, Journal of Optimiziation Theory and Applications 7 (1971), 181–196

[24] K. G. Murty, *Linear complementary, linear and nonlinear programming* (Heldermann Verlag, Berlin, 1988)

[25] D. G. Luenberger, *Linear and Nonlinear Programming* (Addison–Wesley, Reading, MA, 1984)

[26] M. R. Osborne, „Dual Barrier Functions with superfast rates of convergence for the Linear Programming Problem“, J. Austral. Math. Soc. Ser. B 29 (1987), 39–58

[27] R. Polyak, „Modified Barrier Functions (Theory and Methods)“, Mathematical Programming 54 (1992), 177–222

[28] R. Polyak, „Modified Barrier Functions In Linear Programming“, IBM Research Report RC 177790, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY

[29] M. J. D. Powell, „Some convergence properties of the shifted log barrier method for linear programming“, Numerical Analysis Reports, University of Cambridge, England, November 1992

[30] J. Renegar, M. Shub, „Unified complexity analysis for Newton LP methods“, Mathematical Programming 53 (1992), 1–16

[31] R. T. Rockafellar, *Convex Analysis* (Princeton University Press, Princeton, NJ, 1970)

[32] R. T. Rockafellar, „A dual approach to solving nonlinear programming problems by unconstrained optimization“, Mathematical Programming 5 (1973), 354–373

[33] A. Schrijver, *Theory of linear and integer programming* (John Wiley and Sons, 1986)

[34] H. R. Schwarz, *Numerische Mathematik* (Teubner, Stuttgart, 1988)

[35] J. Stoer, *Numerische Mathematik 1* (Springer–Verlag, 1989)

[36] J. Stoer and C. Witzgall, *Convexity and Optimiziation in Finite Dimensions I* (Springer–Verlag, 1970)

[37] P. Tseng, D. P. Bertsekas, „On the convergence of the Exponential Multiplier Method for Convex Programming“, Mathematical Programming 60 (1993), 1–19

[38] M. H. Wright, „Determining subspace information from the Hessian of a barrier function“, Numerical Analysis Manuscript 92–02, AT&T Bell Laboratories, May 1992

Erklärung

Ich erkläre, daß ich die vorliegende Diplomarbeit selbständig erstellt und keine Hilfsmittel außer den im Literaturverzeichnis angegebenen verwendet habe.

Würzburg, den 11. Dezember 1995 Sven Oliver Krumke