Sichere Mehrparteienberechnungen und datenschutzfreundliche Klassifikation auf Basis horizontal partitionierter Datenbanken

Dissertation zur Erlangung des Grades eines Doktors der Wirtschaftswissenschaften

eingereicht an der

Fakultät für Wirtschaftswissenschaften
der Universität Regensburg

vorgelegt von
Dipl.-Math. Manuel Liedel

Berichterstatter

Prof. Dr. Peter Lory

Prof. Dr. Dieter Bartmann

Tag der Disputation: 12. Dezember 2012

Mein ganz besonderer Dank gilt Professor Dr. Peter Lory für die Möglichkeit mehr als 3 Jahre an kryptographischen Algorithmen forschen zu können und auftretende Probleme jederzeit besprechen zu können. Nicht minder dankbar bin ich meiner lieben Frau Kathrin für permanentes, geduldiges Korrekturlesen und das kontinuierliche Ausbessern von Tippfehlern zu allen Tages- und Nachtzeiten.

Inhaltsverzeichnis

Einleitung

Die zentrale Aufgabe einer jeden Bank ist die Vergabe von Krediten. Untrennbar einher geht damit die Unterteilung der Kreditantragsteller in kreditwürdige und -unwürdige Kunden. Irrtümer in der Kreditvergabe können für Banken existenzgefährdend sein, wie sich in den letzten Jahren bei vielen Banken, insbesondere in Irland, Spanien und den USA gezeigt hat, als einzelne Institute durch faule Kredite in die Insolvenz getrieben wurden und von Mitbewerbern oder dem Staat aufgefangen werden mussten. Techniken, mit deren Hilfe die Bonität eines potentiellen Kreditnehmers beurteilt werden kann, bezeichnet man als Credit-Scoring (Kreditwürdigkeitsbewertung). Dabei werden persönliche Eigenschaften des Antragstellers mit denen anderer Kunden verglichen, die ihre Kredite bedient bzw. nicht bedient haben. Mit Hilfe mathematischer und statistischer Verfahren kann eine Prognose darüber getroffen werden, ob und mit welcher Wahrscheinlichkeit ein Darlehen zurückgezahlt werden wird. Insbesondere für kleinere Kreditinstitute mit einem geringen Kreditvolumen oder für Kreditanträge von Firmen aus sehr kleinen Branchen, für die auch bei einer großen Bank möglicherweise nicht genug Vergleichsdaten von Wettbewerbern vorliegen, kann eine solche Beurteilung jedoch schwierig sein. Wäre es möglich, die *anderen* Banken vorliegenden Kredithistorien vergleichbarer Antragsteller für eine Beurteilung hinzuzuziehen und die Entscheidung über die Erteilung eines Kredits so auf eine breitere Basis zu stellen, könnte diese mit einer größeren Sicherheit getroffen werden: Der Anteil nicht bedienter Kredite würde sinken, genauso wie die Anzahl der nicht-gewährten Kredite an eigentlich solvente Antragsteller. Allerdings ist es einer Bank aus Datenschutzgründen nicht möglich, anderen Finanzinstituten Zugriff auf die Kredithistorien ihrer Kunden zu gewähren.

Dies ist nur ein Beispiel für das allgemeinere Problem der Klassifikation von Daten auf Basis horizontal partitionierten Datenbanken. Dabei soll zur Klassifikation eines Datensatzes dessen Ähnlichkeit mit Daten *verschiedener* Datenbanken verwendet werden. Unterliegen die Daten dabei dem Datenschutz, etwa weil sie persönliche Daten oder Geschäftsgeheimnisse beinhalten und wenn selbst die Weitergabe an eine vertrauenswürdige dritte Instanz nicht denkbar ist, ist eine solche datenbankübergreifende Klassifikation - wie bei dem eingangs erwähnten Problem der Kreditwürdigkeitsbeurteilung - auf herkömmliche Weise nicht möglich. Andere Anwendungsbeispiele sind die Klassifikation

von Patientendaten auf Basis von Daten verschiedener Krankenhäuser oder die genetische Klassifikation anhand verschiedener Datenbanken. Ziel dieser Arbeit ist, mit Hilfe von *Sicheren Mehrparteienberechnungen* eine Lösung für dieses Problem zu konstruieren, also ein Klassifikationswerkzeug zu entwickeln, das Daten *verschiedener* Datenbanken verwendet, ohne dass die Besitzer die Daten aus der Hand geben müssen. Konkret handelt es sich dabei um eine datenschutzfreundliche Implementierung von *Support Vector Machines*.

Dies stellt einen vollkommen neuen Anwendungsbereich für sichere Mehrparteienberechnungen dar: In den vergangenen Jahren haben diese in zunehmendem Maße Anwendungen gefunden, so bei Auktionen ohne Auktionator, dem datenschutzfreundlichen Abgleich biometrischer Merkmale mit Datenbanken oder - im Rahmen des SecureSCM-Projekts - bei der firmenübergreifenden (linearen) Optimierung von Zulieferungsketten. Letzteres ist vielleicht das beste Anwendungsbeispiel und das, das der vorliegenden Arbeit am ehesten verwandt ist: Eine Zulieferungskette funktioniert dann optimal, wenn die beteiligten Firmen ihre Produktion aufeinander abstimmen. Es stellt sich dabei jedoch ein ähnliches Problem wie beim finanzinstitutübergreifenden Credit-Scoring: Die dazu benötigten Informationen, wie Stückkosten oder Kapazitäten, sind so sensibel, dass sie unter keinen Umständen außerhalb der Firma bekannt werden dürfen. Im Rahmen des oben genannten Projekts gelang es ein lineares Modell einer Zulieferungskette mit Hilfe einer sicheren Implementierung des Simplex-Algorithmus zu optimieren: Jede Firma steuerte ihre relevanten Informationen, wie Kosten oder Kapazitäten, zum Modell bei, das dann in einem interaktiven Protokoll ausgewertet wurde. Keine der beteiligten Firmen musste sich dabei Sorgen machen, dass ihre Daten anderen Teilnehmern bekannt wurden. Als Ergebnis erhielten alle Teilnehmer die sie betreffenden Werte, die die Zulieferungskette optimierten. Auf diese Weise konnten Effizienzgewinne realisiert werden, die allen Beteiligten zugute kamen, die aber ohne SecureSCM, aufgrund des Schutzes von Geschäftsgeheimnissen, nicht verwirklicht hätten werden können. Anders als bei vielen anderen Anwendungen von sicheren Mehrparteienberechnungen, bei denen es primär um das Lösen von Datenschutzproblemen geht, handelt es sich hier - ähnlich wie bei dem in dieser Arbeit vorgestellten Werkzeug zum datenschutzfreundlichen Credit-Scoring - darum, die Produktivität verschiedener Firmen zu steigern. Die Ähnlichkeiten mit SecureSCM gehen aber noch weiter: Die dafür entwickelte sichere Implementierung der Fixpunktarithmetik ist ein wichtiger Baustein dieser Arbeit.

Allgemein ermöglichen es sichere Mehrparteienberechnungen einer Anzahl verschiedener Parteien (oder Spieler) eine Funktion auszuwerten, zu der jeder von ihnen einen Eingabewert besitzt, ohne dass die Eingabewerte der Teilnehmer den Mitspielern oder einer dritten Instanz mitgeteilt werden müssen. Der

Fokus hat sich dabei in den letzten 30 Jahren immer mehr verschoben von einer theoretischen Betrachtungsweise, die - zumindest am Anfang - stark darauf ausgerichtet war, die Robustheit sicherer Mehrparteienberechnungen in verschiedenen Sicherheits- und Angreiferszenarien zu überprüfen, hin zu den oben erwähnten (und anderen) praktischen Anwendungen.

Zu Beginn einer sicheren Mehrparteienberechnung „verschleiert" dabei jeder Teilnehmer jeden seiner Eingabewerte (Kapitel 1), indem er ihn in Teile (*Shares*) zerlegt, von denen jeder für sich genommen rein zufällig ist, und sendet diese den Mitspielern zu. Die Berechnung wird dann mit diesen Teilen - unter Zuhilfenahme interaktiver Protokolle - durchgeführt, ohne dass Zwischenergebnisse, die wieder aus *Shares* bestehen oder die Eingabewerte der Spieler bekannt werden. Erst zur Rekonstruktion des Ergebnisses werden die Shares aller Spieler benötigt.

Für das in dieser Arbeit vorgestellte datenschutzfreundliche Klassifikationswerkzeug ist das Rechnen mit Shares von nicht-ganzen Zahlen essentiell. Verschiedene Ansätze dafür werden in Kapitel 2 vorgestellt. Dazu zählt insbesondere der in dieser Arbeit verwendete: Die oben erwähnte - im Rahmen des SecureSCM-Projekts entwickelte - Fixpunktarithmetik von Catrina et al. sowie Erweiterungen wie die im Rahmen dieser Arbeit entstandene sichere Berechnung der Wurzelfunktion.

Genauso notwendig sind Protokolle zum Rechnen mit Matrizen und Vektoren von Shares. Die dazu notwendigen Methoden sind Inhalt von Kapitel 3.

Diese werden in Kapitel 4 für die erstmalige Implementierung von Algorithmen zum Lösen linearer Gleichungssysteme bestehend aus Sharings vertraulicher Werte angewandt und sind ein Kernstück der später vorgestellten Verfahren. Präsentiert werden zwei allgemeine Verfahren, die Lösung linearer Gleichungssysteme bestehend aus Sharings mit Hilfe der LR- und QR-Zerlegung sowie das Cholesky- und das CG-Verfahren, die beide eine symmetrisch positiv definite Matrix voraussetzen.

Thema von Kapitel 5 ist das Herzstück der vorliegenden Arbeit: Sichere Implementierungen, d.h. Implementierungen, die es ermöglichen mit Shares zu rechnen, ohne dass Zwischenergebnisse oder Eingabewerte von Spielern bekannt werden, zweier quadratischer Optimierungsalgorithmen. Es handelt sich dabei um zwei verschiedenartige, iterative Verfahren. Einerseits ist dies die Standard Aktive Mengen Strategie, bei der alle Iterationspunkte *primal* zulässig sind und andererseits das duale Verfahren von Goldfarb und Idnani, dessen Iterationspunkte alle *dual* zulässig sind. Da für erstere ein (primal) zu-

lässiger Startwert benötigt wird, wird ebenfalls gezeigt, wie sich ein solcher datenschutzfreundlich berechnen lässt. Dies bedeutet insbesondere, dass er den Teilnehmern nicht bekannt gemacht werden muss!

In Kapitel 6 schließlich wird - neben einem kleinen anderen Anwendungsbeispiel der sicheren, verteilten quadratischen Optimierung - die datenschutzfreundliche Implementierung von Support Vector Machines konstruiert. Es handelt sich dabei um ein zweistufiges Verfahren: Zuerst wird für einen Satz von Trainingsdaten, deren Klassenzugehörigkeit (dem Eigentümer) bekannt ist und die im hier betrachteten Szenario bei verschiedenen Datenbanken liegen, eine Hyperebene bestimmt, die die beiden Klassen möglichst gut trennt. Deren Berechnung besteht im Kern in der Lösung eines quadratischen Optimierungsproblems, das mit den in Kapitel 5 vorgestellten Methoden bestimmt werden kann. In einem zweiten Schritt wird dann bestimmt, auf welcher Seite der Hyperebene die zu klassifizierenden Daten sich befinden und so ihre Klasse ermittelt. Wichtig dabei ist, dass alle Berechnungen so durchgeführt werden können, dass keiner der Mitspieler Einblick in die Eingabedaten (außer seinen eigenen) oder Zwischenwerte erhält.

Alle beschriebenen Verfahren wurden mit Hilfe einer selbst entwickelten Programmbibliothek in C++ als sichere Mehrparteienberechnungen implementiert. Kapitel 7 enthält Evaluierungen und Tests der in den vorherigen Kapiteln beschriebenen Algorithmen. Neben einem praktischen Test der Wurzelfunktion sind dies ein Vergleich bzgl. Effizienz und Genauigkeit der Verfahren zum Lösen linearer Gleichungssysteme, der Test der quadratischen Optimierungsprotokolle an Standardbeispielen sowie Tests der Verfahren zur sicheren Berechnung von Support Vector Machines und Klassifikation. Alle vorliegenden Ergebnisse sind durchwegs gut und insbesondere in der Genauigkeit der Ergebnisse den nicht-sicheren Verfahren vergleichbar.

Im Anhang finden sich die Rohdaten des Vergleichs der sicheren Implementierungen von Gaußalgorithmus und QR-Verfahren (Anhang A), eine kurze Beschreibung der implementierten Bibliothek (Anhang B) und ihrer Funktionsweise, die zugehörigen Header-Files (Anhang C), sowie eine Zusammenstellung der Optimierungs- und Klassifikationsroutinen (Anhänge D und E).

1 Sichere Mehrparteienberechnungen

In diesem Kapitel soll eine kurze Einführung in die Theorie der sicheren Mehrparteienberechnungen, insbesondere in deren Umsetzung mit Hilfe von Secret Sharing Schemes und die sichere Implementierung von Kontrollstrukturen und Protokollen, gegeben werden. Da es sich dabei aufgrund des Themenumfangs nur um eine Übersicht handeln kann, wird der Leser gebeten, bei weitergehendem Interesse die referenzierte Literatur zu Rate zu ziehen. Grundsätzlich unterschiedlich sind Techniken, mit denen sich sichere Mehrparteienberechnungen einerseits für zwei und andererseits für drei oder mehr Spieler implementieren lassen. Die vorliegende Arbeit konzentriert sich auf letzteres Szenario. Es werden, wenn nicht anders erwähnt, immer 3 oder mehr Spieler angenommen.

1.1 Ansätze

Ziel einer sicheren Mehrparteienberechnung ist, eine Funktion

$$f : X_1 \times \cdots X_n \longrightarrow Y \tag{1.1}$$

auszuwerten, ohne dass die Eingabewerte, die verteilt bei mehreren Spielern $P_1, \ldots, P_n$ liegen, oder Zwischenergebnisse bekannt werden. Nur das Ergebnis $f(x_1, \ldots, x_n) \in Y$ ist allen Teilnehmern zugänglich. Implementierungen von Sicheren Mehrparteienberechnungen lassen sich grob in 3 Kategorien unterteilen:

- Generische Konstruktionen mittels „garbled circuits“
- Konstruktionen mit Hilfe homomorpher Verschlüsselung
- Konstruktionen mit Hilfe von Secret Sharing

Einen vollständigen Überblick über die angesprochenen Methoden zu geben, würde den Umfang dieser Arbeit übersteigen. In diesem Abschnitt werden deswegen nur die grundlegenden Prinzipien erläutert. Weitergehende Informationen sind in den im Folgenden angegebenen Literaturquellen erhältlich. Gute Übersichten bieten z.B. [JF06] oder [Sec08].

1.1.1 Implementierung mittels generischer Konstruktionen

Der generische Ansatz ist wohl der älteste. Auf ihm beruht die Lösung des Millionärsproblems ([Yao82]) von Yao, das die Forschung an sicheren Mehrparteienberechnungen begründetete: Zwei Millionäre wollen herausfinden, wer der Reichere ist, ohne das eigene Vermögen preiszugeben. Yao zeigte, dass dies mit Hilfe von „garbled circuits“, d.h. verschleierten Schaltkreisen, mit denen die Berechnung implementiert, aber während der Ausführung keine Eingabewerte oder Zwischenergebnisse abgelesen werden können, möglich ist. Der Hauptaufwand besteht dabei darin, den Schaltkreis zu konstruieren. Die Auswertung ist in einer konstanten Anzahl an Kommunikationsrunden (unabhängig von der Problemstellung!) möglich ([BMR90]). Diesen Ansatz bezeichnet man auch als *generisch*, da gezeigt wurde ([BMR90],[GMW87]), dass sich damit *jeder* Algorithmus mit verteilten Eingabewerten mit Hilfe von garbled circuits implementieren lässt, wenn die Mehrzahl der Spieler ehrlich ist, d.h. das Protokoll befolgt und nicht kollaboriert.[1] Allerdings sind die resultierenden Programme nur für allereinfachste Fragestellungen geeignet, da für komplexere Probleme Speicher- und Rechenkapazitäten auch von modernen Computern nicht ausreichen.

1.1.2 Implementierung mittels homomorpher Verschlüsselung

Bei der Verwendung von homomorphen, asymmetrischen Verschlüsselungsverfahren, wie z.B. RSA ([RSA78]), werden die Eingabewerte eines jeden Spielers mit dem Public Key verschlüsselt. Mit den verschlüsselten Werten kann dann unter Ausnutzung der Homomorphieeigenschaft gerechnet werden. Die Schlüssel können entweder von einer vertrauenswürdigen dritten Instanz (Trusted Third Party) generiert werden oder von den Spielern durch ein interaktives Protokoll gemeinsam erzeugt werden ([ACS02],[GRR98],[Lor09]). Je nach dem verwendeten Kryptosystem können Multiplikationen (z.B. RSA, El-Gamal ([EG85])) oder Additionen (Paillier [Pai99]) von geheimen Werten durchgeführt werden. Das relativ neue voll-homomorphe Kryptosystem von Gentry ([Gen09]) erlaubt Additionen und Multiplikationen auf verschlüsselten Daten, ist aber trotz einiger Verbesserungen und Varianten ([vDGHV10],[Gen11], [SV09]) noch nicht praktikabel. Zudem steht die Entwicklung einer Variante für mehr als 2 Spieler noch am Anfang ([AJW11]). Somit bedeutet die praktische Beschränkung auf *eine* der Operationen $+$ oder $\cdot$ (einschließlich von $-$ bzw. $\div$) zusammen mit der Langsamkeit asymmetrischer, kryptographischer Verfahren, dass Rechnungen mit nicht-ganzen Zahlen, die für die Zielsetzung der vorliegenden Arbeit essentiell sind, nur mit großen Schwierigkeiten und

[1] Mehr dazu in Abschnitt 1.2.2

mit dem Einsatz komplizierter Protokolle (Kapitel 2,[FDH+11],[FK11]) realisierbar sind.

1.1.3 Implementierung mittels Linearen Secret Sharing Schemes

Mit Hilfe eines Secret Sharing Schemes (SSS) kann eine Information s so in n Teile (Shares), die den *Spielern* (Personen) $P_1, \ldots, P_n$ zugeordnet werden können, zerlegt werden, dass sie nur von festgelegten *qualifizierten* Teilmengen der Spieler rekonstruiert werden kann. Alle anderen Teilmengen von Spielern erhalten *keine* Information über s. Die Gesamtheit der qualifizierten Teilmengen bezeichnet man auch als *Zugriffsstruktur*. Formal definiert ist der Begriff wie folgt:

Definition 1 (Zugriffsstruktur)**.** *Sei $\mathcal{M} = \{A_1, \ldots, A_n\}$ eine endliche Menge. Eine* Zugriffsstruktur *für M ist eine Teilmenge $\Gamma \subset \mathcal{P}(\mathcal{M})$, so dass gilt:*

$$X \in \Gamma \wedge X \subset Y \Rightarrow Y \in \Gamma$$

Ist die Zerlegung mit linearen Operationen kompatibel, d.h. lineare Operationen auf den Teilen liefern wieder Teile des Ergebnisses der linearen Operation, so spricht man von einem *Linearen Secret Sharing Scheme*. In der Praxis ist die Zugriffsstruktur häufig eine *Schwellenwertstruktur* der Größe t, d.h. alle Teilmengen der Größe $t+1 \leq n$ oder mehr der Spieler sind zur Rekonstruktion befähigt. Die bekannteste Umsetzung ist das Lineare Secret Sharing Scheme von Shamir ([Sha79]). Dabei wird ein Geheimnis $s \in \mathbb{F}_q$ (für einen endlichen Körper $\mathbb{F}_q$) verteilt, indem das Polynom

$$p(x) = s + \sum_{i=1}^{t} a_i x^i \tag{1.2}$$

an n paarweise verschiedenen Stellen $x_1, \ldots, x_n$ ausgewertet wird (i.d.R. $x_i = i$). Der *Share* oder *Anteil* von Spieler P_i an s ist dann $p(x_i)$. Die Koeffizienten $a_1, \ldots, a_n$ sind zufällig gewählt. Offensichtlich erfüllt ein derartiges Schema o.g. Forderung nach Kompatibilität mit linearen Operationen. Die Rekonstruktion ist mit Hilfe der Lagrange-Interpolation von Polynomen oder äquivalenten Verfahren (z.B. Dividierte Differenzen) möglich. Der Grad t des Polynoms darf dazu $n-1$ nicht übersteigen. Da ein Polynom über einem Körper vom Grad t erst durch $t+1$ Werte an bekannten Stellen eindeutig festgelegt ist und t Werte Teil *jedes* Polynoms vom Grad t sein können, erhält eine Gruppe von t oder weniger Spielern *keinerlei* Information über das Geheimnis. Die Zugriffsstruktur eines solchen Shamir-Secret-Sharing-Schemes ist also eine Schwellenwertstruktur der Größe t. Man bezeichnet es auch einfach als (n,t)*-Shamir-Schema*.

Ist $x < q$ eine Zahl, so bezeichnet im Folgenden $[x]$ das *Sharing* von x, d.h. den Vektor $\begin{pmatrix} p(x_1) & \cdots & p(x_n) \end{pmatrix}$, der die den Teilnehmern zuzuordnenden Werte enthält. Ist x eine ganze Zahl, so muss q als Primzahl und nicht als Primzahlpotenz gewählt werden, da sonst arithmetische Operationen, selbst wenn die Ergebnisse und alle Zwischenwerte kleiner als q sind, nicht denen in den ganzen Zahlen entsprechen. Aus diesem Grund ist q in der Praxis immer eine Primzahl und nur für Bitdarstellungen von Zahlen wird q manchmal als p^n, $n > 1$ gesetzt. Im Gegensatz dazu ist ein Wert α, der *ohne* eckige Klammern geschrieben wird, immer eine öffentliche, d.h. allen Spielern bekannte Konstante. Mehr dazu in Abschnitt 2.2.

Die Multiplikation zweier Geheimnisse $[x]$ und $[y]$ ist mittels spezieller Protokolle wie z.B. [GRR98],[Lor09],[LW11] möglich. Der Grad des Polynoms, auf dem die Anteile liegen, ist dabei zum Ende des Protokolls so groß wie am Anfang, darf aber $t = \frac{n-1}{2}$ nicht übersteigen. Man schreibt auch einfach $\mathsf{Mul}([x],[y])$ oder $[x] \cdot [y]$.[2] Die analoge Schreibweise für die Multiplikation (Addition) mit einer (öffentlichen) Konstanten α ist $\alpha \cdot [x]$ (bzw. $\alpha + [x]$) und für Addition bzw. Subtraktion zweier Sharings $[x] \pm [y]$. Nach jeder elementaren arithmetischen Operation von Sharings erhält jeder Spieler wieder einen Anteil an einem Sharing, das das Ergebnis der Operation darstellt.

Soll ein konkretes Problem, dessen Eingabedaten verteilt bei mehreren Spielern vorliegen, mit Hilfe von sicheren Mehrparteienberechnungen basierend auf einem Shamir Secret-Sharing Scheme, gelöst werden, so verteilt jeder Spieler *vor* der eigentlichen interaktiven Berechnung jeden seiner Eingabewerte z, indem er dafür die Funktion (1.2) mit zufälligen Koeffizienten und konstantem Term z an den Stellen $1, \ldots, n$ auswertet und die Funktionswerte den Spielern $P_1, \ldots, P_n$ zukommen lässt. Dieses Vorgehen wird im Folgenden, insbesondere bei der Beschreibung der sicheren Implementierungen der Optimierungsalgorithmen, für die jeder Teilnehmer andere Eingabewerte beisteuert, implizit angenommen.

Weitere Informationen zum Secret Sharing, insbesondere Erweiterungen und der Zusammenhang mit Error Correcting Codes finden sich in [CC06],[CCG+07], [CDdH07],[CDM00] und [CFS05]. Geschieht ein so beschriebenes polynomiales Secret Sharing über einem endlichen Körper, so sind die Shares zufällig und das Secret Sharing Scheme ist informationstheoretisch (Abschnitt 1.2.2) sicher ([Sha79],[DN07]).

[2] Es muss $x \cdot y < q$ gelten, damit eine Multiplikation zweier Sharings das gewünschte Ergebnis liefert. I.d.R. wird q so groß gewählt, dass es die Größe der erwarteten Zahlen weit übersteigt.

1.2 Angreifermodell und Sicherheit

1.2.1 Angreifermodell

Ein Angreifer wird als ein Außenstehender modelliert, der eine Teilmenge der Spieler kontrolliert. Grundlegend unterscheidet man zwischen zwei Arten von Angreifern ([GMW87]).

- Die Spieler, die von einem *passiven Angreifer* kontrolliert werden, befolgen das Protokoll, versuchen aber in Zusammenarbeit Zwischenergebnisse, Eingabewerte oder Informationen, die über ihre eigenen hinausgehen, abzuleiten.

- Die Spieler, die von einem *aktiven Angreifer* kontrolliert werden, sind während der Ausführung des Protokolls keinen Restriktionen unterworfen.

Obwohl Sicherheit gegen aktive Angreifer erstrebenswert ist und Sicherheit nur gegen passive Angreifer auf den ersten Blick wenig wertvoll erscheinen mag, ist sie doch in vielen Fällen, insbesondere dann, wenn jeder Teilnehmer einer Berechnung ein (wirtschaftliches) Interesse an einem korrekten Ergebnis besitzt, ausreichend. Zudem ist Sicherheit gegen aktive Angreifer oft nur mit beträchtlichem Mehraufwand ([GRR98]) zu gewährleisten, der viele Anwendungen unpraktikabel macht. Dieses Szenario wird auch in vielen anderen bisherigen Anwendungen von Sicheren Mehrparteienberechnungen (.B. [BCD+09],[Sec10],[CdH10b]) vorausgesetzt. Es werden im Folgenden deswegen nur passive Angreifer betrachtet.

1.2.2 Sicherheit

Unabhängig vom Angreifermodell gibt es zwei verschiedene Arten von Sicherheitsanforderungen. Sie lauten informell:[3]

- Ein Protokoll ist *informationstheoretisch sicher*, wenn auch mit unendlichen Rechenkapazitäten aus den Daten, die einer Teilmenge der Spieler, die nicht in der Zugriffsstruktur enthalten ist, während der Ausführung des Protokolls bekannt werden, keinerlei Rückschlüsse über Eingabewerte der anderen Spieler oder Zwischenwerte der Berechnung gezogen werden können.

- Ein Protokoll ist *kryptographisch sicher*, wenn mit begrenzten Rechenressourcen, die während der Ausführung eines Protokolls produzierten Daten einer Teilmenge der Spieler, die nicht in der Zugriffsstruktur enthalten ist, nicht von zufälligen Daten unterschieden werden können ([GMW87]).

[3] Für die formale Definition siehe [BOGW88] und [GMW87]

	Kryptographisch	Informationstheoretisch
Aktive Angreifer	$t < \frac{n}{2}$	$t < \frac{n}{3}$
Passive Angreifer	$t < n$	$t < \frac{n}{2}$

Tabelle 1.1: Maximale Anzahl korrumpierter Teilnehmer in den verschiedenen Sicherheits- und Angreiferszenarien ([BOGW88],[GMW87])

Kryptographische Sicherheit ist möglich, wenn Trapdoor-Einwegfunktionen existieren.

Die Höchstzahl korrumpierter Teilnehmer, mit denen noch die ordnungsgemäße Beendigung eines Protokolls, das eine sichere Mehrparteienberechnung implementiert, garantiert werden kann, findet sich in Tabelle 1.1.

Beispiel: Die Verteilung eines Geheimnisses $s \in \mathbb{F}_q$ mittels eines Secret Sharing Schemes (Abschnitt 1.1.3) ist informationstheoretisch sicher. Die Anteile der einzelnen Spieler sind rein zufällig und sagen, solange keine zur Rekonstruktion befähigte Teilmenge vorliegt, nichts über s aus. Ist der Körper nicht endlich, gilt dies nicht.

Für eine genauere Definition der kryptographischen Sicherheit wird noch die Definition der *vernachlässigbaren Abbildung* benötigt.

Definition 2 (Vernachlässigbare Abbildung)**.** *Eine Abbildung*

$$\varphi : \mathbb{N} \longrightarrow \mathbb{R}^+ \tag{1.3}$$

heißt vernachlässigbar*, wenn für jedes Polynom $P : \mathbb{Z} \to \mathbb{Z}$ ein $n_0 \in \mathbb{N}$ existiert, so dass für alle $n > n_0$*

$$\varphi(n) \leq \frac{1}{P(n)} \tag{1.4}$$

gilt.

Die folgende Definition stammt aus [CS10].

Definition 3 (Statistische Sicherheit)**.** *Seien X und Y Zufallsvariablen über endlichen Mengen V und W und sei $\Delta(X,Y) = \frac{1}{2}\sum_{v \in V \cup W} |Pr(X = v) - Pr(Y = v)|$ die statistische Differenz. Die Verteilungen sind* vollkommen ununterscheidbar*, wenn $\Delta(X,Y) = 0$ und* statistisch ununterscheidbar*, wenn $\Delta(X,Y)$ vernachlässigbar ist in einem Sicherheitsparameter κ. Ein Protokoll ist* vollkommen *oder* statistisch privat*, wenn alle Daten, die während der Ausführung des Protokolls einem Gegner bekannt werden können, simuliert werden können, so dass die echten und simulierten Verteilungen vollkommen oder statistisch ununterscheidbar sind.*

Statistische Sicherheit ist eine Form kryptographischer Sicherheit. Die später vorgestellten Protokolle bieten teils vollkommene, teils statistische Sicherheit. Die Ergebnisse aus [Can00], [Can01] und [CDD+04] stellen sicher, dass die Komposition sicherer Protokolle wieder sicher ist - vorausgesetzt es werden bei der Komposition keine Informationen preisgegeben. Insbesondere müssen also für die Komposition sicherer Protokolle keine neuen Sicherheitsbeweise durchgeführt werden!

1.3 Kontrollstrukturen

Kontrollstrukturen sind für Algorithmen essentiell. Bei sicher verteilten Berechnungen darf der Verlauf eines Programms aber keine Rückschlüsse über die verwendeten Daten liefern. Somit sind z.B. **if**-Abfragen oder Abbruchbedingungen von **while**-Schleifen, die von geheimen Daten abhängen, unzulässig. Im folgenden Abschnitt soll gezeigt werden, wie sich Kontrollstrukturen bei sicher verteilen Berechnungen trotzdem realisieren lassen.

1.3.1 If

Die Anweisung

If$(a = 1)$

$x_1 \leftarrow y_1$

$\vdots$

$x_n \leftarrow y_n$

Else $(a = 0)$

$x_1 \leftarrow z_1$

$\vdots$

$x_n \leftarrow z_n$

lässt sich transformieren zu

$x_1 \leftarrow a \cdot y_1 + (1 - a) \cdot z_1$

$\vdots$

$x_n \leftarrow a \cdot y_n + (1 - a) \cdot z_n$

Auf diese Weise sind die Schritte, die während der Berechnung ausgeführt werden, unabhängig von der entscheidenden Variable *a*. Komplizierter wird es, wenn sich ein Algorithmus auf mehreren Stufen verzweigt. In diesem Fall müssen auf die oben beschriebene Weise *alle* Zweige durchlaufen werden. Dies kann eine deutliche Verlangsamung bedeuten bzw. die Ausführung mancher Algorithmen auch unmöglich machen. In diesem Fall sollte über eine andere Implementierung des Algorithmus nachgedacht werden.

1.3.2 While

Wie eingangs erwähnt ist es nicht möglich, eine **while**-Schleife zu implementieren, deren Abbruchbedingung geheim ist. Aus diesem Grund müssen **while**-Schleifen innerhalb eines Algorithmus durch Schleifen fester Länge (d.h. **for**-Schleifen), in denen das Problem ordnungsgemäß bearbeitet wird bzw. die eine Fehlermeldung liefern, wenn dies nicht der Fall ist, ersetzt werden.

Abweichend davon ist eine **while**-Schleife als äußere Hülle eines Algorithmus zulässig, da sonst viele Probleme nicht sinnvoll sicher bearbeitet werden können. Dies ist z.B. der Fall in den sicheren Implementierungen des Simplex-Algorithmus in [Tof07],[Tof09] und [CdH10b]. Eine Implementierung des Simplex-Algorithmus *ohne* Abbruchkriterien müsste, um sicherzugehen, dass alle potentiellen Lösungen durchlaufen sind, alle Ecken überprüfen. Deren Anzahl ist jedoch exponentiell in der Anzahl der Variablen! Praktisch lässt sich ein solcher Abbruch durchführen, indem der Wert des Abbruchkriteriums aus den Shares der Spieler rekonstruiert wird und anschließend überprüft wird, ob das Abbruchkriterium erfüllt ist. Die so veröffentlichte Informationsmenge beschränkt sich also auf die Anzahl der Iterationen, die nötig sind das Problem zu lösen. Über das Problem selbst sagt dies i.d.R. nur sehr wenig aus.

1.3.3 Goto

Goto-Vorschriften in iterativen Algorithmen lassen sich leicht mit Hilfe von Schaltervariablen umsetzen: Wird eine **goto**-Anweisung aktiv, setzt der Algorithmus eine entsprechende Schaltervariable auf den Wert 1, sonst 0. Beim nächsten Durchlauf des Algorithmus wird dann - mit Hilfe der sicheren Implementierung der **if**-Abfragen aus Abschnitt 1.3.1 - nur bei denjenigen Variablen der Wert geändert, für die der Wert der Schaltervariable 1 ist, d.h. die im Bereich der **goto**-Anweisung liegen. Hat der Algorithmus keine Schleifenstruktur, muss eine hinreichende Anzahl an Schleifen eingeführt werden, die die ordnungsgemäße Beendigung des Programms garantiert!

1.4 Komplexität

1.4.1 Rechenkomplexität

Die (Rechen-)Komplexität eines sicheren, verteilten Algorithmus basierend auf Secret Sharing wird i.d.R. mit der Anzahl der zu seiner Ausführung benötigten sicheren, verteilten Multiplikationen angegeben. Diese sind derjenige elementare Baustein eines jeden Protokolls, zu dessen Ausführung Interaktion (eine Kommunikationsrunde für das Protokoll aus [GRR98]) zwischen den teilnehmenden Parteien nötig ist. Die anderen elementaren Operationen, Addition, Subtraktion und Multiplikation mit einer Konstanten[4] können nicht-interaktiv durchgeführt werden. Andere Operationen, wie Rekonstruieren einer geheimen Variable, sind im Rechen- und Kommunikationsaufwand zu einer Multiplikation äquivalent. Grund für diese Metrik ist die Annahme, dass die Zeit, die für die Kommunikation zwischen den Rechnern benötigt wird, d.h. Latenz des Netzwerks und Warten auf andere Teilnehmer, die für das Berechnen der lokalen Operationen benötigte Zeit bei weitem übersteigt. Diese können also - in der Theorie - während der Wartezeit durchgeführt werden.

Die Komplexität der Algorithmen hängt weiterhin von den Parametern k und f der verwendeten Fixpunktzahlen (s. Kapitel 2), davon abgeleiteten Werten sowie der Größe der auftretenden Matrizen und Vektoren ab. Aus Gründen der Übersichtlichkeit wird teilweise die $\mathcal{O}$-Notation verwendet und Ausdrücke der Form $\lceil \log_2(k-1) \rceil$ und $\lceil \log_k \rceil$ ($k \in \mathbb{N}$) zu $2\lceil \log_2 k \rceil$ zusammengefasst, auch wenn dies nicht in allen Fällen vollkommen richtig ist.

1.4.2 Rundenkomplexität

Ein weiteres Maß für die Komplexität eines Algorithmus ist die Rundenzahl, d.h. die Anzahl an Kommunikationsrunden, die für die Ausführung eines sicheren Programms benötigt werden. Diese ist i.d.R. ungleich der Anzahl der sicheren Multiplikationen, da Operationen häufig parallelisiert werden können. Allerdings kann eine geringe Rundenzahl auch in die Irre führen: Wie in Kapitel 3 zu sehen sein wird, kann eine sichere Matrix-Matrix-Multiplikation - unabhängig von der Größe der Matrix - in einer Kommunikationsrunde durchgeführt werden, da die Berechnungen der einzelnen Einträge nicht voneinander abhängig sind. Jedoch sind in der Realität der Parallelisierung durch die verfügbaren Rechner-Ressourcen Grenzen gesetzt, so dass die Multiplikation großer Matrizen in der Realität immer wesentlich länger dauern wird, als die kleiner. Ein weiteres Beispiel ist die Erzeugung von Zufallsbits: Bereits für relativ kleine Programme werden oft mehrere hundert oder tausend zufällige Bits

[4] Mehr zur Division zweier geheimer Werte in Kapitel 2.

benötigt. Es ist nicht vorstellbar, deren Erzeugung vollständig zu parallelisieren.

Um die Parallelisierungsmöglichkeiten eines Algorithmus voll ausnutzen zu können, muss das Programm individuell an den verwendeten Rechner angepasst werden. Sind die an einer sicheren Mehrparteienberechnung beteiligten Rechner in ihrer Hard- oder Software-Konfiguration verschieden, muss die Parallelisierung jeweils individuell gehandhabt werden. Dies ist in den wenigsten Szenarien realistisch. Weiterhin stellt sich bei beschränkten Ressourcen die Frage, auf welchem Niveau eine Parallelisierung betrieben werden soll: Sollen vorrangig große Bausteine nebeneinander berechnet werden und deren Zusammensetzung aus primitiven Elementen (d.h. Multiplikationen) sequentiell erfolgen, auch wenn eine weitergehende Parallelisierung möglich wäre? Oder sollen die primitiven Elemente innerhalb der Bausteine parallelisiert werden und die Bausteine selbst sequentiell berechnet werden? Die Antwort hängt wohl i.d.R. vom verwendeten Algorithmus ab, der dann entsprechend angepasst werden muss. Dies schränkt die Wiederverwertbarkeit des Codes stark ein und bedeutet einen erheblichen Mehraufwand. Aus diesen Gründen sollte die Rundenkomplexität nur als Hinweis gesehen werden, an welchen Stellen eines Algorithmus Möglichkeiten zur Parallelisierung bestehen. In den Protokollen ist dies bei Schleifen an dem Hinweis **do parallel** zu sehen. Deswegen wird im Folgenden die Rundenkomplexität zwar angegeben, es wird aber bei Effizienzbetrachtungen der Fokus jedoch eindeutig auf die Rechenkomplexität gelegt.

1.4.3 Prä-Berechnungen/Pre-Computation

Viele Quellen geben zusätzlich zu den bereits genannten Werten auch solche an, bei denen Zufallsbits und andere Werte im Voraus berechnet werden (auch bezeichnet als pre-computation). Die zugrunde liegende Annahme ist dabei, dass die entsprechenden Werte vor der eigentlichen Mehrparteienberechnung berechnet werden können (z.B. nachts) und so zur Komplexität eines Programms nicht beitragen. Ein Algorithmus kann in dieser Darstellung schlanker erscheinen. Jedoch ist die Berechnung solcher Werte dem Programm trotzdem zuzurechnen und sollte in der Komplexitätsanalyse auch so betrachtet werden, zumal die Kosten der Prä-Berechnungen bis zu 50% der Gesamtkosten erreichen können ([CdH10b])! Zudem sind häufig wiederkehrende Mehrparteienberechnungen zwischen den selben Parteien sehr selten (bei der in [BCD+09] beschriebenen z.B. einmal jährlich) und es würde die wahren Kosten unnötig verschleiern, vorausberechenbare Werte bei der Betrachtung der Komplexität nicht zu beachten. Aus diesem Grund wird im Folgenden auf diese Angabe verzichtet.

Bitoperation	Formel	Kosten	Körper
NOT($[x]$)	1-$[x]$	0	alle
AND($[x], [y]$)	$[x] \cdot [y]$	1	alle
OR($[x], [y]$)	$[x] + [y] - [x] \cdot [y]$	1	alle
XOR($[x], [y]$)	$[x] + [y] - 2[x] \cdot [y]$	1	$\text{char}(\mathbb{F}) \neq 2$
XOR($[x], [y]$)	$[x] + [y]$	0	$\text{char}(\mathbb{F}) = 2$

Tabelle 1.2: Kosten der Implementierung der logischen Operationen

Protokoll 1.1: $[Z] \leftarrow$ **RS** (d)

1 **P1:** $[S(1)] \leftarrow \mathsf{Share}(d, n, t, q)$
2 **For**$(i = 2, \ldots, n)$
3 **P**i: $[S(i)] \leftarrow \mathsf{Share}(0, n, t, q)$
4 $[Z] \leftarrow \sum_{i=1}^{n} [S(i)]$
5 **Return** $[S]$

Abbildung 1.1: Erzeugen eines Sharings einer beliebigen (ganzen) Zahl. Die Bezeichnung Pi bedeutet dabei, dass die entsprechende Anweisung nur von Spieler i auszuführen ist und die resultierenden Shares an die Mitspieler zu verteilen sind.

1.5 Bitoperationen

Die Bitoperationen OR, XOR, AND und NOT für Bits $([x], [y])$ können bei Verwendung eines Shamir-Secret Sharing Schemas (oder jedem anderen, in dem Addition und Multiplikation geheimer Werte möglich sind) mit elementaren Operationen sicher implementiert werden (Tabelle 1.2). Führt man XOR$([x], [y])$ in einem Körper der Charakteristik 2 durch, lässt sich eine sichere Multiplikation einsparen.

1.6 Zufällige Sharings von 0 und 1

An einigen Stellen in den vorgestellten Protokollen werden zufällige Sharings von 0 oder 1 benötigt. Ein zufälliges Sharing von 0 (RSZ) lässt sich im passiven Sicherheitsmodell implementieren, indem jeder Spieler ein Sharing der 0 erstellt, die Shares an die anderen Spieler verteilt und die erhaltenen Shares aufsummiert. Dies erfordert eine Kommunikationsrunde. Genauso kann ein zufälliges Sharing einer beliebigen anderen (ganzen) Zahl erstellt werden, indem *ein* bestimmter Spieler ein Sharing dieser Zahl erstellt und die Spieler es zu ihren Sharings der Null hinzuaddieren (Protokoll 1.1).

Alternativ können die Techniken aus [CDI05] verwendet werden, zufällige Sharings zufälliger (oder bestimmter) Zahlen zu erzeugen, nach einer Setup-Phase sogar *ohne* Kommunikation! Ein (pseudo-)zufälliges Sharing der 0 wird mit *(P)RSZ* bezeichnet und das einer beliebigen anderen Zahl a mit *(P)RS(a)*.

2 Die Fixpunktarithmetik von Catrina/Saxena und Erweiterungen

Kernstück dieses Kapitels ist die Beschreibung der sicheren, verteilten Fixpunktarithmetik von Catrina/Saxena und der für diese Arbeit notwendigen Erweiterungen, insbesondere der sicheren Berechnung der Wurzelfunktion. Zunächst sollen jedoch die anderen Varianten, einen Algorithmus, der mit nicht-ganzen Zahlen operiert, sicher zu implementieren, kurz erörtert werden.

2.1 Sicheres verteiltes Rechnen mit nicht-ganzen Zahlen

Zur Implementierung des sicheren verteilten Rechnens mit nicht-ganzen Zahlen gibt es verschiedene Ansätze. In den meisten Fällen ist das zentrale Problem, die Division zweier geheimer Werte so umzusetzen, dass das Ergebnis wieder ein geheimer Wert ist. Die verschiedenen Lösungsansätze werden im Folgenden kurz vorgestellt.

2.1.1 Umschreiben der Algorithmen

Der elementarste Weg mit nicht-ganzen Zahlen zu rechnen, besteht darin, einen gegebenen Algorithmus

$$y \leftarrow \mathcal{A}(x) \tag{2.1}$$

für einen Vektor von Eingabewerten x[1] und einen Vektor von Ausgabewerten y so in einen Algorithmus

$$y' \leftarrow \mathcal{A}'(x) \tag{2.2}$$

zu transformieren, dass $\mathcal{A}$ und $\mathcal{A}'$ für dieselben Eingabewerte x Ausgabewerte y und y' liefern, derart, dass für ein selbstgewähltes $\varepsilon > 0$

$$|y - y'| < \varepsilon^2 \tag{2.3}$$

sowie die weiteren Bedingungen erfüllt sind:

[1] Die Zahl der Ein- und Ausgabewerte sowie ob es sich dabei um reelle, ganzzahlige oder komplexe Werte handelt, sei hier offengelassen

[2] Die Existenz einer Betragsfunktion auf dem Ergebnisraum sei vorausgesetzt. In fast allen Fällen ist dies gegeben.

1. Sind einer oder mehrere Eingabewerte nicht-ganzzahlig, so müssen sie im ersten Schritt von $\mathcal{A}'$ in ganzzahlige transformiert werden.
2. Im letzten Schritt ist eine Transformation (öffentlicher) ganzzahliger Werte in nicht-ganzzahlige zugelassen.
3. In allen anderen Schritten werden ausschließlich Rechnungen auf den ganzen Zahlen ausgeführt.
4. In $\mathcal{A}'$ werden auf geheimen Operanden nur die Operationen $+, -, \cdot$ ausgeführt und $\div$ nur dann, wenn der Divisor Faktor des Dividenden ist.

Bemerkungen:

1. Forderung 3 ist sehr restriktiv und führt dazu, dass für die allermeisten numerischen Algorithmen ein derartiges Vorgehen nicht möglich bzw. nicht-praktikabel ist, da die Umformung von Schritten, die eine Division $\frac{[a]}{[b]}$ zweier geheimzuhaltender Zahlen $[a]$ und $[b]$, die in keiner Beziehung zueinander stehen, enthält, in eine Form, die (4) genügt, in der Regel nicht möglich ist.
2. Die Größe der in $\mathcal{A}'$ auftretenden Zahlen kann sehr groß werden. Dies kann die Praktikabilität einer solchen Umsetzung beeinträchtigen.

Das bekannteste Beispiel einer solchen Umsetzung ist die sichere verteilte Implementierung des Simplex-Algorithmus in [Tof07], die sich der Implementierung des Simplex-Algorithmus mit ganzzahliger Pivotisierung bedient ([Ros04]).

2.1.2 Bruchrechnung

Eine naive Implementierung der Bruchrechnung, d.h. mit separater Behandlung des Zählers und Nenners ist theoretisch möglich, aber nicht praktikabel. Da zum jetzigen Zeitpunkt keine effiziente, sichere, verteilte Umsetzung des euklidischen Algorithmus zur Verfügung steht, können Brüche nicht gekürzt werden, so dass die Zahlen in Zähler und Nenner innerhalb kürzester Zeit sehr groß würden.

Der etwas komplexere Ansatz aus [FSW03] verwendet das Paillier-Kryptosystem ([Pai99]) $\mathcal{P}$, um die Bruchrechnung zu implementieren. Zunächst wird für einen Bruch $t = \frac{r}{s}$ mit teilerfremden r und s der *Repräsentant* $r \cdot s^{-1}$ mod N für einen hinreichend großen (öffentlichen) RSA-Modulus N berechnet. Die Addition zweier geheimer Brüche bewerkstelligt sich dann mit Hilfe der additiv-homomorphen Eigenschaften von $\mathcal{P}$. Gleiches gilt für die Multiplikation mit einem öffentlichen Bruch. Die Rückumwandlung eines Repräsentanten geschieht mit Hilfe von Gittern. Allerdings birgt eben skizziertes Vorgehen dreierlei Nachteile.

1. In der beschriebenen Form ist das Verfahren eher geeignet, Inhalte einer Datenbank verschlüsselt zu speichern, um sie vor unbefugtem Zugriff zu schützen und dabei dennoch die Möglichkeit zu behalten, einige elementare Operationen auf den Daten auszuführen, ohne sie zu entschlüsseln, als für „echte“ sichere Mehrparteienberechnungen (dafür müssten z.B. die Parameter des Kryptosystems sicher verteilt erzeugt werden ([BF01]),[DM10]).

2. Eine Multiplikation zweier geheimer Brüche ist nicht möglich.

3. Nach einer begrenzten Anzahl von Operationen muss der Wert entschlüsselt und neu verschlüsselt werden.

Insbesondere aus den Gründen (2) und (3) ist der Ansatz aus [FSW03] für komplexere Algorithmen nicht geeignet.

2.1.3 Gleitkommaarithmetik

Die Implementierung einer sicheren, verteilten Gleitkommaarithmetik wäre der beste Ansatz, mit nicht-ganzen Zahlen naturwissenschaftlichen oder wirtschaftlichen Ursprungs zu rechnen. Eine Gleitkommazahl

$$\pm\, a = \pm c, m \cdot b^e \tag{2.4}$$

der Länge l Bit besteht aus einer Zahl $c \in \{0, \ldots, b-1\}$, der *Mantisse* $m < 1$, der *Basis* b und dem *Exponenten* e. Die (maximalen) Längen von Mantisse und Exponent sind unabhängig voneinander festgelegt (in einem weiteren Bit wird das Vorzeichen gespeichert)[3]. Dies bedeutet, dass sich betragsmäßig kleine Zahlen genauer darstellen lassen als betragsmäßig große Zahlen und dass der *relative Fehler*

$$\frac{(a \times b) - (a \otimes b)}{a \times b}_4 \tag{2.5}$$

für die elementaren Rechenoperationen $\{+, -, \cdot, \div\}$ unabhängig ist von der Größe der beteiligten Zahlen. Als größter Stolperstein bei einer sicheren Implementierung erweist sich dabei der Exponentenangleich bei der Addition: Seien

$$a = \tilde{a_1}, m_1 \cdot b^{e_1} \text{ und } b = \tilde{b_1}, m_2 \cdot b^{e_2} \tag{2.6}$$

[3] Im Sonderfall $b = 2$ ist a immer gleich 1, muss also nicht explizit gespeichert werden. Es steht für m und e ein Bit mehr Speicherplatz zur Verfügung

[4] Es sei $\times$ eine beliebige elementare Rechenoperation und $\otimes$ deren Umsetzung in Gleitpunktarithmetik

zwei Gleitkommazahlen zur Basis b mit Mantissen m_1 und m_2, Exponenten e_1 und e_2 und $\tilde{a_1}, \tilde{b_1} \in \{0, \ldots, b-1\}$ (Das Vorzeichen sei hier aus Gründen der Einfachheit $+$). Möchte man die Summe $a + b$ berechnen und ist (o.E.) $e_1 > e_2$, so müssen zunächst die Exponenten angeglichen werden durch

$$b \to b' = 0, \underbrace{0 \ldots 0}_{e_1 - 1} \tilde{b_1} m_2 \cdot b^{e_1}. \tag{2.7}$$

Der Exponentenangleich beinhaltet also die Bestimmung eines Maximums sowie eine Rechtsverschiebung um $e_1 - e_2$ Stellen. Mit den derzeit bekannten Mitteln ist dies nicht effizient durchführbar. Die Maximumsbestimmung ist zwar möglich, aber für eine elementare Operation wie die Addition viel zu teuer. Der beste (und nach Wissen des Autors bisher einzige) Versuch ist in [FK11] und [FDH+11] dargestellt. Er beschränkt sich allerdings auf das Zwei-Parteienszenario und ist wesentlich von vorausberechneten Tabellen abhängig, die wiederum von der gewünschten Genauigkeit abhängen. Insbesondere bedeutet dies, dass für eine andere Genauigkeit neue Tabellen berechnet werden müssen, was die Anwendbarkeit weiter einschränkt. Aus diesem Grund und weil in dieser Arbeit immer 3 oder mehr Spieler vorausgesetzt werden, ist dieses Schema nicht praktikabel.

Die vielseitigste Methode zum Rechnen mit nicht-ganzen Zahlen ist die im nächsten Abschnitt vorgestellte Fixpunktarithmetik von Catrina/Saxena.

2.2 Die Fixpunktarithmetik von Catrina/Saxena

Fixpunktzahlen sind definiert als Zahlen fester Bitlänge k, von denen – anders als bei Gleitkommazahlen – *immer* f Nachkommastellen sind. Nach der Multiplikation zweier Fixpunktzahlen muss die resultierende Zahl um f Stellen gekürzt werden, was durch Abschneiden oder Runden geschehen kann. Ebenso ist durch k die Größe der Zahlen, die miteinander multipliziert oder dividiert werden können, beschränkt. Der relative Fehler bei den elementaren Rechenoperationen ist - anders als bei der Gleitkommaarithmetik - nicht unabhängig von der Größe der Zahlen. Trotzdem eignet sich die Fixpunktarithmetik besonders gut für eine Implementierung als sichere Mehrparteienberechnung, da kein Exponentenangleich bei der Addition wie bei der Gleitkommaarithmetik vorgenommen werden muss. Die in [CS10], [CdH10a] und [CdH10b] vorgestellten Methoden zur Implementierung einer sicheren, verteilten Fixpunktarithmetik rechnen mit Sharings von ganzen Zahlen der Länge k Bits mit einem gedachten Komma nach f Stellen. Dies bedeutet, dass $e = k - f - 1$ Vorkommastellen zur Verfügung stehen sowie eine Stelle für das Vorzeichen. Benötigt wird ein unterliegendes lineares Secret Sharing Scheme mit einem Multiplikationsprotokoll.

Genauer: Definiert man die Menge

$$\mathbb{Z}_{<k>} = \{x \in \mathbb{Z} | -2^{k-1} + 1 \leq x \leq 2^{k-1} - 1\}, \tag{2.8}$$

dann definiert diese wiederum implizit den Zahlenbereich

$$\mathbb{Q}_{<k,f>} = \{\bar{x} \in \mathbb{Q} | \bar{x} = x \cdot 2^{-f},\ x \in \mathbb{Z}_{<k,f>}\}, \tag{2.9}$$

d.h. die Menge der Fixpunktzahlen zur Basis 2 der Länge k Bits mit f Nachkommastellen. Man betrachtet $\mathbb{Z}_{<k,f>}$ als Teil eines endlichen Körpers $\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z}$ für eine Primzahl q, die so groß ist, dass bei keiner Rechnung eine Modulo-Reduzierung nötig ist (z.B. $k = 100, log_2 q \approx 1024$). Die Rundung geschieht i.d.R. mit Hilfe des Protokolls TruncPr (probabilistisches Abschneiden), das eine vorgegebene Anzahl von Stellen abschneidet, aber einen Fehler im letzten Bit zulässt ([CS10]).[5] Es ist dem deterministischen Abschneiden (Trunc) vorzuziehen, da dieses wesentlich teurer ist.

Bemerkung: Die teilweise von den Werten in [CS10] und [CdH10a] abweichenden Werte aus Tabelle 2.1 sind einerseits auf den Einsatz verbesserter Sub-Protokolle (aus [Sec10], [Sec08] und [Sec09]) zurückzuführen, andererseits auf die Korrektur kleinerer Fehler. Siehe dazu auch 2.2.4.

2.2.1 Sicherheit

Die Fixpunktarithmetik von Catrina/Saxena bietet statistische Sicherheit (vgl. Definition 3) gegen passive Angreifer. Insbesondere bedeutet dies, dass immer eine Mehrheit der Spieler ehrlich sein muss, d.h. nicht kollaborieren darf (vgl. Tabelle 1.1)[6] und die tatsächliche Ausführung der Protokolle statistisch ununterscheidbar (in einem Sicherheitsparameter κ) von einer simulierten ist. Protokolle wie [Bla11], die Sicherheit gegen aktive Angreifer bieten, sind um vieles langsamer und somit für aufwendige numerische Berechnungen nicht geeignet.

2.2.2 Protokolle

In Tabelle 2.1 sind die wichtigsten zur Fixpunktarithmetik von Catrina/Saxena gehörigen Protokolle vorgestellt. Das Protokoll TruncPr dient dem schon besprochenen (probabilistischen) Abschneiden der f überzähligen Stellen nach einer sicheren Multiplikation. Trunc ist die deterministische Variante, die aber

[5] [ACS02] enthält ein ähnliches Protokoll, das jedoch wesentlich ungenauer ist.

[6] Die im Vergleich zu Tabelle 1.1 geringere Toleranzschwelle beruht darauf, dass für n Spieler für das Multiplikationsprotokoll Secret Sharing Polynome vom Grad kleiner als $\frac{n}{2}$ benötigt werden (vgl. Abschnitt 1.1.3).

fast nur als Teilroutine von LTZ, der Funktion, die überprüft, ob ein Sharing kleiner als 0 ist, eingesetzt wird. $\mathsf{FPDiv}([a],[b],k,f)$ implementiert die sichere Division $\frac{[a]}{[b]}$ genauso wie DivNR, das jedoch anstelle von Goldschmidt-Iterationen ([Mar04]) Newton-Raphson Iterationen verwendet ([SB02]). Division durch eine öffentliche Konstante (DivKonst) ist Multiplikation mit dem mit 2^f skalierten Reziproken, gefolgt von einer TruncPr-Operation, also genauso teuer wie diese ([CS10]).

Die Protokolle $\mathsf{LT}([a],[b],k)$ (prüft, ob $a < b$), $\mathsf{GT}([a],[b],k)$ (prüft, ob $a > b$), $\mathsf{GQZ}([a],k)$ (prüft, ob $a \geq 0$) können einfach von LTZ abgeleitet werden. Gleiches gilt für für das Protokoll

$$\mathsf{Signum}([a],k) := 1 - 2 \cdot \mathsf{LTZ}([a],k), \tag{2.10}$$

das das Vorzeichen von $[a]$ berechnet.

2.2.3 Erzeugung von Bit-Shares über $\mathbb{F}_{q_1}$

Die Erzeugung von Sharings zufälliger Bits geschieht mit Hilfe eines Shamir-Secret Sharing Schemes über einem „kleinen" Körper $\mathbb{F}_{q_1}$ für eine Primzahl q_1 mit einer Bitlänge r von ≤ 256 Bits ([CdH10a]). Im Anschluss daran werden die Sharings der Bits dann in Sharings über dem Körper $\mathbb{F}_q$, in dem die eigentliche Berechnung stattfindet, transformiert ([CDI05],[Sec09]). Dies kann unabhängig von allen anderen Operationen und für *alle* in einem Protokoll benötigten Zufallsbits parallel durchgeführt werden, z.B. in einer Pre-Computation-Phase. Zusätzlich zu dieser Parallelisierungsmöglichkeit ist die Verwendung von $\mathbb{F}_{q_1}$ im Vergleich mit einer direkten Erzeugung in $\mathbb{F}_q$ deutlich effizienter: Für die Erzeugung eines zufälligen Bits wird eine sichere Multiplikation über $\mathbb{F}_{q_1}$ benötigt und für die Transformation von $\mathbb{F}_{q_1}$ nach $\mathbb{F}_q$ eine weitere ([CdH10a], [Sec09]). Geht man davon aus, dass eine sichere Multiplikation höchstens linear in der Bitlänge ist ([Lor09],[LW11]), kann dies bei einer deutlich kürzeren Bitlänge von q_1 im Vergleich mit q einen wesentlichen Gewinn an Effizienz bedeuten, insbesondere dann, wenn die zufälligen Bits im Voraus berechnet werden. Als Nachteil fällt nur eine zusätzliche Kommunikationsrunde an.

2.2.4 Verbesserungen einiger elementarer Protokolle

Bei den Betrachtungen der Komplexitäten der hier vorgestellten Algorithmen liegen im Allgemeinen die in [CS10], [CdH10a], [CdH10b], [Sec10] und [Sec09] genannten Werte zu Grunde. Jedoch sind diese nicht in allen Fällen konsistent. In vielen Situationen müssen eine ganze Reihe zufälliger Bits $[b_0],\ldots,[b_{l-1}]$ erzeugt werden, die dann zu einer Zahl

$$\sum_{i=0}^{l-1} 2^i [b_i] \tag{2.11}$$

zusammengesetzt werden. Ist $l < \lfloor \log_2 q_1 \rfloor$ kann die Umwandlung auch erst nach der Zusammensetzung erfolgen. In [CS10], [CdH10a] und [CdH10b] ist dies nicht an allen Stellen hinreichend berücksichtigt. Insbesondere Protokoll TruncPr lässt sich auf diese Weise beschleunigen: In der in [CdH10a] dargestellten Version werden die zufälligen Bits über dem Körper $\mathbb{F}_q$ erzeugt.[7] Führt man dies anstattdessen über $\mathbb{F}_{q_1}$ durch und wandelt die Summe (2.11) *danach* um, so fällt zwar eine zusätzliche sichere Multiplikation an, allerdings sind die Gesamtkosten in Höhe von $m+1$ Multiplikationen über $\mathbb{F}_{q_1}$ wesentlich geringer als die sonst anfallenden Kosten in Höhe von m Multiplikationen über $\mathbb{F}_q$. Eine ähnliche Vereinfachung gilt für Protokoll $\mathsf{LTZ}([a],[b],k)$. Offensichtlich haben diese Modifikationen keine Auswirkungen auf die Sicherheit. Die korrigierten Werte für die wichtigsten Protokolle sind in Tabelle 2.1 angegeben.

2.2.5 Bedeutung des Körpers $\mathbb{F}_{2^8}$

Aus Tabelle 1.2 ist ersichtlich, dass XOR-Operationen von Sharings über dem Körper $\mathbb{F}_{2^8}$ durch lokale Addition und *ohne* Aufruf des Multiplikationsprotokolls möglich sind. Aus diesem Grund transformieren Protokolle wie Norm ([CS10]), die diese Operation benötigen, Shares über $\mathbb{F}_q$ vor der Berechnung in Shares über $\mathbb{F}_{2^8}$, führen dort die Berechnung durch und transformieren das Ergebnis wieder zurück.

2.3 Erweiterungen und Variationen

2.3.1 Invertieren eines Elements

Sollen in einem Algorithmus mehrere Zahlen $[a_1],\ldots,[a_n]$ durch ein und denselben Divisor $[b]$ dividiert werden, ist es nicht sinnvoll $\mathsf{FPDiv}([a_1],[b],k,f),\ldots,\mathsf{FPDiv}([a_n],[b],k,f)$ zu berechnen. Es ist effizienter, einmal $[z] = \frac{1}{[b]}$ zu berechnen und diesen Wert nacheinander mit $[a_1],\ldots,[a_n]$ zu multiplizieren. Eine Möglichkeit $[z]$ zu berechnen ist, mit Hilfe von Newton-Raphson Iterationen eine Nullstelle der Funktion $f(x) = b - \frac{1}{x}$ zu bestimmen. Auf dieser Idee basiert Protokoll $\mathsf{Invert}([x],k,f)$. Es ist eine leichte Abwandlung des Protokolls $\mathsf{DivNR}([a],[b],k,f)$ aus [CdH10b]. Der Unterschied besteht darin, dass Letzteres den Quotienten $\left[\frac{a}{b}\right]$ berechnet. Eine weitere Variation ist Protokoll $\mathsf{InvertNoTrunc}([x],k)$. Die letzte Operation von $\mathsf{Invert}([x],k,f)$ besteht in $\mathsf{TruncPr}([z],2k-f,2k-2f)$. Ist die auf $\mathsf{Invert}([x],k,f)$ folgende Operation eine

[7] In der Version aus [CS10] werden die Bits einzeln erzeugt und umgewandelt. Dies ist ebenfalls ineffizient.

Protokoll	Runden	Multiplikationen	Körper
TruncPr	1	1	$\mathbb{F}_q$
	2	$f+1$	$\mathbb{F}_{q_1}$
FPDiv	$2\theta_{Div}+10$	$4\theta_{Div}+8$	$\mathbb{F}_q$
	2	$8k+2\theta\cdot(2f+1)-3f+1$	$\mathbb{F}_{q_1}$
	$3\lceil\log_2 k\rceil+2$	$1,5\lceil\log_2 k\rceil+3(k-1)$	$\mathbb{F}_{2^8}$
DivNR	$7+3\theta_{DivNR}$	$3\theta_{DivNR}+8$	$\mathbb{F}_q$
	2	$\theta_{DivNR}\cdot(2k+1)+8k-f+1$	$\mathbb{F}_{q_1}$
	$3\lceil\log_2 k\rceil+1$	$3(k-1)+1,5k\lceil\log_2 k\rceil$	$\mathbb{F}_{2^8}$
InvertNoTrunc	$7+3\theta_{DivNR}$	$3\theta_{DivNR}+6$	$\mathbb{F}_q$
	2	$\theta_{DivNR}\cdot(2k+1)+6k$	$\mathbb{F}_{q_1}$
	$3\lceil\log_2 k\rceil+1$	$3(k-1)+1,5k\lceil\log_2 k\rceil$	$\mathbb{F}_{2^8}$
Trunc	1	1	$\mathbb{F}_q$
	2	$2(f+1)$	$\mathbb{F}_{q_1}$
	$\lceil\log_2 f\rceil+1$	$2(f-1)+1$	$\mathbb{F}_{2^8}$
EQZ	1	1	$\mathbb{F}_q$
	$\lceil\log_2 k\rceil+2$	$2k$	$\mathbb{F}_{q_1}$
LTZ	1	1	$\mathbb{F}_q$
	2	$2k$	$\mathbb{F}_{q_1}$
	$\lceil\log_2(k-1)\rceil+1$	$2k-3$	$\mathbb{F}_{2^8}$
EQZappr	3	3	$\mathbb{F}_q$
	2	$2k+f+1$	$\mathbb{F}_{q_1}$
	$\lceil\log_2(k-1)\rceil+1$	$2k-3$	$\mathbb{F}_{2^8}$
Mod2	1	1	$\mathbb{F}_q$
	2	2	$\mathbb{F}_{q_1}$

Tabelle 2.1: Protokolle der Fixpunktarithmetik aus [CS10] [CdH10a] sowie leichten Abwandlungen

Multiplikation mit einer Fixpunktzahl, kann auf die dann zwischengeschaltete TruncPr-Operation verzichtet werden und diese nach der zweiten Multiplikation nachgeholt werden, was einen leichten Effizienzgewinn bedeutet (Die Kosten von TruncPr betragen 1 Multiplikation in $\mathbb{F}_q$ und $f+1$ Multiplikationen in $\mathbb{F}_{q_1}$, d.h. es ist billiger, einmal die doppelte Länge abzuschneiden als zweimal die einfache). Voraussetzung dafür ist natürlich, dass auch Fixpunktzahlen der Länge $2k$ über $\mathbb{F}_q$ dargestellt werden können, ohne eine Modulo-Reduktion hervorzurufen.

2.3.2 EQZ

Das Protokoll EQZ aus [CdH10a] muss für einen Einsatz in der Praxis leicht variiert werden. In Zeile 2 soll die Summe $2^{k-1} + [a] + 2^k[r''] + [r']$ offengelegt werden. Es wird argumentiert, dass – modulo 2^k betrachtet – dieser Ausdruck genau dann r' ist, wenn $a \mod 2^k \equiv 0$. Offensichtlich ist dies so nicht richtig. Für die Gültigkeit muss der Summand 2^{k-1} weggelassen werden. Außerdem erscheint das Protokoll KOrCL, Teil von EQZ, aus [CdH10a] nicht sicher, da in Schritt 2 die $[a_i]$ mit den höheren Indizes bekannt werden können. Dieses Sub-Protokoll wurde deswegen durch das Standard-Verfahren KOR ([Sec09], dort KOpL) ersetzt, das rekursiv immer zwei benachbarte Elemente zusammenfasst. Die genauen Kosten des angepassten Protokolls sind in Tabelle 2.1 zu sehen.

2.3.3 Min

Das Minimum von 2 Elementen kann durch Protokoll 2.1 bestimmt werden. Zusätzlich zum Minimum liefert es noch ein geheimes Bit zurück, das anzeigt, welcher der beiden Werte das Minimum annimmt (bei Gleichheit den zweiten). Dies ist ohne zusätzliche Kosten möglich. Die anfallenden Kosten sind die von LTZ plus zwei sichere Multiplikationen in einer Kommunikationsrunde.

2.3.4 Vergleiche unter Toleranzen

An mehreren Stellen in den später vorgestellten Algorithmen müssen geheime Werte mit Konstanten verglichen werden. Grundlegende Protokolle hierfür sind LTZ und EQZ. Um den Algorithmus nicht auf Grund von Rundungsfehlern falsche Entscheidungen treffen zu lassen, werden die Operatoren $\overset{?}{<}$ und $\overset{?}{=}$ unter *Toleranzen* betrachtet, d.h. kleine Abweichungen sind zulässig. Sei $\varepsilon > 0$ die Toleranzschranke, dann ist Protokoll $\mathsf{LTZappr}([a], \varepsilon, k) := \mathsf{LTZ}([a] - \varepsilon, k)$ und EQZappr wie in Protokoll 2.2 definiert. Für die Komplexität siehe Tabelle 2.1.

Protokoll 2.1: $([a], [b]) \leftarrow \mathsf{Min}([x], [y], k)$

1 $[b] \leftarrow \mathsf{LT}([x], [y], k)$ //vgl. Tabelle 2.1
2 $[a] \leftarrow [b] \cdot [x] + (1 - [b]) \cdot [y]$ //2 Mul, 1 Rnd $\mathbb{F}_q$
3 **Return** $([a], [b])$

Abbildung 2.1: Bestimmung des Minimums zweier Elemente mit Rückgabe eines Bits $[b]$, das den Wert $[1]$ annimmt, wenn $[x] < [y]$ und $[0]$ sonst.

Protokoll 2.2: $[x] \leftarrow \mathsf{EQZappr}([x], \varepsilon, k, f)$

1 $[y] \leftarrow [x] \cdot [x]$ //1 Mul $\mathbb{F}_q$
2 $[y] \leftarrow \mathsf{TruncPr}([y], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3 $[y] \leftarrow \mathsf{LT}([y], \varepsilon, k)$ //vgl. Tabelle 2.1
4 **Return** $[y]$

Abbildung 2.2: Approximativer Vergleich mit 0

2.4 Sichere Berechnung der Wurzelfunktion

Das Folgende ist eine leichte Bearbeitung eines bereits erschienenen Artikels ([Lie12]) des Autors. Beschrieben wird eine Erweiterung der sicheren, verteilten Fixpunktarithmetik von Catrina et al. zur sicheren verteilten Berechnung der Quadratwurzel. Der Algorithmus basiert hauptsächlich auf Goldschmidts Algorithmus zur Berechnung der Quadratwurzel ([Gol64]), da dieser - im Gegensatz zu Newton-Raphson Iterationen - stärker parallelisierbar ist. Da ersterer aber nicht selbstkorrigierend ist, wird als letzte Iteration ein Newton-Raphson Schritt durchgeführt, um aufgelaufene Rundungsfehler zu korrigieren ([Mar04]). Der Startwert - korrekt in den ersten 5,4 Bits - wird durch lineare Approximation berechnet.

2.4.1 Mathematischer Hintergrund

Beide Verfahren - der Goldschmidt-Algorithmus und die Newton-Raphson-Iterationen - approximieren den gesuchten Wert iterativ und konvergieren quadratisch, d.h. die Anzahl der korrekten Stellen verdoppelt sich im Schnitt in jeder Iteration.

2.4.1.1 Iterationen nach Newton-Raphson zur Berechnung der Quadratwurzel

Ziel ist, mit Hilfe einer Newton-Raphson-Iteration ([SB02])) eine Annäherung für $R = \frac{1}{\sqrt{x}}$ zu bestimmen. Man verwendet dafür die Funktion

$$f(R) = \frac{1}{R^2} - x \tag{2.12}$$

und erhält so die Iterationsvorschrift

$$R_{j+1} = \frac{1}{2} \cdot R_j \cdot \left(3 - x \cdot R_j^2\right). \tag{2.13}$$

Am Ende multipliziert man das Ergebnis noch mit x und erhält $\sqrt{x}$. Ist ein hinreichend genauer Startwert bekannt, konvergiert der Algorithmus quadratisch.

Algorithmus 2.3: Goldschmidts Quadratwurzel-Algorithmus

```
1  While |g_i − g_{i−1}| > ε
2      r_{i−1} = 1/2 − g_{i−1} · h_{i−1}
3      g_i = g_{i−1} · (1 + r_{i−1})
4      h_i = h_{i−1} · (1 + r_{i−1})
```

Abbildung 2.3: Goldschmidts Quadratwurzel-Algorithmus mit Startwert y_0

2.4.1.2 Der Algorithmus von Goldschmidt

Ist $x > 0$ die Zahl deren Radikand gesucht ist, so approximiert Goldschmidts Algorithmus ([Gol64]) - auch bekannt als „Multiplicative Normalization Method“ ([EL04]) - iterativ $\sqrt{x}$ und $\frac{1}{\sqrt{x}}$ (es handelt sich eigentlich um eine Spielart einer Newton-Raphson Iteration (2.4.1.1); man zeigt dies mit Hilfe der ursprünglichen Definition aus [Mar04]. Insbesondere gelten also auch die Konvergenzeigenschaften von 2.4.1.1). Die Software-freundliche Version ist in Abb. 2.3 dargestellt. Ein Startwert y_0 von $\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{x_0}}$, d.d.

$$\frac{1}{2} < x_0 \cdot y_0^2 < \frac{3}{2} \tag{2.14}$$

wird vorausgesetzt. Dabei sei $g_0 = x_0 \cdot y_0$ und $h_0 = \frac{y_0}{2}$ und die Iterierten $\sqrt{x}$ und $\frac{1}{2\sqrt{x}}$ seien g_i und h_i. Man beachte, dass die Multiplikationen in den Zeilen 3 und 4 unabhängig voneinander sind.

2.4.1.3 Berechnung des Startwerts

Der Startwert wird in zwei Schritten berechnet: Zunächst wird der Eingabewert x_0 auf das Intervall $[\frac{1}{2}, 1[$ normiert. Dies liefert $x_{normiert}$. In einem zweiten Schritt wird dieser Wert verwendet, um mit Hilfe der affin-linearen Funktion

$$L(x) = \alpha \cdot x + \beta. \tag{2.15}$$

eine lineare Approximation von $\frac{1}{\sqrt{x_{normiert}}}$ zu berechnen. Dies geschieht derart, dass der resultierende Wert eine gute Approximation von $\frac{1}{\sqrt{x_0}}$ ist (vgl. Abschnitt 2.4.2)! Die Koeffizienten α und β können berechnet werden, indem die relative Fehlerfunktion

$$E(x) = \frac{\alpha \cdot x + \beta - \frac{1}{\sqrt{x}}}{\frac{1}{\sqrt{x}}} \tag{2.16}$$

minimiert wird. Die Differenzierung von E liefert ein Maximum bei $x_{max} = -\frac{1}{3} \cdot \frac{\beta}{\alpha}$. Auswertung an x_{max} ergibt

Protokoll 2.4: $([c],[v],[m],[W]) \leftarrow \mathsf{NormSQ}([x],k,f)$

1 $\left([x_{k-1}]^{\mathbb{F}_2^8},\ldots,[x_0]^{\mathbb{F}_2^8}\right) \leftarrow \mathsf{BitDec}([x],k,k)$ //vgl. [CS10]
2 $\left([y_{k-1}]^{\mathbb{F}_2^8},\ldots,[y_0]^{\mathbb{F}_2^8}\right) \leftarrow \mathsf{PreOR}\left([x_{k-1}]^{\mathbb{F}_2^8},\ldots,[x_0]^{\mathbb{F}_2^8}\right)$ //vgl. [CS10]
3 **For** $i \in [0,\ldots,k-1]$ **do parallel**
4 $\quad [y_i] \leftarrow \mathsf{BitF2MtoZQ}\left([y_i]^{\mathbb{F}_{2^8}}\right)$ //vgl. [Sec09]
5 **For** $i \in [0,\ldots,k-2]$
6 $\quad [z_i] \leftarrow [y_i] - [y_{i+1}]$
7 $[z_{k-1}] \leftarrow [y_{k-1}]$
8 $\overrightarrow{[W]} \leftarrow \mathsf{HalfIndexEven}\left(\overrightarrow{[z]},k\right)$ //Protokoll 2.5
9 $\overrightarrow{[W_2]} \leftarrow \mathsf{HalfIndexOdd}\left(\overrightarrow{[z]},k\right)$ //analog Protokoll 2.5
10 $\overrightarrow{[W]} \leftarrow \overrightarrow{W} + \overrightarrow{W_2}$
11 $[w] \leftarrow \sum_{i=0}^{\frac{k}{2}} 2^i \cdot [W_i]$
12 $[m] \leftarrow \sum_{i=0}^{k-1} 2^i \cdot [z_i]$
13 $[v] \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1}[z_i]$
14 $[c] \leftarrow [x][v]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
15 **Return**$([c],[v'],[m],[W])$

Abbildung 2.4: Modifiziertes Protokoll NormSQ

$$M := E(x_{max}) = \frac{\sqrt{3}}{3} \cdot \sqrt{\frac{-\beta}{\alpha}} \cdot \left(\frac{2}{3} \cdot \beta - \frac{\sqrt{3}}{\sqrt{\frac{-\beta}{\alpha}}}\right). \tag{2.17}$$

Einsetzen in E und Auflösen des Gleichungssystems

$$E\left(\frac{1}{2}\right) = -M \tag{2.18}$$
$$E(1) = -M \tag{2.19}$$

nach α und β liefert $\alpha = -0,8099868542$ und $\beta = 1,787727479$. Der relative Fehler der so berechneten linearen Approximation von $\frac{1}{\sqrt{x}}$ für $\frac{1}{2} \leq x < 1$ übersteigt so nicht $0,0222593752$. Dies bedeutet, dass der Startwert bis auf fast $5,5$ Bit exakt ist.

Protokoll 2.5: $\overrightarrow{[W]} \leftarrow \mathsf{HalfIndexEven}\left(\overrightarrow{[z]}, l\right)$

For $(i = 2, 4, \ldots, l)$
  $\left[z_{\frac{i}{2}}\right] \leftarrow [z_i]$
  $[z_i] \leftarrow \mathsf{PRSZ}$
For $(i = \frac{l}{2} + 1, \frac{l}{2} + 3, \ldots, l)$
  $[z_i] \leftarrow \mathsf{PRSZ}$
Return $\overrightarrow{[z]}$

Abbildung 2.5: Berechnung von $\left[2^{\frac{m}{2}}\right]$ für gerades m und l

2.4.2 Beschreibung und Analyse der Algorithmen

2.4.2.1 Norm

Protokoll Norm aus [CS10] liefert Werte $2^{k-1} \leq [c] < 2^k$ und $[v]$, d.d. $[x] \cdot [v] = [c]$. Ist m so dass $2^{m-1} \leq [x] < 2^m$, dann ist

$$[v] = \left[2^{k-m}\right]. \tag{2.20}$$

Norm wird so modifiziert, dass $[m]$ und $[W] = [2^{\frac{m}{2}}]$ zurückgeliefert werden, wenn m gerade ist und $[W] = [2^{\frac{m-1}{2}}]$, wenn m ungerade ist. Dies kann ohne zusätzliche Kommunikation geschehen. Die Berechnung von $[2^{\frac{m}{2}}]$ (für m gerade, sonst wird 0 zurückgegeben) ist in Abb. 2.5 zu sehen. Das Ergebnis ist in Form eines binären Vektors. Beachte, dass im Protokoll NormSQ (Protokoll refNormSQ) - im Unterschied zu [CS10] - auf die Berechnung des Vorzeichens verzichtet wird, da der Radikand als positiv vorausgesetzt wird.

2.4.2.2 Approximation

Korrektheit: Sei m gerade. Nach Auswertung der linear approximierenden Funktion (2.15) erhält man

$$\alpha \cdot 2^k \cdot [c] + \beta \cdot 2^{2k}. \tag{2.21}$$

Multiplikation mit $[v] = [2^{k-m}]$ (cf. (2.20)) liefert

$$\alpha \cdot 2^{2k-m} \cdot [c] + \beta \cdot 2^{3k-m}. \tag{2.22}$$

Aber da $[c]$ nur ein Secret-Sharing von $\frac{\bar{x}}{2^m} \cdot 2^k = \frac{x \cdot 2^f}{2^m} \cdot 2^k$ ist, gleicht dies[8]

[8] aus Gründen der Einfachheit werden von hier ab die Klammern weggelassen

Protokoll 2.6: $[w] \leftarrow \mathsf{LinAppSQ}([b], k, f)$

1 $\alpha \leftarrow \mathsf{fld}_k(-0.8099868542)$
2 $\beta \leftarrow \mathsf{fld}_{2k}(1.787727479)$
3 $([c], [v], [m], [W]) \leftarrow \mathsf{NormSQ}([b], k, f)$ //Protokoll 2.4
4 $[w] \leftarrow \alpha[c] + \beta$
5 $[m] \leftarrow \mathsf{Mod2}([m], \lceil \log_2 k \rceil)$ //vgl. Tabelle 2.1
6 $[w] \leftarrow [w] \cdot [W] \cdot [v]$ //2 Mul $\mathbb{F}_q$, 2 Rnd
7 $[w] \leftarrow \mathsf{TruncPr}([w], 3k, 3k-2f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
8 $[w] \leftarrow \mathsf{DivKonst}\left([w], 2^{\frac{f}{2}}, k, f\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
9 $[w] \leftarrow (1-[m]) \cdot [w] \cdot 2^f + \left(\sqrt{2} \cdot 2^f\right) \cdot [m] \cdot [w]$ // 2 Mul, 1 Rnd $\mathbb{F}_q$
10 $[w] \leftarrow \mathsf{TruncPr}([w], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
11 **Return** $[w]$

Abbildung 2.6: Lineare Approximation von $\frac{1}{\sqrt{x}}$

Protokoll	Sichere Multiplikationen	Runden	Körper
LinAppSqr	$k \cdot \left(\frac{3}{2}l + 1\right) + 11$	9	$\mathbb{F}_q$
	$5k + 1$	2	$\mathbb{F}_{q_1}$
	k	$2l + 1$	$\mathbb{F}_{2^8}$
SQR	$k \cdot \left(\frac{3}{2}l + 1\right) + 6\theta + 12$	$4\theta + 14$	$\mathbb{F}_q$
	$3f \cdot (\theta + 1) + 5k + 1$	2	$\mathbb{F}_{q_1}$
	k	$2l + 1$	$\mathbb{F}_{2^8}$

Tabelle 2.2: Komplexität der Protokolle. Die Bitlänge von k sei eine Potenz von 2, e.g. $k = 2^l$. Alle Vektoren haben Länge n und alle Matrizen seien quadratisch mit n Zeilen und n Spalten.

$$\alpha \cdot 2^{3k-m} \cdot \frac{x \cdot 2^f}{2^m} + \beta \cdot 2^{3k-m}. \tag{2.23}$$

Schneidet[9] man $3k - 2f$ Bits ab, so erhält man

$$2^{2f-m} \cdot \left(\alpha \cdot \frac{x \cdot 2^f}{2^m} + \beta\right), \tag{2.24}$$

was nichts anderes ist, als eine lineare Approximation des normalisierten Werts

[9] Rundungsfehler werden vernachlässigt; aus Gründen der Berechenbarkeit ist die Reihenfolge in Protokoll 2.6 leicht abgewandelt

$\frac{x \cdot 2^f}{2^m}$ von $[x]$, der mit dem Faktor 2^{2f-m} skaliert ist. Dies bedeutet, dass (2.24) gleichbedeutend ist mit

$$K \cdot 2^{2f-m} \cdot \frac{1}{\sqrt{\frac{x \cdot 2^f}{2^m}}} = \frac{2^{\frac{3f}{2}}}{2^{\frac{m}{2}}} \cdot K \cdot \frac{1}{\sqrt{x}}. \tag{2.25}$$

Dabei ist K ein Faktor ≈ 1, abhängig von der Approximation. Multiplikation mit $[W] = 2^{\frac{m}{2}}$ und Division durch $2^{\frac{f}{2}}$ liefern also eine Annäherung an $\frac{1}{\sqrt{x}}$ skaliert mit 2^f, was genau das Gesuchte ist. Ist m ungerade (zur Unterscheidung zwischen gerade und ungerade wird Protokoll Mod2 aus [CdH10a] verwendet), liefert die Funktion NormSQ den Wert $[W] = 2^{\frac{m-1}{2}}$ zurück. Also liefert die Multiplikation mit $[W] \cdot 2^{\frac{f}{2}}$ nur

$$K \cdot \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{x}}. \tag{2.26}$$

zurück. Multipliziere in diesem Fall die Gleichung zusätzlich mit $\sqrt{2}$ und erhalte das gewünschte Ergebnis.

Komplexität: Die Kosten werden dominiert von Protokoll NormSQ und - zu einem geringeren Grad - von Protokoll TruncPr($[w], 3k, 3k-2f$). Alle anderen Schritte sind nur für eine geringe Anzahl an Multiplikationen verantwortlich. Die Komplexität von LinAppSQ ist in Tabelle 2.2 abgebildet.

2.4.2.3 Der Algorithmus von Goldschmidt

Für eine Annäherung $[y_0]$, die (2.14) erfüllt, muss nur noch Algorithmus 2.3 in einen sicheren Mehrparteienalgorithmus umgewandelt werden (Protokoll 2.7).

Im Gegensatz zu Algorithmus 2.3 wird die Anzahl der Iterationen auf $\theta = \left\lceil \log_2 \left(\frac{k}{5.4} \right) \right\rceil$ fixiert, was eine Genauigkeit des Ergebnisses von k Bits sicherstellt. Beachte, dass die letzte Iteration (Zeilen 19-23) durch eine Newton-Raphson-Iteration ersetzt wurde, da diese - im Gegensatz zu Goldschmidt-Iterationen - selbstkorrigierend ist und so akkumulierte Rundungsfehler eliminiert werden können (abgesehen vielleicht vom letzten Bit, das - aufgrund der Inexaktheit der probabilistischen Rundung - falsch sein kann). Da $[g]$ und $[gh]$ zu diesem Zeitpunkt nicht mehr benötigt werden, ist bereits die letzte Goldschmidt Iteration (Zeilen 16-18) verkürzt. In dieser werden sie nicht mehr berechnet. Die Berechnung von $[g]$ und $[h]$ kann in der Schleife parallelisiert werden. Die Komplexität ist wieder in Tabelle 2.2 zu sehen. Die Unterschiede zu den Daten in [Lie12] bestehen im Einsatz verbesserter Sub-Protokolle (z.B. von LTZ), insbesondere aus [Sec09].

Protokoll 2.7: $[g] \leftarrow \mathsf{SQR}([x], k, f)$

1 $\theta \leftarrow \left\lceil \log_2 \left(\frac{k}{5.4} \right) \right\rceil$
2 $[y_0] \leftarrow \mathsf{LinAppSQ}([x], k, f)$ //Protokoll 2.6
3 $[g_0] \leftarrow [y_0] \cdot [x]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
4 $[g_0] \leftarrow \mathsf{TruncPr}([g_0], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul, 2 Rnd $\mathbb{F}_{q_1}$
5 $[h_0] \leftarrow \mathsf{DivKonst}([g_0], 2)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
6 $[gh] \leftarrow [g_0] \cdot [h_0]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
7 $[gh] \leftarrow \mathsf{TruncPr}([gh], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
8 **For** $i = 1, \ldots, \theta - 2$
9 $\quad [r] \leftarrow \frac{3}{2} \cdot 2^f - [gh]$
10 $\quad [g] \leftarrow [g] \cdot [r]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
11 $\quad [h] \leftarrow [h] \cdot [r]$ //1 Mul $\mathbb{F}_q$
12 $\quad [g] \leftarrow \mathsf{TruncPr}([g], k, f)$//1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
13 $\quad [h] \leftarrow \mathsf{TruncPr}([h], k, f)$//1 Mul $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
14 $\quad [gh] \leftarrow [g] \cdot [h]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
15 $\quad [gh] \leftarrow \mathsf{TruncPr}([gh], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
16 $[r] \leftarrow \frac{3}{2} \cdot 2^f - [gh]$
17 $[h] \leftarrow [h] \cdot [r]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
18 $[h] \leftarrow \mathsf{TruncPr}([h], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
19 $[H] \leftarrow (2 \cdot [h])^2$
20 $[H] \leftarrow [H] \cdot [x]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
21 $[H] \leftarrow \left(3 \cdot 2^{2f}\right) - [H]$
22 $[H] \leftarrow [h] \cdot [H]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
23 $[g] \leftarrow [H] \cdot [x]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
24 $[g] \leftarrow \mathsf{DivKonst}([g], 2)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
25 $[g] \leftarrow \mathsf{TruncPr}([g],4k,4f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $4f$+1 Mul $\mathbb{F}_{q_1}$
26 **Return** $[g]$

Abbildung 2.7: Sichere Version von Goldschmidts Algorithmus zur Berechnung der Quadratwurzel

2.4.2.4 Sicherheit

Alle Protokolle setzen sich aus Bausteinen zusammen, deren Sicherheit - entweder perfekt oder statistisch - bewiesen ist ([CdH10a],[CdH10b],[CS10]). Informationen werden nicht veröffentlicht. Demzufolge ([CDD+04],[Can00],[Can01]) sind die Protokolle sicher im passiven Sicherheitsszenario.

3 Sichere Vektor- und Matrizenrechnung

In diesem Kapitel sollen Algorithmen zum sicheren Rechnen mit Vektoren und Matrizen von Sharings vorgestellt werden. Ein Vektor bestehend aus Sharings wird mit $\overrightarrow{[x]}$ und eine Matrix bestehend aus Sharings mit $[[A]]$ bezeichnet. Für sichere Matrix- und Vektoroperationen spielt es keine Rolle, ob die Sharings ganze Zahlen oder Fixpunktzahlen repräsentieren. Jedoch muss nach einer Matrix-Matrix oder einer Matrix-Vektor-Operation zweier Objekte bestehend aus Fixpunktzahlen die Operation $\mathsf{TruncPr}$ für jedes Element zum Abschneiden von f Nachkommastellen durchgeführt werden. Besteht eine Matrix oder ein Vektor *nicht* aus Sharings von Fixpunktzahlen, dann wird dadurch i.d.R. eine Struktur wie das geheime Vertauschen von Matrixzeilen (Abschnitt 3.2.3) oder der sichere Zugriff auf Vektorelemente (Abschnitt 3.1.2) repräsentiert.

3.1 Vektorrechnung

Soll in einem Protokoll explizit ein Vektor der Länge n konstruiert werden, wird die Schreibweise

$$\overrightarrow{[v]} \leftarrow \text{Vektor}(n) \tag{3.1}$$

verwendet.

3.1.1 Skalarprodukt

Für die Berechnung eines Skalarprodukts zweier Vektoren $\overrightarrow{[v]}$ und $\overrightarrow{[w]}$ der Länge n wird mit dem Algorithmus von [CdH10a] nur eine sichere Multiplikation benötigt. Grundlegende Idee dabei ist – ähnlich wie beim Multiplikationsprotokoll – zunächst die lokalen Produkte zu bilden und zu addieren, diese dann zu verteilen und dann aus den erhaltenen Shares den eigenen Anteil am Skalarprodukt zu rekonstruieren. Das entsprechende Protokoll wird in der vorliegenden Arbeit mit $\mathsf{Skalar}\left(\overrightarrow{[v]}, \overrightarrow{[w]}, n\right)$, $\mathsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[w]}, n\right)$ oder $\langle \overrightarrow{[v]}, \overrightarrow{[w]} \rangle$ bezeichnet.

Protokoll 3.1: $[z] \leftarrow \textbf{VektorNorm}\left(\overrightarrow{[v]}, n\right)$

1 $[z] \leftarrow \mathsf{Inner}(\overrightarrow{[v]}, \overrightarrow{[v]}, n)$ //1 Mul $\mathbb{F}_q$, 1 Rnd
2 $[z] \leftarrow \mathsf{TruncPr}([z], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3 $[z] \leftarrow \mathsf{SQR}([z], k, f)$ //Protokoll 2.7
4 **Return** $[z]$

Abbildung 3.1: Berechnung der euklidischen Norm eines Vektors

3.1.2 Sicherer Zugriff/Schreiben auf/von Vektorelemente(n)

Es sei $\overrightarrow{[v]}$ ein Vektor der Länge n. Der sichere Zugriff auf ein Element, das in Form eines binären Vektors[1] $\overrightarrow{[I]}$ vorliegt, lässt sich mit Hilfe eines Skalarprodukts realisieren:

$$[x] \leftarrow \mathsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[I]}, n\right) \tag{3.2}$$

Anstelle von Inner schreibt man in dieser Situation auch häufig SecRead ([CdH10b]). Die Kosten betragen eine sichere Multiplikation. Soll andererseits in den Vektor $\overrightarrow{[v]}$ an der in $\overrightarrow{[I]}$ kodierten Stelle ein neuer Wert $[x]$ eingefügt werden, so kann dies umgesetzt werden durch:

$$\overrightarrow{[v]} \leftarrow [x] \cdot \overrightarrow{[I]} + \mathsf{MulZeilenweise}\left(\overrightarrow{1} - \overrightarrow{[I]}, \overrightarrow{[v]}, n\right) \tag{3.3}$$

Man schreibt dafür auch $\mathsf{SecWrite}\left(\overrightarrow{[v]}, \overrightarrow{[I]}, [x], n\right)$. Die Kosten betragen $2n$ sichere Multiplikationen.

3.1.3 Norm eines Vektors

An verschiedenen Stellen wird die (euklidische) Norm eines Vektors der Länge n benötigt. Protokoll 3.1 implementiert die Standardformel.

3.1.4 Vergleich eines Vektors mit dem Nullvektor

Ein Vektor wird mit dem Nullvektor verglichen, indem jeder Eintrag quadriert wird und die Summe mit Null verglichen wird. Handelt es sich um Fixpunktzahlen, ist es zweckmäßiger die Summe nicht mit 0 zu vergleichen, sondern

[1] Bei einem binären Vektor ist ein Eintrag ein Sharing von 1 und die anderen Einträge Sharings von 0.

Protokoll 3.2: $[B] \leftarrow \textsf{VektorGleichNull}\left(\overrightarrow{[v]}, n\right)$

1 $[B] \leftarrow \textsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[v]}, n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
2 $[B] \leftarrow \textsf{TruncPr}([B], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3 $[B] \leftarrow \textsf{LT}([h], \epsilon, k, f)$ //vgl. Tabelle 2.1
4 **Return** $[B]$

Abbildung 3.2: Überprüfen eines Vektors auf Gleichheit mit Null

zu überprüfen, ob eine Toleranzschwelle überschritten wird. Auf diese Weise lassen sich falsche Ergebnisse auf Grund von Rundungsfehlern vermeiden. Dies ist die Variante die in Abb. 3.2 beschrieben ist. Soll die exakte Gleichheit mit Null überprüft werden, muss nur das Protokoll LT in Zeile 4 durch das Protokoll EQZ ersetzt werden. Die Kosten betragen n Multiplikationen sowie einen Aufruf von LTZ (bzw. EQZ).

3.1.5 Bestimmung des Minimums eines Vektors

Der Index des Minimums eines Vektors wird rekursiv bestimmt, indem in jeder Iteration benachbarte Werte miteinander verglichen und die jeweiligen Minima in einem neuen Vektor gespeichert werden (Protokoll 3.3). Mit diesem ruft sich die Funktion erneut selbst auf. Ist die Länge des Vektors ungerade, wird die erste Komponente nicht betrachtet. Zurückgegeben wird ein Vektor, der die Position des Minimums binär kodiert. Der Index des Maximums des Vektors kann berechnet werden, indem vor der Ausführung jede Komponente mit -1 multipliziert wird.

Um die Komplexität von Protokoll 3.3 angeben zu können, betrachte zunächst die Funktion

$$\varphi : \mathbb{N} \to \mathbb{N}, \quad n \mapsto \begin{cases} \frac{n}{2} & n \text{ gerade} \\ \frac{n-1}{2} + 1 & n \text{ ungerade} \end{cases} . \tag{3.4}$$

Diese gibt die Länge des resultierenden Vektors nach einer Iterationsrunde an.

Satz 1. *Sei $n > 0$, $n \in \mathbb{N}$ und v ein Vektor der Länge n. Dann werden zur Berechnung des Minimums $n-1$ Vergleiche benötigt.*

Beweis: Verwende zum Beweis Induktion nach n: Die Behauptung ist offensichtlich korrekt für $n = 2, 3$. Sei nun $n > 3$. Dann existieren 2 Fälle:

Fall 1: $n+1$ gerade. Dann ist

Protokoll 3.3: $\overrightarrow{[I]} \leftarrow \mathsf{MinVektor}\left(\overrightarrow{[v]}, l, k\right)$

1 $r \leftarrow l \mod 2$
2 **If** $(l = 1)$
3 $\quad [I(1)] \leftarrow \mathsf{PRS}(1)$
4 $\quad$ **Return** $\overrightarrow{[I]}$
5 **If** $l > 1$
6 $\quad$ **If**$(r = 0)$
7 $\quad\quad d \leftarrow \frac{r}{2}$
8 $\quad$ **Else**
9 $\quad\quad d \leftarrow \frac{r+1}{2}$
10 $\quad$ **If**$(r = 0)$
11 $\quad\quad$ **For** $(i = 1, \ldots, d)$ **do parallel**
12 $\quad\quad\quad [B(i)] \leftarrow \mathsf{LT}([v(2i-1)], [v(2i)], k)$ //vgl. Tabelle 2.1
13 $\quad\quad\quad [A'(i)] \leftarrow [v(2i-1)] \cdot [B(i)] + (1 - [B(i)]) \cdot [v(2i)]$ //2 M., 1 R. $\mathbb{F}_q$
14 $\quad\quad \overrightarrow{[v]} \leftarrow \overrightarrow{[A']}$
15 $\quad$ **Else**
16 $\quad\quad$ **For** $(i = 2, \ldots, d)$ **do parallel**
17 $\quad\quad\quad [B(i)] \leftarrow \mathsf{LT}([v(2i-2)], [v(2i-1)], k)$ //vgl. Tabelle 2.1
18 $\quad\quad\quad [A'(i)] \leftarrow [v(2i-2)] \cdot [B(i)] + (1 - [B(i)]) \cdot [v(2i-1)]$ //2 M, 1 R $\mathbb{F}_q$
19 $\quad\quad \overrightarrow{[v(2:d)]} \leftarrow \overrightarrow{[A'(2:d)]}$
20 $\quad\quad [v(1)] \leftarrow [B(1)]$
21 $\quad \overrightarrow{[I]} \leftarrow \mathsf{MinVektor}\left(\overrightarrow{[v]}, d, k\right)$
22 $\quad$ **If**$(r = 0)$
23 $\quad\quad$ **For**$(i = 1, \ldots, l)$ **do parallel**
24 $\quad\quad\quad [I'(2i-1)] \leftarrow [B(i)] \cdot [I(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
25 $\quad\quad\quad [I'(2i)] \leftarrow [I(i)] - [I'(2i-1)]$
26 $\quad\quad \overrightarrow{[I]} \leftarrow \overrightarrow{[I']}$
27 $\quad$ **Else**
28 $\quad\quad$ **For**$(i = 2, \ldots, d)$ **do parallel**
29 $\quad\quad\quad [I'(2i-2)] \leftarrow [B(i)] \cdot [I(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
30 $\quad\quad\quad [I'(2i-1)] \leftarrow [I(i)] - [I'(2i-2)]$
31 $\quad\quad \overrightarrow{[I(2:d)]} \leftarrow \overrightarrow{[I'(2:d)]}$

Abbildung 3.3: Berechnung des Minimums eines Vektors

$$\varphi(n+1) = \frac{n+1}{2}.$$

D.h. ein Vektor der Länge $n+1$ verkürzt sich nach der ersten Rekursionsrun-

Protokoll 3.4: $\overrightarrow{[W]} \leftarrow \mathsf{MinBruch}\left(\overrightarrow{[Z]}, \overrightarrow{[N]}, l\right)$

1 **For** $(i = 1, \ldots, l)$
2 $\quad [D(i)] \leftarrow \mathsf{EQZ}([N(i)])$ // vgl. Tabelle 2.1
3 $\quad [N(i)] \leftarrow (1 - [D(i)]) \cdot [N(i)] + [D(i)] \cdot \epsilon$ // 1 Mul, 1 Rnd $\mathbb{F}_q$
4 $\overrightarrow{[W]} \leftarrow \mathsf{MinCons}\left(\overrightarrow{[Z]}, \overrightarrow{[N]}, l\right)$ //Protokoll 3.5
5 **Return** $[W]$

Abbildung 3.4: Berechnung des Minimums einer Folge von Brüchen mit vorheriger Korrektur des Nenners

de durch Anwendung von $\frac{n+1}{2}$ Vergleichen auf die Länge $\frac{n+1}{2}$. Nach Induktionsvoraussetzung werden für die Bestimmung des Minimums eines solchen Vektors $\frac{n-1}{2}$ Vergleiche benötigt. Insgesamt kann also das Minimum in n Vergleichen berechnet werden.

Fall 2: Sei $n+1$ ungerade. Dann hat der Vektor nach Anwendung von $\frac{n}{2}$ Vergleichen die Länge $\varphi(n+1) = \frac{n}{2} + 1$. Zu dessen Minimumsbestimmung sind jedoch $\frac{n}{2}$ Vergleiche nötig, insgesamt also wieder n. □

Satz 1 besagt also, dass für die Minimumsbestimmung eines Vektors der Länge n, insgesamt $n-1$ Auswertungen der Zeilen 12-13 und 24-25 bzw. 17-18 und 29-30 nötig sind. Die Zeilen 12 und 13 (bzw. 17 und 18) benötigen drei sichere Multiplikationen über $\mathbb{F}_q$, $2k$ sichere Multiplikationen über $\mathbb{F}_{q_1}$ und $2k-3$ sichere Multiplikationen über $\mathbb{F}_{2^8}$. Dazu kommt noch eine sichere Multiplikation in den Zeilen 24 bzw. 29. Die Gesamtkosten betragen also $4 \cdot (n-1)$ sichere Multiplikationen über $\mathbb{F}_q$, $2k \cdot (n-1)$ sichere Multiplikationen über $\mathbb{F}_{q_1}$ und $(2k-3) \cdot (n-1)$ sichere Multiplikationen in $\mathbb{F}_{2^8}$. Die Rundenkomplexität beträgt $3\lceil \log_2 n \rceil$ Runden in $\mathbb{F}_q$, 2 Runden in $\mathbb{F}_{q_1}$ und $\lceil \log_2 n \rceil \cdot (2k-3)$ Runden in $\mathbb{F}_{2^8}$.

3.1.6 Bestimmung des Minimums (Maximums einer Folge von Brüchen)

Zur Bestimmung von

$$\arg\min\left(\frac{[a_1]}{[b_1]}, \ldots, \frac{[a_n]}{[b_n]}\right) \tag{3.5}$$

wird Protokoll MinCons aus [CdH10b] verwendet. Elementar werden dabei zwei Werte $\frac{[a_1]}{[b_1]}$ und $\frac{[a_2]}{[b_2]}$ verglichen, indem der Wert der Relation

$$[a_1] \cdot [b_2] \stackrel{?}{=} [a_2] \cdot [b_1]$$

Protokoll	Runden	Multiplikationen	Körper
VektorNorm	$4\theta_{SQR}+16$	$13+6\theta_{SQR}$	$\mathbb{F}_q$
	4	$7k+4f+3\theta_{SQR}(f+1)$	$\mathbb{F}_{q_1}$
	$1+2\lceil\log_2 k\rceil$	$1,5k\lceil\log_2 k\rceil$+k	$\mathbb{F}_{2^8}$
FindFirst	$2k$	$2n$	$\mathbb{F}_q$
MinVektor	$3\lceil\log_2 n\rceil$	$4(n-1)$	$\mathbb{F}_q$
	2	$2k(n-1)$	$\mathbb{F}_{q_1}$
	$\lceil\log_2 n\rceil\cdot(2k-3)$	$(2k-3)\cdot(n-1)$	$\mathbb{F}_{2^8}$
MinBruch	$4\lceil\log_2 n\rceil+2$	$10n-7$	$\mathbb{F}_q$
	2	$6kn-4k$	$\mathbb{F}_{q_1}$
	$\lceil\log_2 n\rceil\cdot(2k-3)$	$(4k-3)\cdot(n-1)$	$\mathbb{F}_{2^8}$
MinCons	$4\lceil\log_2 n\rceil$	$8n-7$	$\mathbb{F}_q$
	2	$4k\cdot(n-1)$	$\mathbb{F}_{q_1}$
	$\lceil\log_2 n\rceil\cdot(4k-3)$	$(4k-3)\cdot(n-1)$	$\mathbb{F}_{2^8}$
Kompaktifiziere	m^2+2k+3	$3m^2+2m$	$\mathbb{F}_q$
	$2mk$	$2mk$	$\mathbb{F}_{q_1}$
Kompaktifiziere (binär)	m^2+2	$3m^2$	$\mathbb{F}_q$

Tabelle 3.1: Kosten der Protokolle der Vektorrechnung

berechnet wird. Dies ist dann Baustein in einem Protokoll, das den Index des Minimums eines Vektors berechnet (ähnlich wie in Protokoll 2.1). Voraussetzung ist, dass die Nenner immer $\neq 0$ sind und alle dasselbe Vorzeichen besitzen. Ist dies nicht gewährleistet, müssen sie ggf. vor dem Protokoll entsprechend angepasst werden, etwa indem Null-Nenner durch die kleinstmögliche Zahl ε ersetzt werden (Protokoll 3.4) bzw. indem etwaige Vorzeichen der Nenner in die Zähler multipliziert werden (nicht dargestellt). Die Komplexität wird analog zu Protokoll 3.3 berechnet. Die genauen Werte finden sich in Tabelle 3.1.

3.1.7 Kodieren einer Zahl in einem Vektor

Ein Protokoll, mit Hilfe dessen aus dem Sharing einer Zahl $[i]$, $0 \leq i \leq n-1$ ein binärer Vektor $\overrightarrow{[v]} = \overrightarrow{[\delta_{ij}]}$ abgeleitet wird, der $[i]$ repräsentiert, ist Protokoll BitMask aus [Sec10]. Eine Alternative soll hier vorgestellt werden. Die Idee dabei ist - wie bei BitMask - für jede Komponente des Vektors ein Polynom P_i vom Grad $n-1$ über $\mathbb{Z}/q\mathbb{Z}$ zu definieren, das genau an i eine 1 besitzt und 0 an den Stellen $j = 0,\ldots,n-1$, $j \neq i$. Im Unterschied zu [Sec10] werden hier (Abb. 3.6) jedoch die Polynome nach dem Hornerschema ausgewertet und es

Protokoll 3.5: $\overrightarrow{[W]} \leftarrow \mathsf{MinCons}\left(\overrightarrow{[Z]}, \overrightarrow{[N]}, \overrightarrow{[I]}, l\right)$

$r \leftarrow l \mod 2$
If $(l = 1)$
  $[I(0)] \leftarrow 1$
  return $\overrightarrow{[I]}$
Else
  If $(r = 0)$
    $d \leftarrow \frac{l}{2}$
  Else
    $d \leftarrow l - 1$
  If $(r = 0)$
    For $(i = 1, \ldots, l)$ **do parallel**
      If $(i \equiv 0 \mod 2)$
        $[x(i)] \leftarrow [Z(i)] \cdot [N(i+1)]$ //1 Mul $\mathbb{F}_q$
      Else
        $[x(i)] \leftarrow [Z(i)] \cdot [N(i-1)]$ //1 Mul $\mathbb{F}_q$
    For $(i = 1, \ldots, d)$ **do parallel**
      $[D(i)] \leftarrow \mathsf{LT}([x(2i-1), x(2i)], 2k)$ // vgl. Tabelle 2.1
      $[A'(i)] \leftarrow [D(i)] \cdot [Z(2i-1)] + (1 - [D(i)]) \cdot [Z(2i)]$ //2 Mul $\mathbb{F}_q$
      $[B'(i)] \leftarrow [D(i)] \cdot [N(2i-1)] + (1 - [D(i)]) \cdot [N(2i)]$ //2 Mul $\mathbb{F}_q$ } 1 Rnd $\mathbb{F}_q$
  Else
    $[x(1)] \leftarrow [Z(1)] \cdot [N(2)]$ //1 Mul $\mathbb{F}_q$
    $[x(2)] \leftarrow [Z(2)] \cdot [N(1)]$ //1 Mul $\mathbb{F}_q$ } 1 Rnd $\mathbb{F}_q$
    $[D(1)] \leftarrow \mathsf{LT}([x(1), x(2)], 2k)$ //vgl. Tabelle 2.1
    $[A'(1)] \leftarrow [D(1)] \cdot [Z(1)] + (1 - [D(1)]) \cdot [Z(2)]$ //2 Mul $\mathbb{F}_q$
    $[B'(1)] \leftarrow [D(1)] \cdot [N(1)] + (1 - [D(1)]) \cdot [N(2)]$ //2 Mul $\mathbb{F}_q$ } 1 Rnd $\mathbb{F}_q$
    For $(i = 2, \ldots, l-1)$
      $[A'(i)] \leftarrow [Z(i+1)]$
      $[B'(i)] \leftarrow [N(i+1)]$
  $\left(\overrightarrow{[I]}\right) \leftarrow \mathsf{MinCons}\left(\overrightarrow{[A']}, \overrightarrow{[B']}, \overrightarrow{[I]}, d\right)$
  If $(r = 0)$
    For $(i = 1, \ldots, d)$ **do parallel**
      $[I'(2i-1)] \leftarrow [D(i)] \cdot [I(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
      $[I'(2i-1)] \leftarrow [I(i)] - [I'(2i-1)]$
    $\overrightarrow{[I]} \leftarrow \overrightarrow{[I']}$
  Else
    $[I'(1)] \leftarrow [D(1)] \cdot [I(1)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
    $[I'(2)] \leftarrow [I(0)] - [I'(1)]$
    For $(i = l-1, \ldots, 2)$
      $[I(i+1)] \leftarrow [I(i)]$
    $[I(2)] \leftarrow [I'(2)]$
    $[I(1)] \leftarrow [I'(1)]$

Abbildung 3.5: Berechnung des Maximums einer Folge von Brüchen

Protokoll 3.6 : $\overrightarrow{[v]} \leftarrow \mathsf{BitMask}([i], n)$

For $(i = 0, \ldots, n-1)$ **do parallel**
  $[v(i)] \leftarrow \lambda_{i,n}$
  For $(j = n-1, \ldots, 2)$
    If $(j > i)$
      $[v(i)] \leftarrow [v(i)] \cdot ([x] - j) + \lambda_{i,j}$ //1 Mul $\mathbb{F}_q$, 1Rnd
    Else
      $[v(i)] \leftarrow [v(i)] \cdot ([x] - (j-1)) + \lambda_{i,j}$ //1 Mul $\mathbb{F}_q$, 1Rnd
  $[v(i)] \leftarrow [v(i)] \cdot ([x] - i) + \lambda_{i,0}$
Return $\overrightarrow{[v]}$

Abbildung 3.6: Kodieren einer Zahl in einem Vektor

kann so auf die Funktion PreMulC verzichtet werden. Die Interpolationskoeffizienten $\lambda_{i,j}$ können lokal im Voraus mit Hilfe der dividierten Differenzen berechnet werden.

3.1.8 Prefix-Or

Sei v ein binärer Vektor und i eine natürliche Zahl. Um die Funktion

$$\mathsf{Prev}_i(v) = \bigvee_{k=1}^{i} v_k.$$

sicher zu implementieren, wird Protokoll Prefix-Or(v, i) ($\mathsf{Prev}_i(v)$) aus [CdH10a] verwendet. Die Kosten betragen $5i - 1$ Multiplikationen in $\mathbb{F}_q$ bzw. $2i - 1$ Multiplikationen nach Pre-Computing.

3.1.9 FindFirst

In vielen Situationen muss von einem binären Vektor das erste Element bestimmt werden, das [1] ist, z.B. bei der Funktion Norm, die Teil der Division der Fixpunktarithmetik ist. Zur Verfügung steht hier einerseits Protokoll SelectFirst aus [Sec10]. Dessen Komplexität beträgt für einen Vektor der Länge k $\frac{k \log_2 k}{2}$[2] sichere Multiplikationen in $\log_2 k$ Runden. Schneller ist es jedoch, die erste 1 mit Hilfe von Schaltervariablen $[S]$ und $[C]$ zu bestimmen. Die Idee dabei ist, bei Auftreten einer [1] eine Schaltervariable $[S]$ so umzulegen, dass bei allen folgenden Variablen eine Null im Ergebnisvektor eingetragen wird (Protokoll 3.7). Die Komplexität beträgt $2k$ Multiplikationen in k Runden, hat

[2] k sei hier eine Zweierpotenz

Protokoll 3.7 : $\overrightarrow{[I]} \leftarrow \textbf{FindFirst}\left(\overrightarrow{[v]}, k\right)$

1 $[S] \leftarrow 1$
2 $[C] \leftarrow 0$
3 **For** $(i = 1, \ldots, k)$
4 $\quad [I(i)] \leftarrow [S] \cdot [v(i)]$ //1 Mul $\mathbb{F}_q$, 1 Rnd
5 $\quad [C] \leftarrow 1 - [v(i)]$
6 $\quad [S] \leftarrow [C] \cdot [S]$ //+1 Mul $\mathbb{F}_q$
7 **Return** $\overrightarrow{[I]}$

Abbildung 3.7: Bestimmen der ersten [1] einem binären Vektor

also einen geringeren Rechenaufwand bei höherer Rundenzahl als SelectFirst aus [Sec10].

3.1.10 Zusammenschieben eines Vektors

In vielen Anwendungen ist es notwendig, die Nullzeilen eines Vektors (oder einer Matrix) zu eliminieren, d.h. (z.B.) eine Transformation

$$\begin{pmatrix} * \\ 0 \\ \vdots \\ 0 \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} * \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

durchzuführen. Es können dabei zwei Fälle unterschieden werden:

1. Die Positionen der Nullen sind bekannt, d.h. kodiert in einem verteilten binärem Vektor.
2. Die Positionen der Nullen sind nicht bekannt.

Das zweite Problem wird mit Protokoll 3.8 gelöst.

In den ersten beiden Schritten wird ein Indikatorvektor konstruiert, dessen erstes Element mit [1] initialisiert wird. Im Lauf des Algorithmus ist immer genau eine Komponente von $\overrightarrow{[I]}$ gleich 1 und die restlichen gleich 0. Die Position der 1 zeigt an, an welche Position des Vektors der nächste Nicht-Null-Eintrag

Protokoll 3.8 : $\left(\overrightarrow{[I]}, [[P]]\right) \leftarrow$ **Kompaktifiziere** $\left(\overrightarrow{[v]}, d\right)$

1 $\overrightarrow{[I]} \leftarrow \mathsf{Vektor}(d)$
2 $[I(1)] \leftarrow 1$
3 **For**$(j = 1, \ldots, d)$
4 $\quad [w(j)] \leftarrow \mathsf{EQZ}([v(j)], k)$ //vgl. Tabelle 2.1
5 $\quad [w(j)] \leftarrow 1 - [w(j)]$
6 $\quad$ **For**$(i = 1, \ldots, d)$ **do parallel**
7 $\quad\quad [P(i,j)] \leftarrow [w(j)] \cdot [I(i)]$ //1 Mul $\mathbb{F}_q$, 1 Rnd
8 $\quad \overrightarrow{[I]} \leftarrow \mathsf{ErhoeheIndexKonditional}\left(\overrightarrow{[I]}, [w(j)], d\right)$ //Protokoll 3.9
9 $\overrightarrow{[v]} \leftarrow [[P]] \cdot \overrightarrow{[v]}$
10 **Return** $\left(\overrightarrow{[v]}, [[P]]\right)$

Abbildung 3.8: Elimination von Null-Elementen eines Vektors

gesetzt werden muss. In den Schritten 4 und 5 wird überprüft, ob die entsprechende Komponente gleich 0 ist und eliminiert wird. Ist dies der Fall, wird $[w(j)]$ auf 0 gesetzt, sonst 1. In der inneren `for`-Schleife (Schritte 6 - 7) wird die Permutationsmatrix $[[P]]$ spaltenweise erstellt. Eintrag i in Spalte j wird genau dann auf 1 gesetzt, wenn die j-te Komponente des Vektors $\neq 0$ ist und in Zeile i ein Nicht-Null-Eintrag gesetzt wird. Anschließend (ErhoeheIndexKonditional, Schritt 8) werden alle Komponenten in $\overrightarrow{[I]}$ um eine Position nach unten verschoben, wenn $[w(j)] \neq 0$. Die tatsächliche Eliminierung findet dann in Schritt 9 statt, wenn die soeben konstruierte Permutationsmatrix angewendet wird.

Protokoll 3.9 : $\overrightarrow{[v]} \leftarrow$ **ErhoeheIndexKonditional** $\left(\overrightarrow{[v]}, [B], d\right)$

1 $[G] \leftarrow$Share
2 **For**$(i = 1, \ldots, d)$
3 $\quad [H] \leftarrow [v(i)]$
4 $\quad [v(i)] \leftarrow [G]$
5 $\quad [v(i)] \leftarrow [B] \cdot [v(i)] + (1 - [B]) \cdot [H]$ //2 Mul, 1 Rnd $\mathbb{F}_q$
6 $\quad [G] \leftarrow [H]$
7 **Return** $\overrightarrow{[v]}$

Abbildung 3.9: Verschieben aller Elemente eines Vektors um eine Position nach unten, wenn $[B] = 1$

Findet eine solche Verschiebung statt, wird die erste Position auf Null gesetzt. Die letzte Komponente verschwindet. Die Verschiebung ist also nicht zirkulär. Sind die Positionen der Nicht-Null-Elemente bekannt (d.h. kodiert in einem binären Vektor, der dann der Funktion zusätzlich übergeben wird), können die Zeilen 4-5 von Protokoll 3.8 auch entfallen. Das Protokoll wird dann mit KompaktifiziereBinär bezeichnet. Es wird dafür zusätzlich der die Nicht-Null-Elemente kodierende Vektor $[\overrightarrow{I}]$ übergeben. Sollen die Nullzeilen einer *Matrix* $[[A]]$ eliminiert werden, so werden Schritte 3-8 auf die erste Spalte der Matrix angewendet und anschließend $[[P]]$ mit $[[A]]$ multipliziert. Ein analoges Vorgehen bietet sich an, wenn aus zwei oder mehr verschiedenen Vektoren Elemente eliminiert werden müssen: Protokoll 3.8 wird für *einen* Vektor ausgeführt und die resultierende Permutationsmatrix auf die anderen Vektoren angewandt. Soll in einem späteren Schritt die Permutation rückgängig gemacht werden, multipliziert man den Vektor mit $[[P^t]]$.

Der Hintergrund ist der folgende: In vielen Fällen (Kapitel 5) ist von einem Vektor der Länge m bekannt, dass höchstens $n < m$ Positionen von 0 verschieden sind. Für einige Operationen, z.B. für die Bestimmung von Minima und Maxima, reicht es deshalb aus, einen mit Hilfe von Protokoll 3.8 verkürzten Vektor zu betrachten. Muss in einem späteren Schritt die Position im Gesamtvektor eines so bestimmten Maximums (oder einer anderen Größe) ermittelt werden, kann diese einfach bestimmt werden, indem der Vektor, der die Position des Maximums im verkürzten Vektor kodiert, in einen leeren Vektor der Länge n eingebettet und anschließend mit $[[P^t]]$ multipliziert wird.

Die Berechnungen EQZ in Zeile 4 können parallelisiert werden. Für die von ihnen abhängigen Berechnungen der Permutationsmatrix $[[P]]$ (Schritt 6-7) gilt dies nicht! Der Indikatorvektor $[\overrightarrow{I}]$, in dem der Index der nächsten Nullzeile, in die die nächste Nicht-Nullzeile permutiert wird, gespeichert ist, und von dem die Elemente von $[[P]]$ abhängen, kann sich in jeder Iteration ändern!

3.2 Matrizenrechnung

3.2.1 Sicheres Schreiben und Lesen von Elementen

Analog zu Abschnitt 3.1.2 definiert sind die Funktionen SecReadColumn$([[A]], [\overrightarrow{I}], m, n)$ (bzw. SecReadRow) und SecWriteColumn$([[A]], [\overrightarrow{I}], [\overrightarrow{v}], m, n)$ (bzw. SecWriteRow) für das sichere Schreiben und Lesen einer $m \times n$-Matrix $[[A]]$. Im Fall SecReadColumn betragen die Kosten dafür n und im Fall SecWriteColumn $2mn$ sichere Multiplikationen.

Protokoll 3.10: $[[N]] \leftarrow$ **MatrixMult** $([[L]], [[M]], m, n, l)$

1 **For** $(i = 1, \ldots, m)$ **do parallel**
2 **For** $(j = 1, \ldots, n)$ **do parallel**
3 $[N(i,j)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[L(i,1:l)]}, \overrightarrow{[M(1:l,j)]}, m\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
4 $[N(i,j)] \leftarrow \mathsf{TruncPr}([N(i,j)], k, f)$ //vgl. Tabelle 2.1
5 **Return** $[[N]]$

Abbildung 3.10: Matrizenmultiplikation, bei der die oberen $m - l$ Zeilen der linken Matrix keine Rolle spielen.

3.2.2 Matrizenmultiplikation

Der Algorithmus zur Multiplikation zweier Matrizen entspricht dem nichtverteilten Algorithmus. Ein wesentlicher Unterschied besteht darin, dass das Multiplizieren von Zeilen mit Spalten als Berechnung von Skalarprodukten aufgefasst wird und somit Protokoll Inner (Abschnitt 3.1.1) verwendet werden kann, das die Kosten pro Skalarprodukt auf die einer sicheren Multiplikation reduziert. Die Gesamtkosten einer Matrizenmultiplikation (Abb. 3.10) einer $m \times l$- mit einer $l \times n$-Matrix sind somit $m \cdot n$ sichere Multiplikationen. Anstelle von $\mathsf{MatrixMult}([[L]], [[M]], m, n, l)$ wird im Folgenden auch einfach $[[L]] \cdot [[M]]$ geschrieben, wenn die Dimensionen klar sind.

Beachte, dass die Operation TruncPr (Schritt 4) entfällt, wenn eine (oder beide) Matrizen aus Nicht-Fixpunktzahlen besteht. Ist nach einer Matrix-Matrix-Multiplikation eine TruncPr-Operation auszuführen, kann diese für jedes Element der Matrix parallel durchgeführt werden! Man schreibt in diesem Fall:

$$\mathsf{TruncPr}([[A]], k, f)$$

Die analoge Aussage gilt für TruncPr-Operationen nach Matrix-Vektor Multiplikationen.

3.2.3 Erstellen einer Permutationsmatrix

In diesem Abschnitt wird die Erstellung von Permutationsmatrizen zum sicheren Vertauschen der Zeilen r und $[l]$ beschrieben (Protokoll 3.11). Dabei ist r öffentliche Variable und $[l]$ liegt als binärer Vektor $\overrightarrow{[v]}$ vor.[3] Ersetzt man in einer Einheitsmatrix die r-te Zeile und Spalte durch $\overrightarrow{[v]}$ und zudem die Diagonale $\overrightarrow{[d]}$ durch $[d(i)] - [v(i)]$ für $i \neq r$, so erhält man eine Permutationsmatrix, die das Gewünschte leistet.

[3] Hat $\overrightarrow{[v]}$ keine solche Form, so lässt es sich z.B. mit Hilfe von Protokoll 3.6 erzeugen.

Protokoll 3.11: $[[P]] \leftarrow \mathsf{PMatrix}\left(\overrightarrow{[p]}, r, n\right)$

```
1  For (i = l, ..., n)
2      [P(i,r)] ← [p(i)]
3      [P(r,i)] ← [p(i)]
4      If (i! = r)
5          [P(i,i)] ← 1 − [p(i)]
6  return [[P]]
```

Abbildung 3.11: Geheime Erstellung einer Permutationsmatrix

1. Anwendung: Zeilenpermutation im Rahmen der Pivotisierung der LR-Zerlegung einer quadratischen Matrix. Hier wird die r-te Zeile mit derjenigen vertauscht, die in der Restspalte von r das betragsmäßig größte Element besitzt.

2. Anwendung: Eine Matrix $[[A]]$ enthält mehrere Null-Zeilen, deren Positionen in einem binären Vektor $\overrightarrow{[v]}$ kodiert sind.

$$A = \begin{pmatrix} \overrightarrow{[v_1]}^t \\ 0 \\ \cdots \\ 0 \\ \overrightarrow{[v_2]}^t \\ \overrightarrow{[v_3]}^t \\ 0 \\ \cdots \\ 0 \\ \overrightarrow{[v_4]}^t \end{pmatrix} \tag{3.6}$$

Es sollen die Nicht-Nullspalten in den oberen Zeilen der Matrix A' zusammengefasst werden.

$$A' = \begin{pmatrix} \overrightarrow{[v_1]}^t \\ \overrightarrow{[v_2]}^t \\ \overrightarrow{[v_3]}^t \\ \overrightarrow{[v_4]}^t \\ 0 \\ \cdots \\ 0 \end{pmatrix} \tag{3.7}$$

Dies geschieht analog zum Kompaktifizieren von Vektoren (Protokoll 3.8).

4 Lineare Gleichungssysteme

Das Lösen linearer Gleichungssysteme

$$Ax = b \tag{4.1}$$

für eine quadratische Matrix A ist fundamentaler Baustein der allermeisten numerischen Verfahren. Die bisher entwickelten sicheren Verfahren sind alle nur sehr eingeschränkt praxistauglich. Häufig wird z.B. gefordert, dass alle Werte der Matrix einem endlichen Körper entstammen ([DA01],[CD01])[1]. In [DA01] wird zusätzlich ein Zweiparteienszenario vorausgesetzt. Ein anderes Protokoll ([AF10]) geht gar nicht erst auf das Lösen linearer Gleichungssysteme ein, sondern beschränkt sich auf sichere Matrizenmultiplikation. In diesem Abschnitt werden (nach Wissensstand des Autors erstmalig) sichere Implementierungen von zwei allgemein anwendbaren Verfahren (dem Gaußalgorithmus und der Lösung mittels QR-Zerlegung), die beide keine Anforderungen außer der Invertierbarkeit an die Matrix A stellen sowie von zwei Verfahren (der Lösung mittels Cholesky-Zerlegung und das CG-Verfahren), die eine symmetrisch positiv definite Matrix voraussetzen, vorgestellt. Die Verfahren werden bzgl. ihrer Komplexität verglichen.

4.1 Gaußverfahren

4.1.1 Sichere Berechnung der LR-Zerlegung

Die LR-Zerlegung (auch als Vorwärtselimination bekannt) einer invertierbaren Matrix ist eines der wichtigsten Werkzeuge der numerischen Mathematik. Ziel ist, für eine quadratische Matrix A eine linke untere Dreiecksmatrix L mit 1en auf der Diagonale und eine rechte obere Dreiecksmatrix R zu finden, so dass gilt

$$L \cdot R = A. \tag{4.2}$$

Details finden sich z.B. in [SB02]. Um eine ordnungsgemäße Beendigung des Algorithmus zu garantieren, findet eine Zeilenpivotisierung statt, also eine Zeilenvertauschung, die sicherstellt, dass in jedem Eliminationsschritt durch das betragsmäßig größte Element der Restspalte dividiert wird. Der wesentliche

[1] In diesem Zusammenhang ist es *nicht* möglich, die Werte als ganze Zahlen kleiner als der gewählte Modulus zu betrachten, da die Division *innerhalb* des endlichen Körpers Teil des Protokolls ist!

Protokoll 4.1: $([[A]],[[P]]) \leftarrow \mathsf{LR}([[A]],n)$

1 **For**$(l = 1,\ldots,n-1)$
2 $\quad[[P']] \leftarrow 0$
3 $\quad\overrightarrow{[p]} \leftarrow \overrightarrow{[A(l:n,l)]}$
4 $\quad\overrightarrow{[p]} \leftarrow \mathsf{MaxVektor}(\overrightarrow{[p]}, n-l+1)$ //Protokoll 3.3
5 $\quad\left(\overrightarrow{[I]} = \left([0]_1 \quad \cdots \quad [0]_{l-1} \quad \overrightarrow{[p]}\right)^t\right) \hookleftarrow \overrightarrow{[p]}$
6 $\quad[[P']] \leftarrow \mathsf{PMatrix}\left(\overrightarrow{[I]}, n-l+1, n\right)$ //Protokoll 3.11
7 $\quad$**If**$(l = 1)$
8 $\quad\quad[[P]] \leftarrow [[P']]$
9 $\quad$**Else**
10 $\quad\quad[[P(l{:}n,l{:}n)]] \leftarrow [[P'(l{:}n,l{:}n)]] \cdot [[P(l{:}n,l{:}n)]]$ //Protokoll 3.10
11 $\quad[[A(l{:}n,l{:}n)]] \leftarrow [[P'(l{:}n,l{:}n)]] \cdot [[A(l{:}n,l{:}n)]]$ //Protokoll 3.10
12 $\quad$**For**$(i = l+1,\ldots,n)$
13 $\quad\quad$**If**$(i = l+1)$
14 $\quad\quad\quad[H] \leftarrow \mathsf{InvertNoTrunc}([A(l,l)],k,f)$ //s. Tabelle 2.1
15 $\quad\quad[A(i,l)] \leftarrow [A(i,l)] \cdot [H]$
16 $\quad\quad[A(i,l)] \leftarrow \mathsf{TruncPr}([A(i,l)],3k-f,2k-f)$ //vgl. Tabelle 2.1
17 $\quad\quad$**For** $(j = l+1,\ldots,n)$ **do parallel**
18 $\quad\quad\quad[H] \leftarrow [A(i,l)] \cdot [A(l,j)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
19 $\quad\quad\quad[H] \leftarrow \mathsf{TruncPr}([H],k,f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
20 $\quad\quad\quad[A(i,j)] \leftarrow [A(i,j)] - [H]$
21 **Return** $([[A]],[[P]])$

Abbildung 4.1: Sichere Berechnung der LR-Zerlegung einer quadratischen Matrix.

Unterschied zwischen der sicheren (Protokoll 4.1) und der nicht-sicheren Implementierung besteht darin, dass die Pivotisierung hier mit Hilfe von Permutationsmatrizen durchgeführt wird, die verdeckt erstellt werden können (vgl. Abschnitt 3.2.3).

In den Zeilen 2-11 wird die Zeilen-Pivotisierung durchgeführt und die Permutationsmatrix aktualisiert. Dazu wird in Schritt 4 zunächst das Pivotelement bestimmt und aus dessen Position in der Matrix - nach der Einbettung des Positionsvektors in einen der der Größe des Gleichungssystems entspricht (Schritt 5) - in Schritt 6 die Permutationsmatrix konstruiert, die die Pivotzeile an die benötigte Stelle transponiert (Schritt 11). Zusätzlich wird noch die Permutationsmatrix $[[P]]$, in der alle aufgelaufenen Permutationen gespeichert sind, aktualisiert (Schritt 7-10). Da im Iterationsschritt (Schritt 12-20) die Divi-

Input: LR-Zerlegung L und R der Matrix A, Vektor b
Output: Lösung des Gleichungssystems $Ly = b$

$y_1 \leftarrow \frac{b_1}{L_{1,1}}$
For$(l = 2, \ldots, n)$
    For$(i = l + 1, \ldots, n)$
        $b_i \leftarrow b_i - L_{il} \cdot b_l$
    $y_l \leftarrow \frac{b_l}{L_{(l,l)}}$
Return y

Abbildung 4.2: (Standard) Vorwärtssubstitution

sion durch das Pivotelement immer für die gesamte Restspalte durchgeführt werden muss, wird die Funktion InvertnoTrunc verwendet, um zunächst $\left[\frac{1}{a_{ii}}\right]$ zu berechnen und dieses unmittelbar anschließend mit $[A(i,l)]$ zu multiplizieren (Schritt 15). Das Abschneiden wird nach der Invertierung des Pivotelements weggelassen und erst nach der Multiplikation mit $[A(i,l)]$ nachgeholt (Schritt 16). Die Schritte 7-10 und 11 sowie 17-20 können parallelisiert werden.

4.1.2 Sichere Vorwärts- und Rückwärtssubstitution

Besitzt man die Zerlegung $A = L \cdot R$ einer invertierbaren Matrix A, so kann man mit Hilfe von Vorwärts- und Rückwärtssubstitution (s. Abb. 4.2 und 4.3), d.h. Lösen der Systeme

$$Ly = b \text{ und } Rx = y \tag{4.3}$$

jedes Gleichungssystem der Form

$$A \cdot x = b \tag{4.4}$$

für einen bekannten Vektor b und einen unbekannten Vektor x lösen. Ändert sich nur der Vektor b, so kann die zerlegte Matrix genutzt werden und es müssen nur die Substitutionen durchgeführt werden. Beachte, dass bei der Vorwärtssubstitution im Zusammenhang mit der LR-Zerlegung das Diagonalelement als 1 vorausgesetzt wird, d.h. die Divisionen aus Abbildung 4.2 entfallen in diesem Fall.

Üblicherweise wird die Vorwärtssubstitution, wie in Abb. 4.2 dargestellt, zeilenweise ausgeführt. Für eine sichere Implementierung ist dies jedoch ungünstig: Je größer der Index l der zu berechnenden Variable y_l ist, desto häufiger muss das Element b_l in der inneren Schleife modifiziert werden. Fasst man

Input: LR-Zerlegung L und R der Matrix A, Lösung y von $Ly = b$
Output: Lösung des Gleichungssystems $Rx = y$

For$(l = n, \ldots, 1)$
  $x_l \leftarrow y_l$
  For$(i = l + 1, \ldots, n)$
    $x_l \leftarrow x_l - R_{li} \cdot x_i$
  $x_l \leftarrow \frac{x_l}{R_{l,l}}$
Return x

Abbildung 4.3: (Standard) Rückwärtssubstitution

alle Modifikationen (vor der Division) des Elements y_l durch Umordnung des Algorithmus zusammen, ergibt sich die Formel:

$$y_l \leftarrow b_l - L_{l,1} \cdot b_1 - L_{l,2} \cdot b_2 - \ldots - L_{l,l-1} \cdot b_{l-1}. \tag{4.5}$$

Man sieht, dass sich dies als Skalarprodukt der Vektoren

$$\begin{pmatrix} b_l \\ -b_1 \\ \vdots \\ -b_{l-1} \end{pmatrix} \text{ und } \begin{pmatrix} 1 \\ L_{l,1} \\ \vdots \\ L_{l,l-1} \end{pmatrix}$$

darstellen lässt. Bei Verwendung von Protokoll Inner (Abschnitt 3.1.1) kann so jede Komponente von y in *einer* sicheren Multiplikation und einer TruncPr-Operation (und ggf. einer Division) berechnet werden, die innere Schleife entfällt also. Wird die Vorwärtssubstitution also auf diese Weise (spaltenweise) durchgeführt, sind die Kosten der sicheren Implementierung (Protokoll 4.4) nicht mehr quadratisch, sondern nur noch linear in der Dimension der Matrix. Zusätzlich sollte sich die Genauigkeit durch die Reduktion der Anzahl der sicheren Multiplikationen und der TruncPr-Operationen erhöhen. Die Schleifenausführungen der verbleibenden (äußeren) Schleife können parallelisiert werden.

Analog ist die Formel für die Berechnung *einer* Komponente x_l bei der Rückwärtssubstitution gegeben durch

$$x_l \leftarrow \frac{x_l - R_{l,l+1} \cdot y_l - \ldots - R_{l,n} \cdot y_n}{R_{l,l}}. \tag{4.6}$$

Auch hier lässt sich in der sicheren Implementierung (Abb. 4.5) der Zähler effizient mit Hilfe eines Skalarprodukts berechnen. Wie bei der sicheren Vorwärtssubstitution wächst die Anzahl der sicheren Multiplikationen nun nicht

Protokoll 4.4: $\overrightarrow{[y]} \leftarrow \textbf{VWSubs}\left([[L]], \overrightarrow{[b]}, n\right)$

1 $[x(1)] \leftarrow \mathsf{FPDiv}([x(1)], [L(1,1)], k, f)$ //vgl. Tabelle 2.1
2 **For**$(i = 2, \ldots, n)$
3 $\quad \overrightarrow{[B]} \leftarrow \left([b(i)] \quad -[b(1)] \quad \cdots \quad -[b(i-1)]\right)^t$
4 $\quad \overrightarrow{[v]} \leftarrow \left([1] \quad [L(i,1)] \quad \cdots \quad [L(i,i-1)]\right)^t$
5 $\quad [y(i)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[B]}, \overrightarrow{[v]}, i\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
6 $\quad [y(i)] \leftarrow \mathsf{TruncPr}([y(i)], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
7 $\quad [y(i)] \leftarrow \mathsf{FPDiv}([y(i)], [L(i,i)], k, f)$ //vgl. Tabelle 2.1
8 **Return** $\overrightarrow{[y]}$

Abbildung 4.4: Sichere Implementierung der Vorwärtssubstitution einer quadratischen Matrix.

mehr quadratisch mit der Größe des Gleichungssystems, sondern nur noch linear.

Die Komplexität, die beim Lösen eines linearen Gleichungssystems mit Hilfe der LR-Zerlegung anfällt, findet sich wieder in Tabelle 4.2.

4.2 Das Sichere QR-Verfahren

Es sei A eine quadratische Matrix. Ziel der QR-Zerlegung ist, eine orthogonale Matrix Q (d.h. $Q^tQ = QQ^t = Id$) und eine rechte obere Dreiecksmatrix R zu bestimmen, d.d.

$$Q \cdot R = A. \tag{4.7}$$

Ein lineares Gleichungssystem $Ax = b$ kann dann mit Hilfe der Umformung

$$Ax = b \Leftrightarrow Q \cdot Rx = b \Leftrightarrow Rx = Q^t b \tag{4.8}$$

und einer Rückwärtssubstitution gelöst werden. Die QR-Zerlegung kann durch sequentielle Anwendung von *Householder-Matrizen* berechnet werden. Jede solche Householder-Matrix ist verantwortlich für die Berechnung einer Spalte von R. Ihr Produkt bildet Q. Um eine Householder-Matrix zu berechnen, muss zuerst der *Householder-Vektor* $\overrightarrow{v}$ aus der entsprechenden Spalte $\overrightarrow{v}$ von A berechnet werden. Die Householder-Matrix kann daraus einfach abgeleitet werden. Für die Lösung eines linearen Gleichungssystems ist die explizite Kenntnis von Q jedoch nicht nötig: Die Linksmultiplikation mit Q^t kann auch nur mit Kenntnis der Householder-Vektoren direkt implementiert werden (4.11).

Protokoll 4.5: $\overrightarrow{[x]} \leftarrow \mathsf{RWSubs}\left([[A]], \overrightarrow{[b]}, n\right)$

1 $[x(n)] \leftarrow \mathsf{FPDiv}([x(n)], [R(n,n)], k, f)$ //vgl. Tabelle 2.1
2 **For**$(l = n-1, \ldots, 1)$
3 $\quad \overrightarrow{[v]} \leftarrow \left([1] \quad -[R(l,l+1)] \cdots \quad -[R(l,n]]\right)$
4 $\quad \overrightarrow{[w]} \leftarrow \left([b(l)] \quad [y(l+1)] \quad \cdots \quad [y(n)]\right)$
5 $\quad [x(l)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[w]}, n-l\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
6 $\quad [x(l)] \leftarrow \mathsf{TruncPt}([x(l)], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
7 $\quad [x(l)] \leftarrow \mathsf{FPDiv}([x(l)], [R(l,l)], k, f)$ //Tabelle 2.1
8 **Return** $\overrightarrow{[x]}$

Abbildung 4.5: Sichere Implementierung der Rückwärtssubstitution einer quadratischen Matrix.

Protokoll 4.6: $\overrightarrow{[v]} \leftarrow \mathsf{House}\left(\overrightarrow{[x]}, n\right)$

1 $[\mu] = \mathsf{VektorNorm}\left(\overrightarrow{[x]}, n\right)$ //Protokoll 3.1
2 $[\sigma] \leftarrow \mathsf{Sgn}([x(1)], k)$ // Formel (2.10)
3 $[x(1)] \leftarrow [x(1)] + [\sigma] \cdot [\mu]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
4 **Return** $\overrightarrow{[v]}$

Abbildung 4.6: Berechnung eines Householder-Vektors

4.2.1 Sichere Berechnung von Householder-Vektoren

Definition 4. *Sei $v \in \mathbb{R}^n \backslash \{0\}$ ein Vektor. Dann ist ein zu v gehöriger* Householder-Vektor *ein Vektor $v_h \in \mathbb{R}^n$, d.d.*

$$\left(Id - \frac{2v_h v_h^t}{v_h^t v_h}\right) \cdot v = \begin{pmatrix} * \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{4.9}$$

Die Matrix

$$H = \left(Id - \frac{2v_h v_h^t}{v_h^t v_h}\right), \tag{4.10}$$

die alle bis auf die erste Komponente von v annuliert, bezeichnet man auch als Householder-Matrix *zu v. Offensichtlich ist ein solcher Vektor nicht eindeutig.*

Protokoll 4.7: $([[A]], [\beta]) \leftarrow$ **PreMultHouse** $\left([[A]], \overrightarrow{[v]}, m, n\right)$

1	$[\tilde{v}] \leftarrow \mathsf{Inner}\left(\overrightarrow{[v]}, \overrightarrow{[v], m}\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
2	$[\tilde{v}] \leftarrow \mathsf{TruncPr}([\tilde{v}], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3	$[\beta] \leftarrow -2 \cdot \mathsf{DivNR}(1, [\tilde{v}], k, f)$ //vgl. Tabelle 2.1
4	$\overrightarrow{[\tilde{v}]} \leftarrow \mathsf{Matrix\text{-}Mult\text{-}Vector}([[A]]^t, \overrightarrow{[v]})$ //m Mul, 1 Rnd $\mathbb{F}_q$
5	$\overrightarrow{[w]} \leftarrow [\beta] \cdot \overrightarrow{[\tilde{v}]}$ //1 Mul, 1 Rnd $\mathbb{F}_q$
6	$\overrightarrow{[w]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[w]}, 2k, 2f\right)$ //m Mul, 1 Rnd $\mathbb{F}_q$, $m(f+1)$ Mul $\mathbb{F}_{q_1}$
7	$[[V]] \leftarrow \mathsf{Matrix\text{-}Matrix\text{-}Multiply}\left(\overrightarrow{[v]}, \overrightarrow{[w]}^t\right)$ //mn Mul, 1 Rnd $\mathbb{F}_q$
8	$[[V]] \leftarrow \mathsf{TruncPr}([[V]], k, f)$ //mn Mul, 1 Rnd $\mathbb{F}_q$, $(mn) \cdot (f+1)$ Mul $\mathbb{F}_{q_1}$
9	$[[A]] \leftarrow [[A]] + [[V]]$
10	**Return** $([[A]], [\beta])$

Abbildung 4.7: Prä-Multiplikation von A mit der Householder-Matrix, die durch den Householder-Vektor $\overrightarrow{[v]}$ bestimmt ist

Insbesondere kann durch solche Operationen (4.9) eine Matrix auf obere Dreiecksform gebracht werden. Geometrisch bedeutet die Transformation (4.9), dass der Vektor v - ggf. mit einem Streckungsfaktor - auf die erste Koordinatenachse gespiegelt wird. Die sichere Berechnung der Householder-Vektoren (Protokoll 4.6) folgt dem in [GL96] dargelegten Algorithmus mit der Ausnahme, dass die erste Komponente des Vektors nicht auf 1 skaliert wird, was bei einem Vektor der Länge n die Berechnung einer Invertierung, $n-1$ Multiplikationen und eine TruncPr-Operation erspart. Zuerst wird die Norm des Vektors $\overrightarrow{[x]}$ berechnet (Schritt 1) und in Schritt 2 das Vorzeichen der 1. Komponente von $\overrightarrow{[x]}$. Diese beiden Schritte können parallel ausgeführt werden. In Schritt 3 wird dann die erste Komponente von $\overrightarrow{[x]}$ entsprechend [GL96] modifiziert.

4.2.1.1 Multiplikation mit einer Householder-Matrix

Ist ein verteilter Householder-Vektor $\overrightarrow{[v]}$ und eine Matrix $[[A]]$ gegeben, so kann die Prä-Multiplikation der zu $\overrightarrow{[v]}$ gehörigen Householder-Matrix $[[H]]$ mit $[[A]]$ effizient mit dem in Abb. 4.7 beschriebenen Algorithmus durchgeführt werden. Dieser implementiert die Formel

$$H \cdot A = A + vw^t, \text{ wobei } w = \beta A^t v \text{ und } \beta = \frac{-2}{v^t v}. \tag{4.11}$$

Die Korrektheit verifiziert sich einfach mit Hilfe von Gleichung (4.9). Post-Multiplikation unterscheidet sich nur marginal.

Protokoll 4.8: $[[H]] \leftarrow \mathsf{HouseholderMatrix}\left(\overrightarrow{[v]}, n\right)$

1. $\overrightarrow{[h]} \leftarrow \mathsf{House}\left(\overrightarrow{[v]}, n\right)$ //Protokoll 4.6
2. $[\tilde{u}] \leftarrow \mathsf{Inner}\left(\overrightarrow{[h]}, \overrightarrow{[h]}\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
3. $[\tilde{u}] \leftarrow \mathsf{TruncPr}([\tilde{u}], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
4. $[\beta] \leftarrow -2 \cdot \mathsf{DivNR}(1, [\tilde{u}], k, f)$ //vgl. Tabelle 2.1
5. $\overrightarrow{[g]} \leftarrow [\beta] \cdot \overrightarrow{[h]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
6. $[[V]] \leftarrow \mathsf{Matrix\text{-}Matrix\text{-}Multiply}\left(\overrightarrow{[g]}, \overrightarrow{[h^t]}\right)$ //n^2 Mul $\mathbb{F}_q$, 1 Rnd
7. $[[V]] \leftarrow \mathsf{TruncPr}([[V]], 3k-f, 2k-f)$ //n^2 Mul, 1 Rnd $\mathbb{F}_q$, $n^2(f+1)$ Mul $\mathbb{F}_{q_1}$
8. $[[H]] \leftarrow Id - [[V]]$
9. **Return** $[[H]]$

Abbildung 4.8: Erstellung der zum Vektor $\overrightarrow{[v]}$ gehörenden Householder-Matrix

Das Protokoll zum Erstellen der zum Vektor $\overrightarrow{[v]}$ gehörenden Householder-Matrix $[[H]]$ (d.h. der Formel aus (4.9)) findet sich in Protokoll 4.8.

4.2.2 Sichere Berechnung der QR-Zerlegung

Mit obigen Werkzeugen kann die sichere Berechnung der QR-Zerlegung einer quadratischen Matrix einfach durchgeführt werden (Protokoll 4.9). Für jede Spalte wird zunächst der entsprechende Householder-Vektor (Schritt 2) berechnet. Mit dessen Hilfe können in Schritt i die Elemente unterhalb der Diagonale in Spalte i (Schritt 3) eliminiert werden. Die Matrix R wird auf und oberhalb der Diagonalen der Eingabematrix gespeichert und die Householder-Vektoren (ohne die jeweils ersten Komponenten, die extern in einem Vektor $\overrightarrow{[w]}$ gespeichert werden müssen (Schritt 6), da die Householder-Vektoren ja aus Effizienzgründen nicht normiert wurden) unterhalb (Schritt 5). Falls nötig kann aus ihnen die Matrix Q explizit berechnet werden.

4.2.3 Sicheres Lösen Linearer Gleichungssysteme durch QR-Verfahren

Der Algorithmus zum Lösen linearer Gleichungssysteme $Ax = b$ (Abb. 4.10) implementiert Formel (4.8). In Schritt 1 wird eine QR-Zerlegung der gegebenen Matrix A durchgeführt. Zurückgegeben werden der Vektor $\overrightarrow{[w]}$, der die ersten Einträge der Householder-Vektoren enthält (vgl. Abb. 4.6), der Vektor

Protokoll 4.9: $\left([[A]], \overrightarrow{[w]}, \overrightarrow{[\beta]}\right) \leftarrow \mathbf{QR}\,([[A]], n)$

```
1  For (j = 1,...,n-1)
2      [v(j:n)]  ← House ([A(j:n,j)], n-j+1) // Protokolle 4.6 und 4.7
3      ([[A(j:n,j:n)]], [β]) ←PreMultHouse ([[A(j:n,j:n)]], [v(j:n)], n-j, n-j)
4      If j < n
5          [[A(j+1:n,j)]] ← [v(j+1:m)]
6      [w(j)] ← [v(j)]
7  Return ([[A]], [w], [β])
```

Abbildung 4.9: Sichere Berechnung der QR-Zerlegung einer quadratischen Matrix mit Hilfe von Householder-Matrizen

$\overrightarrow{[\beta]}$, der die Koeffizienten $[\beta(i)]$ der Prä-Multiplikation mit den entsprechenden Householder-Vektoren enthält (vgl. Abb. 4.7) und die Matrix $[[A]]$, die auf und über der Diagonalen die Matrix $[[R]]$ enthält und unter der Diagonalen die restlichen Householder-Vektoren enthält. In den Schritten 2-5 wird sukzessive der Vektor $\overrightarrow{[b]}$ mit den zu den erzeugten Householder-Vektoren gehörigen Householder-Matrizen von vorne multipliziert. Dies ist günstiger als die Matrix $[[Q]]$ explizit zu berechnen (Abschnitt 4.2.1.1). Beachte, dass obwohl entsprechend (4.8) $\overrightarrow{[b]}$ mit $[[Q^t]]$ multipliziert werden muss, die Householder-Matrizen (4.9) symmetrisch sind und dies somit irrelevant ist. Allerdings kehrt sich durch die Transponierung die Reihenfolge um, in der mit den Householder-Matrizen multipliziert werden muss. Protokoll 4.10 trägt dem Rechnung. Die Prä-Multiplikation mit der Householder-Matrix (Zeile 5) ist eine leichte Abwandlung von Protokoll 4.7: Der dort zu berechnende Koeffizient

$$\beta = -2\frac{vv^t}{v^t v}$$

für den entsprechenden Householder-Vektor v wurde bereits bei der QR-Zerlegung berechnet und kann wiederverwendet werden. Desweiteren wird die Prä-Multiplikation immer nur mit einem Vektor und nicht wie in Protokoll 4.7 mit einer Matrix ausgeführt. Beachte, dass der Vektor $\overrightarrow{[v]}$ in jede Iteration neu instantiiert wird. Zurückgegeben werden neben der Lösung $\overrightarrow{[x]}$ die zerlegte Matrix $[[A]]$ sowie die Koeffizienten $\overrightarrow{[w]}$ und $\overrightarrow{[\beta]}$. Die Komplexität findet sich in Tabelle 4.2.

Bemerkung: Mit den in den Abschnitten 4.1 und 4.2 beschriebenen Verfahren lassen sich auch Gleichungssysteme der Form

Protokoll 4.10: $\left(\overrightarrow{[x]}, [[A]], \overrightarrow{(\beta)}, \overrightarrow{[w]}\right) \leftarrow$ **QRSolve** $\left([[A]], \overrightarrow{[b]}, n\right)$

```
1  ([[A]], [w], [β]) ← QR ([[A]], n) //Protokoll 4.9
2  For (i = 1, ..., n − 1)
3      [v(2 : n − i + 1)] ← [A(i + 1 : n, i)]
4      [v(1)] ← [β(i)]
5      [b(n−i+1:n)] ← PreMultHouseβ ([b(n−i+1:n)], [v], [w(i)], m, n) //Prot. 4.7
6  [x] ←RWSubs([[A]], [b], n) //Protokoll 4.5
7  Return ([x], [[A]], (β), [w])
```

Abbildung 4.10: Lösen des Gleichungssystems $Ax = b$ mit Hilfe der QR-Zerlegung

$$\begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ 0 \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{4.12}$$

lösen. Bei Verwendung des Gauss-Algorithmus liefert die beschriebene sichere Implementierung eine LR-Zerlegung, bei der die unteren Zeilen der Matrix L und die rechten Spalten der Matrix R 0 sind! Dies ist insbesondere dadurch begründet, dass $\frac{[0]}{[0]} = [0]$ - egal welcher der beiden Divisionsalgorithmen verwendet wird! Dies ist auch der Grund weshalb Vorwärts- und Rückwärtssubstitution von Nullzeilen nicht beeinträchtigt werden. Nach deren Ausführung erhält man einen Vektor, dessen obere Komponenten die Lösung $\overrightarrow{[x]}$ von $[[A]] \cdot \overrightarrow{[x]} = \overrightarrow{[b]}$ darstellen und dessen untere Komponenten 0 sind.

Eine QR-Zerlegung einer quadratischen Matrix ist möglich unabhängig davon, ob diese invertierbar ist, insbesondere also auch für Matrizen der Form (4.12). Es kann dann auch die Berechnung $Q^t \cdot b$ auf jeden Fall ausgeführt werden. Zur Lösung von (4.12) fehlt dann nur noch die Rückwärtssubstitution. Wie beim Gauss-Algorithmus ist diese problemlos möglich.

4.3 Das Sichere Cholesky-Verfahren

4.3.1 Die Sichere Cholesky-Zerlegung

Ist A eine positiv definite $n \times n$-Matrix, so kann man für A die sogenannte Cholesky-Zerlegung ([GL96],[SB02])

Verfahren	Multiplikationen	Runden	Körper
House	$15 + 6\theta_{SQR}$	$18 + 4\theta_{SQR}$	$\mathbb{F}_q$
	$9k + 3\theta_{SQR}(f+1) + 6$	2	$\mathbb{F}_{q_1}$
	$1,5k\log_2 k + 3k - 3$	$3\lceil\log_2 k\rceil$	$\mathbb{F}_{2^8}$
PreMultHouse	$\mathcal{O}(n^2 + \theta_{DivNr})$	$13 + 3\theta_{DivNR}$	$\mathbb{F}_q$
	$\mathcal{O}(n^2 f + \theta_{DivNR} k)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(k\log_2 k)$	$3\lceil\log_2 k\rceil$	$\mathbb{F}_{2^8}$
HH-Matrix	$\mathcal{O}(n^2 + \theta_{DivNr})$	$3\theta_{DivNR} + 2f + 28$	$\mathbb{F}_q$
	$\mathcal{O}(n^2 f + \theta_{DivNR} k)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(k\log_2 k)$	$3\lceil\log_2 k\rceil$+6	$\mathbb{F}_{2^8}$
BerechneQ	$\frac{2}{3}n^3 + n^2 - \frac{5}{3}n$	$6(n-1)$	$\mathbb{F}_q$
	$(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n) \cdot (f+1)$	2	$\mathbb{F}_{q_1}$

Tabelle 4.1: Komplexität der zum QR-Verfahren gehörigen Protokolle

$$A = LL^t \tag{4.13}$$

angeben. Dabei ist L eine untere Dreiecksmatrix. Die Zerlegung kann ohne Pivotisierung berechnet werden und kann zur Lösung eines linearen Gleichungssystems $Ax = b$ verwendet werden. Die sichere Version des Algorithmus aus [GL96] ist in Abb. 4.11 dargestellt. Die Matrix wird dabei im unteren Teil mit L überschrieben. Anschließend wird L^t in den Bereich oberhalb der Diagonale kopiert. Beachte, dass die innerste Schleife (Zeilen 8-11) parallelisiert werden kann.

4.3.2 Sicheres Lösen Linearer Gleichungssysteme durch Cholesky-Verfahren

Verfügt man über eine Zerlegung $A = LL^t$, so kann mit Hilfe der Vorwärtssubstitution (Abb. 4.4) $Ly = b$ und der Rückwärtssubstitution $L^t x = y$ ein lineares Gleichungssystem $Ax = b$ gelöst werden (Abb. 4.12). Zurückgegeben wird neben der Lösung $[\overrightarrow{x}]$ die Cholesky-Zerlegung der Matrix $[[A]]$. Die Komplexität kann wieder aus Tabelle 4.2 abgelesen werden.

4.4 Das Sichere CG-Verfahren

Ist $A \in M_{n,n}(\mathbb{Q})$ eine symmetrisch positiv definite Matrix, dann lässt sich die Lösung des linearen Gleichungssystems

Protokoll 4.11: $[[A]] \leftarrow$ **Cholesky** $([[A]], n)$

For $(i = 1, \ldots, n)$
  $[A(l,l)] \leftarrow$ SQR $([A(l,l)], k, f)$ //Protokoll 2.7
  $[I] \leftarrow$ Invers$([A(l,l)])$ //vgl. Tabelle 2.1
  For $(l = i+1, \ldots, n)$
    $[A(l,i)] \leftarrow [A(l,i)] \cdot [I]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
    $[A(l,i)] \leftarrow$ TruncPr $([A(l,i)], k, f)$//1 M., 1 R. $\mathbb{F}_q$, f+1 M. $\mathbb{F}_{q_1}$
  For $(l = i+1, \ldots, n)$
    For $(j = l, \ldots, n)$ **do parallel**
      $[H] \leftarrow [A(l,i)] \cdot [A(j,i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
      $[H] \leftarrow$ TruncPr$([H], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul $\mathbb{F}_{q_1}$
      $[A(j,l)] \leftarrow [A(j,l)] - [H]$
For $(i = 1, \ldots, n-1)$
  For $(l = n, \ldots, i+1)$
    $[A(i,l)] \leftarrow [A(l,i)]$
Return $[[A]]$

Abbildung 4.11: Sichere verteilte Cholesky-Zerlegung

$$Ax = b$$

auch bestimmen mit Hilfe des Verfahrens der konjugierten Gradienten (CG-Verfahren). Erstmals beschrieben wurde es in [HS52] und ist eines der Standard-Verfahren zur iterativen Lösung großer, insbesondere dünn besetzter, Gleichungssysteme. Es handelt sich um ein approximatives Verfahren: Bei großen Dimensionen kann abgebrochen werden, sobald die Iterierten hinreichend nahe zusammen sind. Ist n die Dimension der Matrix, so liefert das Verfahren aber auch nach n Iterationen das korrekte Ergebnis. Die schematische Darstellung ist in Abb. 4.13 zu sehen. Die Idee ist, die Zielfunktion $f(x) =$

Protokoll 4.12: $\left([[A]], \overrightarrow{[x]}\right) \leftarrow$ **CholeskySolve** $\left([[A]], \overrightarrow{[b]}, n\right)$

$[[A]] \leftarrow$ Cholesky $([[A]], n)$ //Protokoll 4.11
$\overrightarrow{[y]} \leftarrow$VWSubs$\left([[A]], \overrightarrow{[b]}, n\right)$ //Protokoll 4.4
$\overrightarrow{[x]} \leftarrow$RWSubs$\left([[A]], \overrightarrow{[y]}, n\right)$ //Protokoll 4.5
Return $\left([[A]], \overrightarrow{[x]}\right)$

Abbildung 4.12: Sichere verteilte Cholesky-Zerlegung

Input: Symmetrisch positiv definite Matrix $A \in M_{n,n}(\mathbb{Q})$, $b \in \mathbb{Q}^n$
Output: Die Lösung von $Ax = b$; dies entspricht dem Minimum von $\frac{1}{2}x^tAx - b^tx$.

1. $x^{(0)}$ sei Startwert
2. $d^{(0)} := -g^{(0)} = b - Ax^{(0)}$
3. for $k = 0, \ldots, n$
   a) $\alpha_k = \frac{g^{(k)t}g^k}{d^{(k)t}Ad^{(k)}}$
   b) $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
   c) $g^{(k+1)} = g^(k) + \alpha_k Ad^{(k)}$
   d) $\beta_k = \frac{g^{(k+1)t}g^{(k+1)}}{g^{(k)t}g^{(k)}}$
   e) $d^{(k+1)} = -g^{(k+1)} + \beta_k d^{(k)}$

Abbildung 4.13: Der CG-Algorithmus

$\frac{1}{2}x^tGx + b^tx$, die ihr globales Minimum bei x_0 mit $Ax_0 = b$ besitzt, in n Iterationen in Richtung n linear unabhängiger Vektoren sukzessive zu minimieren. Dabei sind aufeinanderfolgende Suchrichtungen d_k, d_{k+1} bzgl. der G-Metrik senkrecht, d.h. $\langle d_k, Gd_{k+1}\rangle = 0$.

Die Umsetzung als sichere Mehrparteienberechnung findet sich in Protokoll 4.14. Aus den in Abschnitt 1.3.2 erwähnten Gründen kommt ein Abbruch des Programms vor Beendigung von n Schleifendurchläufen - anders als in einer nicht-sicheren Implementierung - nicht in Frage. Der Algorithmus durchläuft also n Iterationen. Somit spielt der Startwert keine Rolle und wird als $\overrightarrow{[x]} = 0$ gesetzt, was n sichere Multiplikationen und TruncPr-Operationen erspart.
Die vergleichsweise geringen Kosten von $\mathcal{O}(n^2)$ sicheren Multiplikationen über $\mathbb{F}_q$ liegen vor allem darin begründet, dass keine teuren Matrix-Matrix-Multiplikationen anfallen und als „teure“ Operationen nur die zwei Divisionen pro Runde anfallen.

4.5 Komplexität der Algorithmen

Die ungefähre Anzahl an flops der nicht-verteilten Implementierungen verhält sich wie in Tabelle 4.3 angegeben. Man sieht, dass für allgemeine Glei-

Protokoll 4.14: $\overrightarrow{[x]} \leftarrow \mathsf{CG}\left([[A]], \overrightarrow{[b]}, n\right)$

1 $\overrightarrow{[x]} \leftarrow 0$
2 $\overrightarrow{[d]} \leftarrow \overrightarrow{[b]}$
3 $\overrightarrow{[g]} \leftarrow -\overrightarrow{[b]}$
4 $[Z] \leftarrow \mathsf{Inner}\left(\overrightarrow{[g]}, \overrightarrow{[g]}, n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
5 $[Z] \leftarrow \mathsf{TruncPr}([z], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, 2 Mul, 2 Rnd $\mathbb{F}_{q_1}$
6 **For** $(k = 1, \ldots, n)$
7 $\quad \overrightarrow{[Ad]} \leftarrow [[A]] \cdot \overrightarrow{[d]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
8 $\quad \overrightarrow{[Ad]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[Ad]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $2n$ Mul $\mathbb{F}_{q_1}$
9 $\quad [N] \leftarrow \mathsf{Inner}\left(\overrightarrow{[d]}, \overrightarrow{[Ad]}, n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
10 $\quad [N] \leftarrow \mathsf{TruncPr}\,([N], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, 2 Mul $\mathbb{F}_{q_1}$
11 $\quad [\alpha] \leftarrow \mathsf{FPDiv}([Z], [N], k, f)$ //vgl. Tabelle 2.1
12 $\quad \overrightarrow{[\alpha d]} \leftarrow [\alpha] \cdot \overrightarrow{[d]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
13 $\quad \overrightarrow{[\alpha d]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[\alpha d]}, k, f\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, 2 Mul $\mathbb{F}_{q_1}$
14 $\quad \overrightarrow{[x]} \leftarrow \overrightarrow{[x]} + \overrightarrow{[\alpha d]}$
15 $\quad \overrightarrow{[\alpha Ad]} \leftarrow [\alpha] \cdot \overrightarrow{[Ad]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
16 $\quad \overrightarrow{[\alpha Ad]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[\alpha Ad]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $2n$ Mul $\mathbb{F}_{q_1}$
17 $\quad \overrightarrow{[g]} \leftarrow \overrightarrow{[g]} + \overrightarrow{[\alpha Ad]}$
18 $\quad [Z2] \leftarrow \mathsf{Inner}\,([g], [g])$ //1 Mul, 1 Rnd $\mathbb{F}_q$
19 $\quad [Z2] \leftarrow \mathsf{TruncPr}\,([Z2], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, 2 Mul $\mathbb{F}_{q_1}$
20 $\quad [\beta] \leftarrow \mathsf{FPDiv}([Z2], [Z], k, f)$ //vgl. Tabelle 2.1
21 $\quad [Z] \leftarrow [Z2]$
22 $\quad \overrightarrow{[\beta d]} \leftarrow [\beta] \cdot \overrightarrow{[d]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
23 $\quad \overrightarrow{[\beta d]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[\beta d]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $2n$ Mul $\mathbb{F}_{q_1}$
24 $\quad \overrightarrow{[d]} \leftarrow \overrightarrow{[\beta d]} - \overrightarrow{[g]}$
25 **Return** $\overrightarrow{[x]}$

Abbildung 4.14: Der Sichere CG-Algorithmus

chungssysteme der Gauß-Algorithmus das effizienteste Verfahren ist. Ist das Gleichungssystem positiv definit, ist das Cholesky-Verfahren überlegen. Ist die Matrix zusätzlich dünn besetzt, empfiehlt sich das CG-Verfahren, da sich die angegebenen Kosten, die im Wesentlichen aus Matrix-Vektor Multiplikationen zusammensetzen, erheblich reduzieren lassen. Zudem liefert es als approximatives Verfahren i.d.R. in sehr viel weniger als n Iterationen ein hinreichend

Verfahren	Multiplikationen	Runden	Körper
Gaußalgorithmus	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathbb{F}_q$
	$\mathcal{O}(fn^3+n^2k+\theta_{Div}nk+\theta_{DivNR}nf)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(n^2k+nk\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$
QR-Algorithmus	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathcal{O}(\theta_{DivNR}n)$	$\mathbb{F}_q$
	$\mathcal{O}(fn^3+\theta_{DivNR}nk)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(nk\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$
Cholesky	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathbb{F}_q$
	$\mathcal{O}(fn^3+\theta_{DivNR}kn)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(nk\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$
CG-Algorithmus	$\mathcal{O}(n^2+\theta_{Div}n)$	$\mathcal{O}(n\theta_{Div})$	$\mathbb{F}_q$
	$\mathcal{O}(fn^2+nk+nf\theta_{Div})$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(nk\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$

Tabelle 4.2: Komplexität der Sicheren Verfahren zum Lösen Linearer Gleichungssysteme. Beachte, dass $f < k$ und $\theta_{DivNR} > \theta_{Div}, \theta_{SQR}$.

genaues Ergebnis. Allerdings muss es für jede neue rechte Seite vollständig neu berechnet werden - anders als alle anderen vorgestellten Algorithmen.

Im Vergleich dazu sind die Kosten der sicheren Implementierungen zu sehen (Tabelle 4.2). Anders als bei den nicht-verteilten Implementierungen besitzt hier der CG-Algorithmus die geringste Komplexität. Obwohl die Kosten von Gaußalgorithmus und QR-Verfahren ähnlich erscheinen, zeigt sich doch bei näherer Betrachtung, dass letzterer - anders als bei der nicht-sicheren Implementierung - bei zunehmender Größe überlegen ist. Mehr dazu in Kapitel 7.

Die vergleichsweise geringen Werte der Rundenkomplexität, insbesondere für die Operationen in $\mathbb{F}_{q_1}$, sind darauf zurückzuführen, dass alle in den Algo-

Gauß-Algorithmus	$\frac{2n^3}{3}$
QR-Verfahren	$\frac{4n^3}{3}$
Cholesky	$\frac{n^3}{3}$
CG	n^3

Tabelle 4.3: Rechenaufwand in flops der nicht-verteilten Verfahren zum Lösen linearer Gleichungssysteme bei großem n (Anzahl der Gleichungen).

rithmen benötigten Zufallsbits (in der Theorie) parallel erzeugt werden können (vgl. Abschnitt 2.2). Praktisch durchführbar ist dies jedoch nicht (vgl. Abschnitt 1.4).

5 Quadratische Optimierung

In diesem Kapitel soll das Kernstück der vorliegenden Arbeit, die sichere Implementierung zweier quadratischer Optimierungsalgorithmen, dargelegt werden. Es handelt sich dabei einerseits um eine *primale Aktive Mengen Strategie* sowie andererseits um eine *duale* Aktive Mengen Strategie. Es werden zunächst die Problemstellung sowie die Optimalitätsbedingungen, die Karush-Kuhn-Tucker Bedingungen, angewandt auf ein quadratisches Optimierungsproblem, erörtert. Nach einem kurzen Abschnitt über Präprozesse, d.h. Möglichkeiten zur Bestimmung der für die unterliegende Fixpunktarithmetik zu verwendenden Parameter k und f sowie über Skalierung, folgt eine kurze Beschreibung eines der frühesten Verfahren zum Lösen quadratischer Optimierungsprobleme, der Hauptpivotisierungsmethode von Dantzig. Eine genaue Betrachtung zeigt, dass sie für eine Implementierung als sichere Mehrparteienberechnung nicht geeignet ist. Es schließt sich an die Beschreibung der sicheren Implementierung der Primalen Aktiven Mengen Strategie - einschließlich der Bestimmung eines zulässigen Startwerts - sowie die der dualen Aktiven Mengen Strategie.

5.1 Allgemeines

5.1.1 Das Quadratische Optimimierungsproblem

Es sei $G \in \mathbb{R}^{n,n}$ symmetrisch positiv definit, $c \in \mathbb{R}^n$. Gesucht ist das Minimum der (quadratischen) Zielfunktion $f : \mathbb{R}^n \to \mathbb{R}$

$$f(x) = \frac{1}{2} x^t G x + c^t x \tag{5.1}$$

unter den Nebenbedingungen

$$a_i^t x \geq b_i, \quad i \in \mathcal{I} \tag{5.2}$$
$$a_j^t x = b_j, \quad j \in \mathcal{E}. \tag{5.3}$$

Dabei ist I die Menge der *Ungleichheitsbedingungen* und $\mathcal{E}$ die Menge der *Gleichheitsbedingungen*. Man definert zusätzlich $c_j := a_j^t x - b_j$, $j \in \mathcal{E} \cup \mathcal{I}$.

Der Einfachheit der Darstellung wegen sei zunächst $\mathcal{E} = \emptyset$. Der Fall, dass auch Gleichheitsnebenbedingungen existieren, ist eine einfache Modifikation, die später bei den einzelnen Verfahren gesondert betrachtet wird. Gemeinsam ist jedoch die Annahme, dass die Anzahl der Gleichheitsnebenbedingungen *öffentliche* Konstante ist, d.h. allen Spielern bekannt. In vielen Fällen, u.a. der wichtigsten betrachteten Anwendung, den Support-Vector-Maschines, stellt dies keine Einschränkung der Sicherheit dar: Dort ist $|\mathcal{E}|$ immer gleich 1.

5.1.2 Definitionen und Bedingungen für Optimalität

Definition 5. *Ein Punkt $x \in \mathbb{R}^n$ heißt* primal zulässig *für ein quadratisches Optimierungsproblem (5.1), wenn alle Nebenbedingungen (5.2)-(5.3) erfüllt sind.*

Definition 6. *Die* aktive Menge *an einem Punkt $x \in \mathbb{R}^n$ ist die Teilmenge $W_x \subset \mathcal{I} \cup \mathcal{E}$ aller Gleichungen (5.2 und der Ungleichungen aus 5.3) für die Gleichheit gilt.*

Da die Karush-Kuhn-Tucker Bedingungen im Algorithmus von Goldfarb und Idnani eine große Rolle spielen, werden sie hier kurz wiederholt. Die Darstellung orientiert sich an [NW99].

Satz 2 (Notwendige Bedingungen 1. Ordnung)**.** *Sei x^* lokale Lösung eines beschränkten Optimierungsproblems und seien die aktiven Nebenbedingungen linear unabhängig. Dann existiert ein Lagrange-Multiplikator $\lambda^* \in \mathbb{R}^m$ ($m = |\mathcal{E} \cup \mathcal{I}|$), so dass die folgenden Bedingungen erfüllt sind*

$$\begin{aligned}
\nabla_x \mathcal{L}(x^*, \lambda^*) &= 0 && (5.4)\\
c_i(x^*) &= 0 \quad i \in \mathcal{E} && (5.5)\\
c_i(x^*) &\geq 0 \quad i \in \mathcal{I} && (5.6)\\
\lambda_i^* &\geq 0 \quad i \in \mathcal{I} && (5.7)\\
\lambda_i^* c_i(x^*) &= 0 \quad \forall i \in \mathcal{E} \cup \mathcal{I} && (5.8)
\end{aligned}$$

Dabei ist die *Lagrange-Funktion* $\mathcal{L}(x, \lambda)$ definiert als

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x). \tag{5.9}$$

Die Variablen λ werden auch als *duale Variablen* bezeichnet. Im hier behandelten Fall ist die Zielfunktion quadratisch und so lässt sich (5.4), wenn alle Nebenbedingungen in der Matrix A zusammengefasst werden, umformen zu

$$Gx^* + c = A^t \lambda^*. \tag{5.10}$$

Die letzte Bedingung (5.8) bezeichnet man auch als Komplementärbedingung. Wegen (5.7) ist die folgende Definition sinnvoll.

Definition 7. *Ein Punkt $x \in \mathbb{R}^n$ heißt* dual zulässig *für das Problem 5.1 unter den Nebenbedingungen 5.2, wenn die Lagrange-Multiplikatoren der aktiven Menge alle ≥ 0 sind.*

Bemerkung: Da die Hessematrix G positiv definit ist, sind die Bedingungen (5.4)-(5.8) auch hinreichend für ein globales Minimum ([NW99]).

Im Folgenden wird - implizit und explizit - angenommen, dass die Nebenbedingungen nicht degeneriert sind. Dies bedeutet, dass die *Linear Independence Constraint Qualification* (LICQ, [NW99]) an jedem Punkt erfüllt ist:

Definition 8 (LICQ)**.** *Sei $x \in \mathbb{R}^n$ ein Punkt mit zugehöriger aktiver Menge $W(x)$. Die* Linear Independence Constraint Qualification (LICQ) *ist erfüllt, wenn die Menge der Gradienten $\{\nabla c_i(x) | i \in W(x)\}$ aktiver Nebenbedingungen linear unabhängig ist.*

Insbesondere bedeutet LICQ, dass an keinem Punkt mehr als n Nebenbedingungen aktiv sein können.

5.2 Präprozess

5.2.1 Bestimmung der Fixpunktparameter

Vor Beginn des Optimierungsverfahrens müssen die unterliegenden Parameter, insbesondere die Bitlänge k der zu verwendenden Fixpunktzahlen und die Anzahl der Nachkommastellen f festgelegt werden. Dies kann z.B. geschehen, indem mit Hilfe von Protokoll 3.3 das Maximum der Beträge der Maxima der von den einzelnen Spielern beigetragenen Werte ermittelt wird, woraus sich ein sinnvoller Wert für k ableiten lässt, z.B. indem er einige Größenordnungen größer gewählt wird.

Es reicht das Maximum ungefähr zu bestimmen, d.h. man kann den ermittelten Wert vor dem Öffnen noch mit einer kleinen zufälligen (geheimen) Konstanten multiplizieren, ohne dass die Größenordnung beeinträchtigt wird. Für diese Berechnungen kann der Modulus des unterliegenden Secret Sharing Schemes so groß gewählt werden (z.B. 2048 Bit Länge), dass nicht davon auszugehen ist, dass eine Modulo-Reduktion nötig ist. Am Ende dieser Berechnung können dann der Modulus q, der für die eigentliche Rechnung verwendet wird sowie die Werte k und f bestimmt werden.

Bemerkung: Die Anzahl der Nachkommastellen f hängt von der gewünschten Genauigkeit ab. Soll eine Genauigkeit f' des Ergebnisses erreicht werden, so ist es i.d.R. ausreichend f als das zwei- oder dreifache von f' zu wählen, um Rundungsfehler im Ergebnis x^* zu vermeiden. Grundsätzlich sollte f stärker mit

Protokoll 5.1: $\left([[A]], \overrightarrow{[b]}\right) \leftarrow$ **Skalierung** $\left([[A]], \overrightarrow{[b]}, k, f, m, n\right)$

1 **For**$(i = 1, \ldots, m)$ **do parallel**
2 **For**$(j = 1, \ldots, n)$ **do parallel**
3 $[v(j)] \leftarrow$Signum$([A(i,j)], k)$ //Protokoll 2.10
4 $[v(j)] \leftarrow [v(j)] \cdot [A(i,j)]$ //1 Mul $\mathbb{F}_q$
5 $[v(n+1)] \leftarrow$Signum$([b(i)], k)$ //Protokoll 2.10
6 $[v(n+1)] \leftarrow [v(n+1)] \cdot [b(i)]$ //1 Mul $\mathbb{F}_q$
7 $\overrightarrow{[I]} \leftarrow$MaxVektor$(\overrightarrow{[v]}, n, k)$ //Protokoll 3.3
8 $[M] \leftarrow$Inner$(\overrightarrow{[I]}, \overrightarrow{[v]}, n)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
9 $[M] \leftarrow$InvertNoTrunc$([M], k)$ //vgl. Tabelle 2.1
10 **For**$(j = 1, \ldots, n)$ **do parallel**
11 $[A(i,j)] \leftarrow [A(i,j)] \cdot [M]$ //1 Mul $\mathbb{F}_q$
12 $[b(i)] \leftarrow [b(i)] \cdot [M]$ //1 Mul $\mathbb{F}_q$
13 $[[A]] \leftarrow$TruncPr$([[A]], 3k-f, 2k-f)$ //vgl. Tabelle 2.1
14 $\overrightarrow{[b]} \leftarrow$TruncPr$(\overrightarrow{[b]}, 3k-f, 2k-f)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $n \cdot (2k-f+1)$ Mul $\mathbb{F}_{q_1}$
15 **Return**$([[A]], \overrightarrow{[b]})$

Abbildung 5.1: Skalierung der Nebenbedingungen

der Größe der verwendeten Zahlen variieren als k, da bei kleinen auftretenden Zahlen die Anzahl der Nachkommastellen wichtiger ist als bei großen.

5.2.2 Skalierung der Nebenbedingungen

Wenn keine Aussagen über die Größenordnung der Nebenbedingungen vorliegen (was beim sicheren, verteilten Rechnen häufig der Fall ist), sollten diese skaliert werden, so dass sie alle ungefähr gleich groß sind, insbesondere dann, wenn sie physikalische oder ökonomische Bedingungen repräsentieren, die für das Problem von gleicher Wichtigkeit sind ([GMW81]), aber verschiedene Größenordnungen besitzen. In diesem Fall bietet es sich an, alle Nebenbedingungen auf ein festes Intervall (z.B. $[-1, 1]$) zu skalieren. Man erreicht dies, indem das betragsmäßig maximale Element einer jeden Nebenbedingung bestimmt wird und im Anschluss daran, jedes Element der entsprechenden Zeile durch diesen Betrag dividiert wird (Protokoll 5.1).

Es wird für jede Nebenbedingung zunächst (Schritte 2-6) der Betrag der Koeffizienten einer jeden Nebenbedingung $a_i^t x \geq b_i$ bestimmt und in einen Vektor $\overrightarrow{[v]}$ geschrieben. Von diesem wird im Anschluss das Maximum bestimmt (Schritt 7) und jedes Element der entsprechenden Zeile durch das Maximum dividiert

(Schritte 9-15). Dazu wird zunächst das Inverse des Maximums $[M]$ gebildet und dieses dann mit den Werten der entsprechenden Zeile multipliziert. Dies ist wesentlich billiger, als jedes Element der Zeile explizit durch das Maximum zu dividieren.

Bemerkung: Es kann sinnvoll sein, die Parameter k und f, die in der Berechnung verwendet werden, erst *nach* der Skalierung zu bestimmen. Auf diese Weise kann ein übergroß dimensionierter Zahlenraum vermieden werden.

5.3 Nur Gleichheitsbedingungen

Sind nur Gleichheitsbedingungen ($I = \emptyset$) gegeben, hat das Optimierungsproblem folgende Gestalt:

$$\text{Minimiere} \quad \frac{1}{2}x^t Gx + x^t c \tag{5.11}$$

$$\text{unter} \quad Ax = b \tag{5.12}$$

Die notwendigen Bedingungen ((5.4),(5.5),(5.8)) erster Ordnung am optimalen Punkt x^* sind dementsprechend:

$$Gx^* - A^t\lambda + c = 0 \tag{5.13}$$

$$Ax^* - b = 0 \tag{5.14}$$

Es existiert eine eindeutige Lösung, wenn die Matrix A vollen Rang besitzt, die sich mit Standardtechniken (Kapitel 4) bestimmen lässt.

5.4 Die Hauptpivotisierungsmethode

5.4.1 Grobe Beschreibung des Verfahrens

Einer der bekanntesten (und einer der ersten) Algorithmen zur Lösung quadratischer Optimierungsprobleme sind der Algorithmus von Dantzig und Wolfe ([Dan63],[Fle87]) sowie die eng verwandte *Hauptpivotisierungsmethode* (Principal pivoting method), die in diesem Abschnitt betrachtet wird. Es ist ein Algorithmus, der das quadratische Optimierungsproblem

$$\text{Minimiere } \frac{1}{2}x^t Gx + c^t x \tag{5.15}$$

unter

$$Ax \geq b, \quad x \geq 0 \tag{5.16}$$

für positiv definites G löst. Man beachte, dass *zusätzlich* zur allgemeinen Formulierung (5.1)-(5.3) $x \geq 0$ gefordert wird. Die zugehörige Lagrangefunktion ist dann

$$\mathcal{L}(x, \lambda, \pi) = \frac{1}{2} x^t G x + c^t x - \lambda^t (Ax - b) - \pi^t x. \tag{5.17}$$

Dabei ist λ der Lagrange-Multiplikator für die Nebenbedingung $Ax \geq b$ und π der für die Nebenbedingung $x \geq 0$. Mit Hilfe der Schlupfvariablen $r = Ax - b$ ergibt sich aus (5.4)-(5.8) das System

$$\begin{aligned} \pi - Gx + A^t \lambda &= c && (5.18)\\ r - Ax &= -b && (5.19)\\ \pi, r, x, \lambda &\geq 0, \quad \pi^t x = 0, \quad r^t \lambda = 0. && (5.20) \end{aligned}$$

Die Darstellung lässt sich vereinfachen zu

$$\begin{aligned} w - Mz &= q && (5.21)\\ w &\geq 0, \quad , z \geq 0, \quad , w^t z = 0. && (5.22) \end{aligned}$$

Dabei sind

$$w = \begin{pmatrix} \pi \\ r \end{pmatrix}, \quad z = \begin{pmatrix} x \\ \lambda \end{pmatrix}, \quad M = \begin{pmatrix} G & -A^t \\ A & 0 \end{pmatrix}, \quad q = \begin{pmatrix} c \\ -b \end{pmatrix}. \tag{5.23}$$

(5.21) ist ein lineares Optimierungsproblem und lässt sich mit einem abgewandelten Simplex-Verfahren lösen. Der wesentliche Unterschied besteht darin, dass die Variablen in *komplementäre Paare* aufgeteilt werden: Eine Variable x_i ist komplementär zu einer Variablen y_i, wenn sie sich als (w_i, z_i) identifizieren lassen. Der Algorithmus sieht vor, dass für jede Basisvariable die komplementäre Variable Nicht-Basisvariable ist. Ist diese Bedingung erfüllt, bezeichnet man das Tableau als komplementär. Beim Basisaustausch soll zunächst versucht werden, eine Basisvariable x_t gegen ihre komplementäre Variable x_q zu tauschen. Wenn dadurch eine andere Basisvariable x_p kleiner als 0 wird und somit eine der Bedingungen aus (5.22) verletzt wird, ist dies nicht möglich und anstatt von x_t muss x_p die Basis verlassen. Man erhält ein nicht-komplementäres Tableau und muss im nächsten Schritt versuchen, das Komplement von x_p in die Basis hereinzunehmen und es gegen x_t zu tauschen. Wird dadurch wieder eine andere Basisvariable kleiner als 0, so muss wiederum diese die Basis verlassen. Man kann zeigen ([Fle87]), dass nach endlich vielen solchen Iterationen die Komplementarität wiederhergestellt ist.

Das Verfahren von Dantzig und Wolfe unterscheidet sich nur darin, dass Basisvariablen, die zu $\lambda-$ oder $\pi-$Variablen korrespondieren (temporär) kleiner als 0 werden dürfen. Man kann zeigen, dass sie äquivalent zur primalen aktiven Mengen-Strategie ist.

5.4.2 Probleme bei der Umsetzung als sichere Mehrparteienberechnung

Die Implementierung der Pivotwahl, wie sie im vorangegangenen Abschnitt beschrieben ist, stellt einer Umsetzung als sicherer Mehrparteienberechnung unüberwindbar scheinende Hindernisse in den Weg. Zunächst soll ja zur Laufzeit nicht bekannt werden, ob der Tausch einer Basisvariable gegen die komplementäre Variable erfolgreich durchgeführt werden kann. Aus diesem Grund müsste für *jede* Basisvariable der Effekt auf die anderen Basisvariablen berechnet werden, wenn diese die Basis verließe. Erst im Anschluss daran würde die geeignete Variable ausgewählt. Dies bedeutet bereits eine deutliche Verlangsamung gegenüber der in [CdH10b] vorgeschlagenen Implementierung des Simplex-Algorithmus. Viel schwerwiegender ist jedoch, dass alle Variablen, die aufgrund von Iterationen, die die Komplementarität verletzen, noch in die Basis aufgenommen werden müssen, in der richtigen Reihenfolge - verdeckt - gemerkt werden müssen. Zusätzlich muss die Liste permanent aktualisiert werden. Da **If**-Abfragen nur mit Hilfe des in Abschnitt 1.3.1 beschriebenen Verfahrens als sichere Mehrparteienberechnungen implementieren lassen, erscheint dies nicht praktikabel.

Die Restriktion $x \geq 0$ stellt zudem eine Einschränkung der Problemstellung gegenüber (5.1)-(5.3) dar und da für jede Ungleichheitsbedingung eine Schlupfvariable eingeführt werden muss, wird das Tableau für sehr viele Ungleichheitsbedingungen sehr groß. Aus diesen und anderen Gründen wird - auch für nicht-verteilte Implementierungen - davon abgeraten, den beschriebenen Algorithmus zur Lösung quadratischer Optimierungsprobleme einzusetzen ([Fle87]).

5.5 Primale Aktive Mengen Strategie

5.5.1 Aktive Mengen Strategien

Definition 9 (Aktive Mengen Strategie)**.** *Eine* Aktive Mengen-Strategie *(AMS) für ein quadratisches Optimierungsproblem löst dieses ausgehend von einem Ausgangspunkt iterativ, indem in jedem Schritt das Problem nur unter den jeweils aktiven Gleichungen betrachtet wird. Es wird ein Korrekturschritt durchgeführt, durch*

den entweder eine Nebenbedingung zur aktiven Menge hinzugefügt wird oder aus dieser gestrichen wird.

Man unterscheidet zwischen *primalen* und *dualen* Aktiven Mengen Strategien. Bei *primalen* Aktiven Mengen Strategien wird eine Folge primal zulässiger Punkte erzeugt, die erst bei Erreichen des Optimums auch dual zulässig sind. Bei *dualen* Aktiven Mengen Strategien ist die Folge der Punkte *dual* zulässig, d.h. optimal für die jeweilige aktive Menge und alle ihre Teilmengen, aber nicht *primal* zulässig. Aktive Mengen Strategien sind so konstruiert, dass - von seltenen Ausnahmefällen abgesehen - der optimale Punkt nach endlich vielen Schritten erreicht wird. Der optimale Punkt ist in beiden Fällen primal wie auch dual zulässig. In beiden Fällen darf sich der Wert der Zielfunktion von einem zum nächsten Iterationspunkt nicht verschlechtern.

5.5.2 Primale Aktive Mengen Strategie

In der hier betrachteten *primalen Aktiven Mengen Strategie* wird das Problem - ausgehend von einem primal zulässigen Startwert - durch sukzessive Hinzunahme von Nebenbedingungen zur aktiven Menge lokal gelöst. Falls an einem Punkt keine weitere Verbesserung mehr möglich ist, ist dieser entweder optimal oder die Bedingung (5.7) ist verletzt. In diesem Fall wird eine Nebenbedingung fallengelassen. Ist die Matrix G positiv definit und ist der zulässige Bereich nicht ausgeartet (d.h. LICQ erfüllt), ist ein stetiger Abstieg in der Zielfunktion garantiert. Da die Anzahl der möglichen aktiven Mengen begrenzt ist, terminiert das Verfahren nach endlich vielen Schritten ([Fle87]).

5.5.3 Berechnung des Startwerts

Die primale Aktive Mengen Strategie benötigt einen zulässigen Startwert. Im Normalfall ist ein solcher nicht bekannt, er lässt sich jedoch - wenn alle Nebenbedingungen linear sind - mit Hilfe verschiedener Varianten des Simplex-Algorithmus (s. z.B. [Fle87]; die Kenntnis des Simplex-Algorithmus sei im Folgenden vorausgesetzt; es werden nur die relevanten Variationen besprochen) bestimmen. Diese sind so konstruiert, dass ein Startwert leicht erkennbar ist (häufig der Nullpunkt). Einen solchen Algorithmus zur Bestimmung eines zulässigen Startwerts bezeichnet man auch als *Phase I-Algorithmus*, der eigentliche Optimierungsalgorithmus ist dann der *Phase II-Algorithmus*.

5.5.3.1 Der Phase I-Algorithmus

Grundlage des in diesem Abschnitt vorgestellten Vorgehens ist die sichere, verteilte Implementierung des Simplex-Algorithmus aus [CdH10b]. Die dort

verwendete Annahme $a_i^t x \leq b_i$, $b_i \geq 0, \quad i = 1, \ldots, m$ ist hier allerdings zu restriktiv: Es sollen zusätzlich Nebenbedingungen, bei denen $b_i < 0$ ist, betrachtet werden. Die Ungleichheitsnebenbedingungen $Ax \leq b$ werden mit Hilfe der Schlupfvariablen $r_i, \quad i = 1, \ldots, m$ überführt ([NW99]) in

$$Ax + Er = b. \tag{5.24}$$

Dabei ist E eine $m \times m$-Diagonalmatrix mit

$$E_{ii} = \begin{cases} 1 \Leftrightarrow b_i \geq 0 \\ -1 \Leftrightarrow b_i < 0 \end{cases}. \tag{5.25}$$

Die zu minimierende Zielfunktion ist dann ([GMW81])

$$f(r) = \sum_{i \in J} r_i \tag{5.26}$$

für die Menge J der am Startpunkt des Phase I - Algorithmus verletzten Nebenbedingungen. Da dieser i.d.R. an $x = 0$ und $r = b$ gestartet wird, gilt dann

$$J = \{i \in \{1, \ldots, m\} | b_i < 0\}. \tag{5.27}$$

An diesem Punkt ist also A_B, die Matrix bestehend aus den den Basisvariablen zugeordneten Variablen, gleich E. Im Hauptteil des Simplex-Tableaus einzutragen ist die Matrix

$$\begin{pmatrix} \hat{A_N} & Id \end{pmatrix} = \begin{pmatrix} A_B^{-1} A_N & Id \end{pmatrix} \tag{5.28}$$

und in die rechte Spalte $\hat{b} = A_B^{-1} b$ (die Indizes B und N bezeichnen dabei die Basis- und Nicht-Basisvariablen). Da $A_B = E$ (5.25) bedeutet dies, dass in A_N und b in den Zeilen, in denen $b_i < 0$, das Vorzeichen umgedreht werden muss (Protokoll 5.2, Schritt 5). Wegen (5.26) gilt für die Koeffizienten der Kostenfunktion

$$c_B(i) = \begin{cases} 1 \Leftrightarrow b_i < 0 \\ 0 \Leftrightarrow b_i = 0 \end{cases}. \tag{5.29}$$

Mit (5.25), $\widehat{A_N} = A_N \cdot A_B^{-1}$, $A_B^{-1} = A_B$ und $c_N = 0$ vereinfacht sich die *reduzierte Kostenfunktion*,

$$\widehat{c_N} = c_N - A_N^t A_B^{-1} c_B, \tag{5.30}$$

die die Auswahl der Pivotspalte bestimmt, zu

$$\widehat{c_N} = -\widehat{A_N}^t c_B. \tag{5.31}$$

Protokoll 5.2: $[[T]] \leftarrow$ **SetupTableau** $\left([[A]], \overrightarrow{[b]}\right)$

1	**For**$(i = 1, \ldots, m)$ **do parallel**
2	$[b_\sigma(i)] \leftarrow$ Signum$([b(i)], k)$ //s. Tabelle 2.1
3	**For**$(j = 1, \ldots, n)$ **do parallel**
4	$[T(i,j)] \leftarrow [A(i,j)] \cdot [b_\sigma(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
5	$[T(j, n+1)] \leftarrow [b(j)] \cdot [b_\sigma(j)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
6	$\overrightarrow{[T(m+1, *)]} \leftarrow -[[A^t]] \cdot \overrightarrow{[b_\sigma]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
7	**Return** $[[T]]$

Abbildung 5.2: Setup des Simplex-Tableaus

(Schritt 7 in Protokoll 5.2). Ist das Starttableau entsprechend Protokoll 5.2 berechnet, kann darauf die sichere Implementierung des Simplex-Algorithmus aus [CdH10b] angewandt werden.

Bemerkungen:

- In der Implementierung [CdH10b] werden die Basisspalten *nicht* angegeben! Dies vereinfacht die Berechnung.
- Eigentlich müsste die Menge J in jedem Schritt aktualisiert werden und die Berechnung nur unter den jeweils verletzen Nebenbedingungen erfolgen. Jedoch würde eine Kenntnis der Größe der aktiven Menge die Sicherheit beeinträchtigen. Aus (5.26) und (5.31) folgt, dass erfüllte Gleichungen *keinen* Einfluss auf die Zielfunktion $\widehat{c_N}$ und somit die Wahl der Pivotspalte besitzen. Gleichzeitig impliziert eine größere Anzahl an Nebenbedingungen, dass Simplex-Schritte ggf. kürzer sind, als mit einer kleineren Anzahl an Nebenbedingungen. Die Konvergenz wird davon jedoch nicht beeinträchtigt, nur u.U. die Anzahl der Schritte erhöht.
- Auch andere Methoden zur Bestimmung eines zulässigen Startwerts sind denkbar - wie z.B. die in [Gol72] vorgestellte auf [Ros60] beruhende. Die hier präsentierte lieferte jedoch bei den Testbeispielen (Kapitel 7) gute Ergebnisse. Die Zeit, die für die Ermittlung eines zulässigen Startwerts benötigt wird, ist nur ein Bruchteil der gesamten Rechenzeit (vgl. Kapitel 7).
- Der Algorithmus aus [CdH10b] formuliert die Nebenbedingungen in der Form $Ax \leq b$ im Gegensatz zu dieser Arbeit, in der sie als $Ax \geq b$ dargestellt sind. Um die Verwendung dieses Algorithmus zu erleichtern und die Kompatibilität mit [CdH10b] zu erhalten, wird für die Phase 1 die Form $Ax \leq b$ verwendet und für die Phase 2 die Form $Ax \geq b$.

Protokoll 5.3: $\overrightarrow{[w]} \leftarrow$ **AktiveMenge**$\left([[A]], \overrightarrow{[x]}, \overrightarrow{[b]}, m, n\right)$

1 $\overrightarrow{[r]} \leftarrow [[A]] \cdot \overrightarrow{[x]}$ //m Mul, 1 Rnd $\mathbb{F}_q$
2 $\overrightarrow{[r]} \leftarrow \mathsf{TruncPr}(\overrightarrow{[r]}, k, f)$ //m Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3 $\overrightarrow{[r]} \leftarrow \overrightarrow{[r]} - \overrightarrow{[b]}$
4 **For**$(i = 1, \ldots, m)$ **do parallel**
5 $\quad [w(i)] \leftarrow \mathsf{LTZ}([r(i)], k)$ //vgl. Tabelle 2.1
6 **Return** $\overrightarrow{[w]}$

Abbildung 5.3: Bestimmung der aktiven Menge an x_0

5.5.3.2 Bestimmung der aktiven Menge

Ist ein zulässiger Punkt x_0 gefunden, muss noch die zugehörige aktive Menge bestimmt werden. Dies geschieht, indem der Vektor $r := Ax_0 - b$ berechnet wird und überprüft wird, welche Komponenten kleiner als 0 sind (Protokoll 5.3).

5.5.4 Der Primale Algorithmus

Ist ein Startwert mit den in Abschnitt 5.5.3 beschriebenen Methoden gefunden bzw. gegeben, besteht das Prinzip des Algorithmus in der wiederholten Anwendung des Verfahrens aus Abschnitt 5.3. Im Detail hat ein Iterationsschritt die Form aus Abb. 5.4: Es sei ein zulässiger Punkt x_k mit zugehöriger aktiver Menge W_k zu Beginn der k-ten Iteration gegeben. Gesucht ist zunächst der Korrekturschritt d_k, so dass $x_{k+1} = x_k + d_k$ Lösung des Optimierungsproblems unter den Nebenbedingungen W_k ist. Die Zielfunktion hat an $x_k + d_k$ die Gestalt:

$$\frac{1}{2}(x_k + d_k)^t G(x_k + d_k) + c_k^t(x_k + d_k) = \frac{1}{2}x_k^t G x_k + x_k^t G d_k + \frac{1}{2}d_k^t G d_k + c^t x_k + c^t d_k \tag{5.32}$$

Bis auf die Terme

$$x_k^t G d_k + \frac{1}{2}d_k^t G d_k + c^t d_k = \frac{1}{2}d_k^t G d_k + d_k^t(G x_k + c) \tag{5.33}$$

sind jedoch alle konstant, da sie nicht von dem gesuchten d_k abhängen. Da der neue Iterationspunkt die selbe aktive Menge W_k besitzen soll, muss für d_k gelten:

$$a_i^t d_k = 0 \quad i \in W_k \tag{5.34}$$

Es ist also die Zielfunktion (5.33) unter den Nebenbedingungen (5.34) zu minimieren. Ersetzt man die Zielfunktion (5.1) für das lineare Gleichungssystem (5.13) und (5.14) durch (5.33), so erhält man das System

$$\begin{pmatrix} G & -A^t \\ A & 0 \end{pmatrix} \cdot \begin{pmatrix} d_k \\ \lambda_k \end{pmatrix} = \begin{pmatrix} -g_k \\ 0 \end{pmatrix}. \tag{5.35}$$

Dabei ist $g_k = Gx_k + c$. Ist $d_k \neq 0$ und $x_k + d_k$ primal zulässig, so setze $x_{k+1} = x_k + d_k$. Ist $x_k + d_k$ nicht zulässig, wird α in Bezug auf primale Zulässigkeit von $x_{k+1} = x_k + \alpha d_k$ maximiert. An dieser Stelle wird eine bisher inaktive Ungleichung aktiv, die der aktiven Menge W_{k+1} hinzugefügt wird. Es ist dann

$$x_{k+1} = x_k + \alpha_k d_k \tag{5.36}$$

wobei

$$\alpha_k = \min_{a_i^t d_k < 0,\, i \notin W_k} \left\{ 1, \frac{b_i - a_i^t x_k}{a_i^t d_k} \right\} \tag{5.37}$$

Sind alle Brüche $\frac{b_i - a_i^t x_k}{a_i^t d_k}$ größer als 1, so ist der Iterationspunkt x_{k+1} gegeben durch $x_k + d_k$. Ist andererseits mindestens ein solcher Bruch kleiner als 1, bedeutet dies, dass bei der dadurch gegebenen Schrittlänge die entsprechende Nebenbedingung aktiviert und bei jeder größeren Schrittlänge verletzt werden würde. Aus diesem Grund ist die Schrittlänge durch den kleinsten solchen Bruch gegeben. Die Bedingungen $a_i^t d_k < 0$ und $i \notin W_k$ stellen sicher, dass nur Nebenbedingungen, die in „Richtung" von d_k liegen, betrachtet werden (nach Voraussetzung ist ja $b_i - a_i^t x_k < 0$) und die gleichzeitig bisher nicht erfüllt sind (Dass die bisher aktiven Nebenbedingungen erfüllt bleiben, wird durch (5.34) sichergestellt).

Diese Schritte werden solange wiederholt, bis ein Punkt erreicht ist, der für die dann aktive Menge optimal ist, d.h. $d_k = 0$. Für diesen werden dann die Lagrange-Multiplikatoren λ berechnet. Sind sie nicht-negativ, d.h. ist die Komplementärbedingung erfüllt für alle Ungleichheitsbedingungen der aktiven Menge, ist der Punkt optimal. Ist andererseits mindestens ein λ_i negativ für $i \in I$, so wird das zu dem kleinsten Lagrange-Multiplikator korrespondierende i aus der aktiven Menge entfernt. Sind die Nebenbedingungen nicht degeneriert, terminiert der Algorithmus ([Fle87]) mit einer (theoretischen) Ausnahme (vgl. Bemerkung).

Bemerkung: Es ist (theoretisch) denkbar, dass $\min_{a_i^t d_k > 0,\, i \notin W_k} \left\{ \frac{b_i - a_i^t x_k}{a_i^t d_k} \right\} = 1$. In diesem Fall wird eine neue Nebenbedingung aktiv und müsste eigentlich zur aktiven Menge hinzugefügt werden. Geschieht dies nicht, können Zyklen

Input: Symmetrisch positiv definite Matrix $G \in M_{m,m}(\mathbb{Q})$, $A \in M_{n,m}(\mathbb{Q})$, $c \in \mathbb{Q}^n$, $b \in \mathbb{Q}^n$
Output: Lösung des Minimierungsproblems $\frac{1}{2}x^tGx + c^tx$ unter $Ax \leq b$.

1. Bestimme Startwert x_0.
2. Bestimme aktive Menge W_0
3. Für $k = 1, \ldots$
   a) Bestimme $g_k = Gx_k + c$
   b) Löse (5.33) durch (5.13) und (5.14) unter (5.34). Ist $d_k = 0$ gehe zu 3f).
   c) Bestimme α_k nach (5.37).
   d) Setze $x_{k+1} = x_k + \alpha_k d_k$.
   e) Ist $\alpha_k < 1$, füge den minimierenden Index zur aktiven Menge W_k hinzu und erhalte W_{k+1}. Setze $k = k + 1$ und gehe zu Schritt 3a).
   f) Berechne die Lagrange-Multiplikatoren von (5.33) und (5.34) und dann $\lambda_q = min_{i \in I \cap W_k} \lambda_i$. Ist $\lambda_q \geq 0$, so ist x_k optimal. Ansonsten entferne q aus W_k und erhalte W_{k+1}. Setze $k = k + 1$ und gehe zu Schritt 3a).

Abbildung 5.4: Aktive-Mengen-Strategie mit gegebenem Startwert

auftreten. Dies ist jedoch sehr unwahrscheinlich und die Methoden (meist eine geringfügige Störung der Variablen), mit denen dies vermieden werden kann, bergen eigene Schwierigkeiten. Aus diesem Grund wird dieser Fall vom vorgestellten Algorithmus - genau wie von vielen nicht-sicheren Implementierungen ([Fle87]) - ignoriert. Eine Möglichkeit, dieses Problem zu vermeiden, besteht darin, die aktive Menge in jedem Schritt bzw. in festgelegten Abständen neu zu berechnen.

5.5.5 Umsetzung als Mehrparteienberechnung

5.5.5.1 Bestimmung der Schrittweite

Formel (5.37) wird - wie in Protokoll 5.5 dargestellt - implementiert. Es wird zunächst das zweite Argument für jedes i berechnet. Nenner und Zähler liefern die Schritte 2-3 bzw. 6-8. Nicht-zulässige Elemente, d.h. solche, bei denen der Nenner ≥ 0 ist bzw. die zu Elementen der aktiven Menge korrespondieren,

Protokoll 5.5: $\left([\alpha],[\tau],\overrightarrow{[I]}\right) \leftarrow$ **Schrittweite** $\left([[A]],\overrightarrow{[d]},\overrightarrow{[x]},\overrightarrow{[b]},\overrightarrow{[w]},m,n\right)$

1	**For** $i = 1,\ldots,m$
2	$[ad(i)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[A(i,*)]},\overrightarrow{[d]},n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
3	$[ad(i)] \leftarrow \mathsf{TruncPr}([ad(i)],k,f)$//1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul, 2 Rnd $\mathbb{F}_{q_1}$
4	$[h(i)] \leftarrow \mathsf{LTZappr}([ad(i)],k)$ //vgl. Tabelle 2.1
5	$[l(i)] \leftarrow (1-[w(i)])$ AND $[h(i)]$ // 1 Mul $\mathbb{F}_q$, 1 Rnd
6	$[ax(i)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[A(i,*)]},\overrightarrow{[x]},n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
7	$[ax(i)] \leftarrow \mathsf{TruncPr}([ax(i)],k,f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
8	$[Z(i)] \leftarrow [b(i)] - [ax(i)]$
9	$[Z(i)] \leftarrow [l(i)] \cdot [Z(i)] - (1-[l(i)]) \cdot \infty$ //1 Mul, 1 Rnd $\mathbb{F}_q$
10	$[N(i)] \leftarrow [l(i)] \cdot [ad(i)] - (1-[l(i)]) \cdot 2^f$ //1 Mul, 1 Rnd $\mathbb{F}_q$
11	$\overrightarrow{[I]} \leftarrow \mathsf{MinBruch}\left(\overrightarrow{[Z]},\overrightarrow{[N]},m\right)$ //vgl. Tabelle 3.1
12	$[z] \leftarrow \mathsf{Inner}\left(\overrightarrow{[Z]},\overrightarrow{[I]},n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
13	$[n] \leftarrow \mathsf{Inner}\left(\overrightarrow{[N]},\overrightarrow{[I]},n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
14	$[\alpha] \leftarrow \mathsf{FPDiv}([z],[n],k,f)$ //vgl. Tabelle 2.1
15	$[\tau] \leftarrow \mathsf{LTappr}([\alpha],1 \cdot 2^f,k)$ //vgl. Tabelle 2.1
16	$[\alpha] \leftarrow [\tau] \cdot [\alpha] + (1-[\tau]) \cdot 2^f$ //1 Mul, 1 Rnd $\mathbb{F}_q$
17	**Return** $\left([\alpha],[\tau],\overrightarrow{[I]}\right)$

Abbildung 5.5: alpha: Bestimmung der Schrittweite α_k

werden im Zähler durch $-\infty$ (d.h. durch die kleinste zulässige Fixpunktzahl $-2^{k-1}+1$) und im Nenner durch -1 (Schritte 5,9-10) ersetzt. Auf diese Weise wird sichergestellt, dass alle Nenner dasselbe Vorzeichen besitzen und nichtzulässige Elemente das Ergebnis der Berechnung nicht beeinflussen. Die aktive Menge ist im Vektor $\overrightarrow{[w]}$ binär kodiert. Da die Division sehr teuer ist, wird Protokoll MinBruch verwendet, um den Index des Minimums zu identifizieren, und der Quotienten $[\alpha]$ nur an der Stelle berechnet, an der das Minimum zu finden ist. Am Ende (Schritt 15) wird noch das soeben berechnete alpha (Schritt 14) mit 1 verglichen und ggf. dadurch ersetzt (Schritt 16). Zurückgegeben wird neben $[\alpha]$ auch der binär kodierte Index, der Nebenbedingung an der das berechnete Minimum angenommen wird sowie das Bit $[\tau]$, das angibt, ob $[\alpha]$ gleich diesem Minimum oder gleich 1 ist (Schritt 17).

5.5.5.2 Berechnung von x_{k+1} und Aktualisierung der aktiven Menge

Es sei ein Startwert gefunden und die dort gültige aktive Menge bestimmt. Im Hauptteil des Algorithmus (Protokoll 5.6) sind nun die Schritte 3 a-f aus Abb.

Protokoll 5.6: $\overrightarrow{[x]} \leftarrow$ **Primal** $\left([[A]], [[G]], \overrightarrow{[c]}, \overrightarrow{[b]}, m\right)$

1 $\overrightarrow{[x]} \leftarrow$ Startwert$([[A]])$ //vgl. Abschnitt 5.5.3
2 $\overrightarrow{[w]} \leftarrow$ AktiveMenge $\left([[A]], \overrightarrow{[x]}, \overrightarrow{[b]}\right)$ // Protokoll 5.3
3 **While** (1)
4 $[[B]] \leftarrow$ MultipliziereZeilenweise $\left([[A]], \overrightarrow{[w]}, m\right)$ //mn Mul $\mathbb{F}_q$, 1 Rnd
5 $([[B]], [[P]]) \leftarrow$KompaktifiziereZeilenweise $\left([[B]], \overrightarrow{[w]}, n, m\right)$//vgl. Tabelle 3.1
6 $([[C]]) \leftarrow [[B(1:n, 1:n)]]$
7 $\overrightarrow{[y]} \leftarrow [[P]] \cdot \overrightarrow{[w]}$ //m Mul $\mathbb{F}_q$, 1 Rnd
8 $\overrightarrow{[w^k]} \leftarrow \overrightarrow{[y(1:n)]}$
9 $\overrightarrow{[g]} \leftarrow [[G]] \cdot \overrightarrow{[x]}$ //n Mul $\mathbb{F}_q$, 1 Rnd
10 $\overrightarrow{[g]} \leftarrow$ TruncPr $\left(\overrightarrow{[g]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $n(f+1)$ Mul $\mathbb{F}_{q_1}$
11 $\overrightarrow{[g]} \leftarrow \overrightarrow{[g]} + \overrightarrow{[c]}$
12 $\begin{pmatrix} \overrightarrow{[d]} \\ \overrightarrow{[\lambda']} \end{pmatrix} \leftarrow$ LinearSolve $\left(\begin{pmatrix} [[G]] & -[[C^t]] \\ [[C]] & 0 \end{pmatrix}, \begin{pmatrix} -\overrightarrow{[g]} \\ 0 \end{pmatrix}\right)$ //vgl. Tabelle 4.2
13 **For** $i = 1, \ldots, n$ **do parallel**
14 $[\lambda'(i)] \leftarrow [w^k(i)] \cdot [\lambda'(i)] + (1 - [w^k(i)]) \cdot \infty$ //1 Mul $\mathbb{F}_q$, 1 Rnd
15 $[\delta] \leftarrow$ EQZappr $\left(\overrightarrow{[d]}, n\right)$ //Protokoll 3.2
16 $\overrightarrow{[I]} \leftarrow$ MinVektor $\left(\overrightarrow{[\lambda]'}, n, k\right)$ //Protokoll 3.3
17 $[q] \leftarrow$ Inner $\left(\overrightarrow{[\lambda']}, \overrightarrow{[I]}, n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
18 $[\nu] \leftarrow$ GQZ$([q], k, f)$ //vgl. Tabelle 2.1
19 $a \leftarrow$ MulPub$([\nu], [\delta])$ //1 Mul, 1 Rnd $\mathbb{F}_q$
20 **If** $(a = 1)$
21 **Return** $\overrightarrow{[x]}$ //Optimaler Punkt erreicht
22 $\left([\alpha], [\tau], \overrightarrow{[K]}\right) \leftarrow$Schrittweite $\left([[B(1{:}n, 1{:}n)]], \overrightarrow{[d]}, \overrightarrow{[x]}, \overrightarrow{[b]}, \overrightarrow{[w^k]}, m\right)$//Prot. 5.5
23 $\overrightarrow{[d]} \leftarrow [\alpha] \cdot \overrightarrow{[d]}$ //n Mul $\mathbb{F}_q$
24 $\overrightarrow{[d]} \leftarrow$ TruncPr $\left(\overrightarrow{[d]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $n(f+1)$ Mul $\mathbb{F}_{q_1}$
25 $\overrightarrow{[x]} \leftarrow \overrightarrow{[x]} + (1 - [\delta]) \cdot \overrightarrow{[d]}$ //n Mul $\mathbb{F}_q$
26 $\overrightarrow{[\Lambda]} \hookleftarrow \overrightarrow{[I]}$
27 $\overrightarrow{[\Lambda]} \leftarrow [[P^t]] \cdot \overrightarrow{[\Lambda]}$ //m Mul $\mathbb{F}_q$, 1 Rnd; $3m{+}2$ Mul $\mathbb{F}_q$, 1 Rnd (Schritt 28)
28 $\overrightarrow{[w]} \leftarrow [\tau] \cdot (1 - [\delta]) \cdot \left(\overrightarrow{[w]} + \overrightarrow{[K]}\right) + (1 - [\tau]) \cdot (1 - [\delta]) \cdot \overrightarrow{[w]} + [\delta] \cdot \left(\overrightarrow{[w]} - \overrightarrow{[\Lambda]}\right)$

Abbildung 5.6: Die Primale Aktive Mengen Strategie

5.4 zu implementieren. Es wird LICQ (Definition 8) vorausgesetzt, d.h. insbesondere dass nie mehr als n Nebenbedingungen aktiv sind und somit nicht mehr als n Lagrange-Multiplikatoren $\neq 0$ sind.

Diese Eigenschaft wird bei der Kompaktifizierung der Nebenbedingungen und der aktiven Menge (Schritte 4-8) genutzt, so dass in der Matrix $[[C]]$ die aktiven Nebenbedingungen oben direkt untereinander stehen, gefolgt ggf. von Nullzeilen. Analog besitzt die kompaktifizierte Form $[w^k]$ des Aktiven-Mengen-Vektors $[w]$ in den entsprechenden Zeilen eine [1] und sonst eine [0].

Zur Laufzeit des Algorithmus soll nicht sichtbar werden, ob eine Nebenbedingung hinzugenommen oder entfernt wird. Deshalb muss in einer sicheren Implementierung in jedem Schritt d_k *und* λ_k berechnet werden. Man erhält diese als Lösung eines Gleichungssystems (Schritt 12, (5.35)). Da darin die kompaktifizierte Form $[[C]]$ von $[[A]]$ verwendet wird, besitzt es - unabhängig von der Größe der aktiven Menge - die Dimension $2n \times 2n$ (ggf. mit Nullzeilen und -spalten rechts und unten; dies stellt jedoch keine Einschränkung bzgl. der Lösbarkeit dar; vgl. Abschnitt 4.2.3). Die rechte Seite des Gleichungssystems wird nach Formel (5.33) gebildet, d.h. durch $\begin{pmatrix} -Gx_k - c & 0 \end{pmatrix}^t$ (Schritte 9-11).

Man erhält die berechneten Lagrange-Multiplikatoren in kompaktifizierter Form $\overrightarrow{[\lambda']}$ zurück und verwendet die kompaktifizierte Form $[w^k]$ der aktiven Menge, um nicht-aktive Komponenten von $\overrightarrow{[\lambda']}$ durch ∞ zu ersetzen (Schritt 14), wodurch sich dann mit Hilfe von Protokoll 3.3 der minimale, zur aktiven Menge korrespondierende, Lagrange-Multiplikator bestimmen lässt (Schritte 16-17). Die Überprüfung, ob $\overrightarrow{[d_k]} = 0$ geschieht mittels Protokolls EQZappr (Schritt 15). Im Anschluss wird $\overrightarrow{[d_k]}$ im Rahmen der Aktualisierung von $[x_k]$ mit dem Ergebnis des Vergleichs (Schritt 25) multipliziert, wodurch sichergestellt wird, dass bei einem Nullschritt keine auch noch so kleine Veränderung an $\overrightarrow{[x]}$ stattfindet. Ist $\overrightarrow{[d_k]} = 0$ und gleichzeitig der minimale Lagrangemultiplikator größer 0, ist der optimale Punkt erreicht und der Algorithmus wird abgebrochen (Schritte 18-21).

Ist dies nicht der Fall, wird $[\alpha_k]$ mit Hilfe von Protokoll 5.5 (Schritt 22) berechnet und - zusammen mit d_k - zur Berechnung des neuen Iterationspunkts x_{k+1} (Schritte 23-25) verwendet. Ist $d_k = 0$, wird ein Nullschritt ausgeführt. Um die aktive Menge anpassen zu können, muss der Index bestimmt werden, der im Falle eines Null-Schritts aus der aktiven Menge entfernt wird. Dazu bettet man (den kompaktifizierten Vektor) $\overrightarrow{[I]}$ in einen Vektor $\overrightarrow{[\Lambda]}$ der Länge m ein (Schritt 26) und macht die Kompaktifizierung mit Hilfe der Ma-

Protokoll	Runden	Multiplikationen	Körper
Skalierung	$k^2+k+2n+5$	$\mathcal{O}(n^2k+nk^2)$	$\mathbb{F}_q$
	2	$12nk$	$\mathbb{F}_{q_1}$
	$3\lceil\log_2 k\rceil+2$	$3kn(\lceil\log_2 k\rceil+2)-6n$	$\mathbb{F}_{2^8}$
SetupTableau	2	$2n^2+5n$	$\mathbb{F}_q$
	2	$4nk$	$\mathbb{F}_{q_1}$
	$\lceil\log_2 k\rceil+1$	$2n(2k-3)$	$\mathbb{F}_{2^8}$
Startwert (Iteration)	$4\lceil\log_2 n\rceil+2k+3\theta_{DivNR}+22$	$4n^2+44n+3\theta_{DivNR}+6$	$\mathbb{F}_q$
	2	$\mathcal{O}(n^2k+\theta_{DivNR}k)$	$\mathbb{F}_{q_1}$
	$\mathcal{O}(\log_2 n \log_2(k-1))$	$\mathcal{O}(k\log_2 k)$	$\mathbb{F}_{2^8}$
Aktive Menge	$2k+3$	$6n$	$\mathbb{F}_q$
	2	$2n\cdot(2k+f+1)$	$\mathbb{F}_{q_1}$
	$\log_2(k-1)+1$	$2n\cdot(2k-3)$	$\mathbb{F}_{2^8}$
Schrittweite	$4\log_2 n+4k+2f+25$	$36n+4\theta_{Div}+5$	$\mathbb{F}_q$
	2	$\mathcal{O}(nk+\theta_{Div}f)$	$\mathbb{F}_{q_1}$
	$6\lceil\log_2 k\rceil+\lceil\log_2 m\rceil\cdot\left(\frac{3}{2}\lceil\log_2(k-1)\rceil\right)+5$	$\mathcal{O}(k\log_2 k+nk)$	$\mathbb{F}_{2^8}$
Primale Iteration	$\mathcal{O}(n^3+\theta_{DivNR}n+k)$	$\mathcal{O}(n^3+\theta_{DivNR}n)$	$\mathbb{F}_q$
	2	$\mathcal{O}(n^3f+n^2k+n\theta_{Div}kn)$	$\mathbb{F}_{q_1}$
	$\mathcal{O}(n\log_2(k-1))$	$\mathcal{O}(n^2k+nk\log_2 k)$	$\mathbb{F}_{2^8}$

Tabelle 5.1: Kosten der beim primalen Optimierungsalgorithmus auftretenden Protokolle für $m = 2n$ Nebenbedingungen

trix $[[P^t]]$ rückgängig (Schritt 27). Mit diesen Informationen ist es möglich, die aktive Menge zu aktualisieren (Schritt 28). Ist α gleich 1, bleibt die aktive Menge unverändert, ist sie kleiner als 1, wird die neu aktivierte Nebenbedingung hinzugenommen und ist $d_k = 0$ wird der Index des kleinsten (negativen) Lagrange-Multiplikators aus der aktiven Menge entfernt.

Der für das Lösen des linearen Gleichungssystems verwendete Algorithmus (Schritt 10) wird hier bewusst offengelassen. Welches Verfahren am effizientesten ist, hängt von den gewählten Parametern ab. In der Regel jedoch wird dies die QR-Zerlegung sein. Es sind aber auch Konfigurationen denkbar, in denen die LR-Zerlegung konkurrenzfähig ist (vgl. Abschnitt 4).

5.5.6 Gleichheitsnebenbedingungen

Sind zusätzlich zu den Ungleichheitsnebenbedingungen auch Gleichheitsnebenbedingungen gegeben, *müssen* diese Teil jeder aktiven Menge und am Optimum erfüllt sein. Gleiches gilt für den Startwert. Müssen Nebenbedingungen fallengelassen werden, werden Gleichheitsnebenbedingungen nicht betrachtet (vgl. Abb. 5.4).

Zusätzlich kann die Suche nach einem zulässigen Startwert auch anders gestaltet werden: Für Gleichheitsnebenbedingungen werden *künstliche* Variablen und für Ungleichheitsnebenbedingungen *Schlupfvariablen* eingeführt. Minimiert wird dann nur die Summe der künstlichen Variablen (vgl. [Fle87]).

5.6 Beschreibung des Algorithmus von Goldfarb und Idnani

Der Algorithmus von Goldfarb und Idnani ([GI83]) ist ein *dualer* Algorithmus. Er kann vom unbeschränkten Minimum gestartet werden, es ist also keine Phase I notwendig. Weiterhin legen empirische Befunde nahe, dass das Minimum meistens in wenigen Schritten erreicht werden kann ([GI83], Abschnitt 7.7.2.2). Er hat sich als robust erwiesen ([Pow85]) und ist Teil vieler Software-Optimierungspakete ([MW93],[Sch02]).

5.6.1 Notation

Wegen (5.8) kann sich die Betrachtung auf die in x^* aktiven Nebenbedingungen beschränken. Im Folgenden sei die Einschränkung von A^t auf die Spalten, die an x^* aktiven Nebenbedingungen entsprechen, mit N bezeichnet. Gleichung (5.10) wird so (unter stillschweigender Verkürzung von λ^* auf die Breite von N) zu

$$Gx^* + c = N\lambda^*. \tag{5.38}$$

5.6.2 Operatoren

Für die effiziente Darstellung des Algorithmus werden einige Definitionen und Operatoren benötigt. Diese sollen hier kurz vorgestellt werden.

5.6.2.1 Die gewichtete Pseudoinverse

Definition 10 (Pseudoinverse)**.** *Sei $A \in M_{m,n}(\mathbb{R})$ eine Matrix, dann ist die* Pseudoinverse *([GL96]) zu A die eindeutig definierte Matrix X für die gilt*

$$AXA = A \tag{5.39}$$
$$XAX = X \tag{5.40}$$
$$(AX)^t = AX \tag{5.41}$$
$$(XA)^t = XA \tag{5.42}$$

Besitzt A vollen Spaltenrang, so ist die Pseudoinverse $\tilde{A}$ definiert durch

$$\tilde{A} = \left(A^t A\right)^{-1} A^t. \tag{5.43}$$

Da die Matrix G aus (5.1) symmetrisch positiv definit ist, existiert eine Matrix X, so dass $X^2 = G$ ([GL96]). Man bezeichnet X auch mit $G^{\frac{1}{2}}$. Betrachtet man nun die Zielfunktion (5.1) unter der Operation

$$x \mapsto \tilde{x} = G^{\frac{1}{2}} x, \tag{5.44}$$

so transformiert sie sich zu

$$\frac{1}{2}\tilde{x}^t\tilde{x} + \tilde{c}^t\tilde{x}. \tag{5.45}$$

Dabei ist $\tilde{c} = G^{-\frac{1}{2}} c$. Die Zielfunktion wird nun nicht mehr durch die Matrix G „verzerrt“. Entsprechend transformiert sich N zu $G^{-\frac{1}{2}} N$ und man erhält die mit G *gewichtete Pseudoinverse* N^*. Da diese auf die mit (5.44) transformierten $\tilde{x}$ anzuwenden ist, es aber zweckmäßiger ist, die untransformierten x zu verwenden, wird das dafür notwendige $G^{-\frac{1}{2}}$ in die Matrix N^* hineinmultipliziert. Man erhält:

$$N^* = \left(N^t G^{-1} N\right)^{-1} N^t G^{-1}. \tag{5.46}$$

Davon abgeleitet ist noch der Operator

$$H = G^{-1}\left(Id - NN^*\right) \tag{5.47}$$

definiert.

Bemerkung: Man kann die Transformation (5.44) auch als Mittel betrachten mit dem sichergestellt wird, dass die Hessematrix des transformierten Problems (5.45) an der unbeschränkten Lösung x^* gut konditioniert ist. Dies erlaubt es, x^* mit guter Genauigkeit zu berechnen ([GMW81]).

Die Operatoren haben folgende Eigenschaften ([GI83]):

$$Hw = 0 \Leftrightarrow w = N\alpha \tag{5.48}$$

$$H \text{ ist positiv semidefinit} \tag{5.49}$$

$$HGH = H \tag{5.50}$$

$$N^*GH = 0 \tag{5.51}$$

Zusätzlich gilt trivialerweise

$$N^*N = id \tag{5.52}$$

5.6.3 Definitionen

Die im Folgenden beschriebenen Begriffe *S-Paar* und *V-verletztes-Tripel* werden zur Beschreibung des dualen Algorithmus von Goldfarb/Idnani ([GI83]) benötigt.

Definition 11 (S-Paar)**.** *Ist $K = \{1, \ldots, m\}$ die Menge der Nebenbedingungen und $J \subset K$, so bezeichnet $P(J)$ das quadratische Optimierungsproblem, gegeben durch die Einschränkung der Nebenbedingungen auf J. Liegt die Lösung x des Teilproblems $P(J)$ auf einer linear unabhängigen Teilmenge $A \subset J$, so heißt das Tupel (x, A) ein* Lösungs-*S*-Paar.

Es sei $K \subset \{1, \ldots, m\}$ eine Menge von Indizes und $p \in \{1, \ldots, m\} \backslash K$, dann sei K^+ die Menge $K \cup p$. N^+ und H^+ sind dann die Operatoren, deren Definitionen um die entsprechenden Nebenbedingungen erweitert wurden. Es gilt ([GI83]):

$$HH^+ = H^+ \tag{5.53}$$

Definition 12 (V-(verletztes)-Tripel)**.** *Ein Tripel (x, K, p) bestehend aus einem Punkt x und einer Menge Indizes K (w.o.) und $p \in \{1, \ldots, m\} \backslash K$ heißt* V-(verletztes)-Tripel*, wenn die zu K^+ korrespondierenden Nebenbedingungen linear unabhängig sind und die Bedingungen*

1. $a_p^t x - b_p < 0$
2. $a_i^t x - b_i = 0 \quad \forall i \in K$
3. $H^+ \nabla_x f(x) = 0$
4. $u^+(x) \equiv \left(N^+\right)^* \nabla_x f(x) \geq 0$

erfüllt sind ([GI83]).

Bemerkung: Die letzten beiden Bedingungen lassen sich mit Hilfe von (5.55), Folgerung 1 und (5.48) zeigen ([GI83]).

5.6.4 Eigenschaften und geometrische Interpretation der Operatoren

N^* erfüllt (5.39) und (5.40), aber nicht (5.41) und (5.42). Die beiden letzteren sind hier allerdings auch nicht von Bedeutung.

Geometrisch ist für einen Vektor x die Abbildung NN^*x die orthogonale Projektion von x auf das Bild von N und die Abbildung $(Id - NN^*)x$ die Differenz des Punktes x zu seiner orthogonalen Projektion auf $im(N)$. $G^{-1} = (H(f(x)))^{-1}$ ist das Inverse der Hessematrix $H(f(x))$ von f. Ist $\bar{x}$ Lösung des Teilproblems $P(A)$, so gilt nach (5.38)

$$\nabla f(\bar{x}) = Nu(\bar{x}), \tag{5.54}$$

und dementsprechend

$$\begin{aligned}
u(\bar{x}) &= N^*\nabla f(\bar{x}) && (5.55)\\
&= N^*(G\bar{x} + c) && (5.56)\\
&= \left(N^tG^{-1}N\right)^{-1} N^tG^{-1} \cdot (G\bar{x} + c) && (5.57)\\
&= \left(N^tG^{-1}N\right)^{-1} N^tx + \left(N^tG^{-1}N\right)^{-1} N^tG^{-1}c && (5.58)\\
&= \left(N^tG^{-1}N\right)^{-1} b + \left(N^tG^{-1}N\right)^{-1} N^tG^{-1}c && (5.59)
\end{aligned}$$

$u(\bar{x})$ ist hierbei der Vektor der Lagrange-Multiplikatoren für Teilproblem $P(A)$ an der Stelle $\bar{x}$. Insbesondere sieht man, dass er *nicht* von $\bar{x}$ abhängt und konstant auf dem affinen Unterraum $\mathcal{M} = \{x \in \mathbb{R}^n | n_i^t x = b_i, i \in A\}$ ist! Es lässt sich also an jedem Punkt von $\mathcal{M}$ sagen, ob das Optimum von (5.1) auf $\mathcal{M}$ dual zulässig ist, ohne es explizit zu bestimmen! Dies ist eine Folge der Transformation (5.44) und wird im Algorithmus eine wichtige Rolle spielen.

Folgerung 1. *Sei $x \in \mathcal{M}$ beliebig. Dann ist das Minimum von f auf $\mathcal{M}$ gegeben durch $\bar{x} = x - H\nabla_x f(x)$.*

Beweis: Zeige zunächst $\bar{x} \in \mathcal{M}$. Es gilt

$$\begin{aligned}
N^t\bar{x} &= N^t(x - H\nabla_x f(x)) && (5.60)\\
&= N^tx - N^tG^{-1}(Id - NN^*)(Gx + c) && (5.61)\\
&= b - N^t(G^{-1} - G^{-1}NN^*)(Gx + c) && (5.62)\\
&= b - N^t\left(x - G^{-1}NN^*Gx + G^{-1}c - G^{-1}NN^*c\right) && (5.63)
\end{aligned}$$

1. Sei ein S-Paar (x, A) gegeben
2. Wiederhole bis alle Nebenbedingungen erfüllt sind
   a) Wähle eine verletzte Nebenbedingung $p \in K \backslash A$
   b) Ist $P(A \cup \{p\})$ unzulässig. STOP. Das Problem ist nicht lösbar.
   c) Sonst, finde neues S-Paar $(\bar{x}, \bar{A} \cup \{p\})$ mit $\bar{A} \subset A$ und $f(\bar{x}) > f(x)$ und setze $(x, A) \leftarrow (\bar{x}, \bar{A} \cup \{p\})$
3. STOP. x ist die optimale Lösung.

Abbildung 5.7: Grobstruktur des Algorithmus von Goldfarb und Idnani

$$= b - b + N^t G^{-1} N \left(N^t G^{-1} N \right)^{-1} b - N^t G^{-1} c + N^t G^{-1} N N^* \quad (5.64)$$

$$= b - N^t G^{-1} c + N^t G^{-1} c \quad (5.65)$$

$$= b \quad (5.66)$$

Da G positiv definit ist, reicht es zu zeigen, dass $\nabla_x f(\bar{x}) = N \cdot l$, für den Lagrange-Multiplikator l. Es gilt:

$$\begin{aligned} \nabla_x f(\bar{x}) &= G\bar{x} + c \\ &= Gx - GH(Gx + c) + c \\ &= Gx - (Id - NN^*) \cdot (Gx + c) + c \\ &= NN^* Gx + NN^* c \\ &= NN^* \cdot \nabla_x f(x) \end{aligned}$$

$N^* \cdot \nabla_x f(x)$ ist aber gleich $u(\bar{x})$ ((5.55)-(5.59)). Somit ist $l = N^* \cdot \nabla_x f(x)$ der gesuchte Lagrange-Multiplikator. □

5.6.5 Der Algorithmus

Der Algorithmus konstruiert eine Folge dual zulässiger Punkte, die aber - bis auf den letzten - primal nicht zulässig sind. Sie sind also Lösung der jeweiligen Teilprobleme $P(A^k)$ (k Iterationsparameter). Als Startwert kann jeder dual zulässige Punkt gewählt werden. Sind keine Informationen über die Gestalt der Lösung der aktiven Menge verfügbar, setzt man üblicherweise x_0 als das

globale Minimum $-G^{-1}c$. Ist nach einer Reihe von Iterationen keine Nebenbedingung mehr verletzt, ist das Optimum gefunden und der Algorithmus bricht ab. Seine Grobstruktur hat die in Abb. 5.7 gezeigte Gestalt.

Der Algorithmus von Goldfarb und Idnani

SCHRITT 0: Bestimme das globale Minimum

1 $x \leftarrow -G^{-1}c$
2 $f \leftarrow \frac{1}{2}c^t x$
3 $H \leftarrow G^{-1}$
4 $W \leftarrow \emptyset$
5 $q \leftarrow 0$

SCHRITT 1: Finde eine verletzte Nebenbedingung

6 $S \leftarrow A \cdot x - b$
7 Falls $S \geq 0$: x ist optimaler Punkt. **Stop**.
8 Sonst: Wähle p, d.d. $S(p) < 0$.
9 $n^+ \leftarrow n_p$
10 $u^+ \leftarrow \begin{pmatrix} u & 0 \end{pmatrix}^t$
11 **If** $(q = 0)$
12 $\quad u \leftarrow 0$

SCHRITT 2: Überprüfe Zulässigkeit und finde neues S-Paar

a) Bestimme Korrekturterm

13 $z \leftarrow Hn^+$
14 **If** $(q > 0)$
15 $\quad r \leftarrow N^* n^+$

b) Bestimme Schrittweite

i) Partielle Schrittweite t_1

16 **If** ($r \leq 0$ oder $q = 0$)
17 $\quad t_1 \leftarrow \infty$
18 **Else**
19 $\quad t_1 \leftarrow \min\limits_{r_j > 0, j=1,\ldots,q} \left\{ \frac{u_j^+(x)}{r_j} \right\} = \frac{u_l^+(x)}{r_l}$

ii) Volle Schrittweite t_2

20 **If** $(|z| = 0)$
21 $\quad t_2 \leftarrow \infty$
22 **Else**
23 $\quad t_2 \leftarrow -\frac{S_p(x)}{z^t n^+}$

iii) Bestimme Schrittweite t

24 $t \leftarrow \min\{t_1, t_2\}$

c) Bestimme neues S-Paar und gehe Korrekturschritt

Der Algorithmus von Goldfarb und Idnani

i) Kein Schritt in den primären oder dualen Variablen
If $(t = \infty)$
    Stop. Kein zulässiger Bereich.
ii) Schritt in den dualen Variablen
If $(t_2 = \infty)$
    $u^+ \leftarrow u^+ + t \begin{pmatrix} -r \\ 1 \end{pmatrix}$
    $W \leftarrow W \backslash \{l\}$
    $q \leftarrow q - 1$
    Aktualisiere N^* und H
    Gehe zu Schritt 2a
iii) Schritt in den primalen und dualen Variablen
$x \leftarrow x + tz$
$f \leftarrow f + tz^t n^+ \cdot \left(\frac{1}{2}t + u^+_{q+1}\right)$
$u^+ \leftarrow u^+ + t \begin{pmatrix} -r \\ 1 \end{pmatrix}$
If $(t = t_2)$
    $u \leftarrow u^+$
    $W \leftarrow W \cup \{p\}$
    $q \leftarrow q + 1$
    Aktualisiere N^* und H
    Gehe zu Schritt 1
If $(t = t_1)$
    $W \leftarrow W \backslash \{l\}$
    $q \leftarrow q - 1$
    Aktualisiere N^* und H
    Gehe zu Schritt 2a

Tabelle 5.2: Der Algorithmus von Goldfarb und Idnani

Der interessanteste - und aufwendigste - Schritt ist Schritt 2c. Da keine duale Variable kleiner als Null werden darf, erkennt der Algorithmus mit Hilfe der Schrittweite, ob eine neue Nebenbedingung hinzugenommen werden kann (*voller Schritt*) oder ob zuerst eine aktive fallengelassen werden muss (*halber Schritt*). Die Details, wie sie in [GI83] beschrieben sind, sind in Tabelle 5.2 zu sehen.[1]

[1] Es gibt auch andere Implementierungen wie z.B. die in [Pow85] vorgestellte. Diese sind der hier präsentierten jedoch nicht überlegen und eignen sich nicht ganz so gut für eine Implementierung als sichere Mehrparteienberechnung.

5.6.6 Die Implementierung von N^* und H

Ist in Schritt 0 das unbeschränkte Minimum bestimmt, wird in Schritt 1 $H = G^{-1}$ benötigt (Setze $N = \emptyset$ in (5.47)). Der erste Korrekturschritt z_1 ist bestimmt durch die Lösung des Gleichungssystems

$$G \cdot z_1 = n^+. \tag{5.67}$$

r ist im 1. Schritt gleich 0.
In den darauffolgenden Iterationen ist die Berechnung aufwendiger. Grundlage ist eine Cholesky-Zerlegung der Matrix G

$$G = LL^t. \tag{5.68}$$

Mit Hilfe von (5.68) lässt sich N^* darstellen als

$$N^* = \left(N^t G^{-1} N\right)^{-1} N^t G^{-1} = \left(\left(L^{-1}N\right)^t L^{-1}N\right)^{-1} \left(L^{-1}N\right)^t L^{-1}. \tag{5.69}$$

Für die Matrix H ergibt sich dann:

$$\begin{aligned} H &= G^{-1}\left(Id - NN^*\right) && (5.70)\\ &= L^{-t}L^{-1}\left(Id - N\left(\left(L^{-1}N\right)^t L^{-1}N\right)^{-1}\left(L^{-1}N\right)^t L^{-1}\right) && (5.71)\\ &= L^{-t}\left(Id - \left(L^{-1}N\right)\left(\left(L^{-1}N\right)^t L^{-1}N\right)^{-1}\left(L^{-1}N\right)^t\right)L^{-1} && (5.72)\\ & && (5.73) \end{aligned}$$

Sei nun eine QR-Zerlegung der Matrix B

$$B = L^{-1}N = Q \cdot R = [Q_1|Q_2] \cdot \begin{pmatrix} R \\ 0 \end{pmatrix} = Q_1 R, \tag{5.74}$$

gegeben, wobei Q_1 aus den ersten q Spalten von Q und Q_2 aus den zweiten $n - q$ Spalten besteht, gegeben. Dann ergibt sich für N^*

$$N^* = \left((Q_1R)^t Q_1 R\right)^{-1}(Q_1R)^t L^{-1} = \left(R^t R\right)^{-1} R^t Q_1 L^{-1} = R^{-1}Q_1 L^{-1}, \tag{5.75}$$

und für H

$$
\begin{aligned}
H &= L^{-t}\left(Id - Q_1 R R^{-1} Q_1^t\right) L^{-1} && (5.76)\\
&= L^{-t}\left(Id - Q_1 Q_1^t\right) L^{-1} && (5.77)\\
&= L^{-t} Q_2 Q_2^t L^{-1} && (5.78)\\
&= \left(L^{-t} Q_2\right)\left(L^{-t} Q_2\right)^t. && (5.79)
\end{aligned}
$$

Im letzten Schritt wird dabei die Identität

$$Id = QQ^t = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} Q_1^t \\ Q_2^t \end{pmatrix} = Q_1 Q_1^t + Q_2 Q_2^t$$

verwendet. Es reicht also die Matrix

$$J = (J_1 | J_2) = (L^{-t} Q_1 | L^{-t} Q_2) = L^{-t} Q \tag{5.80}$$

zu berechnen. Die Matrizen N^* und H sind dann gegeben durch:

$$H = J_2 J_2^t$$

und

$$N^* = R^{-1} J_1^t.$$

Es ist allerdings nicht nötig, diese explizit zu berechnen, da sie nur zur Berechnung von z und r dienen (Zur Bedeutung von r mehr im nächsten Abschnitt). Diese erhält man jedoch bereits durch

$$z = J_2 d_2 \tag{5.81}$$

und

$$r = R^{-1} d_1, \tag{5.82}$$

wobei

$$d = J^t n^+ = \begin{pmatrix} J_1^t \\ J_2^t \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \tag{5.83}$$

ist.
Die in [GI83] vorgeschlagene Implementierung geht noch weiter: Sie unterscheidet zwischen Hinzunahme und Entfernen einer Nebenbedingung. Dies ist bei einer Umsetzung als sichere Mehrparteienberechnung kaum möglich, ohne Informationslecks zu verursachen, d.h. den Spielern zur Laufzeit Informationen über die Gestalt des Problems zu geben und so u.U. vertrauliche Eingabewerte zu kompromittieren. Es müssten also in jedem Schritt beide Alternativen simultan verfolgt werden. Im Vergleich dazu ist es jedoch billiger

die Matrizen neu zu berechnen. Wesentliche Schritte der Implementierung von [GI83] beruhen außerdem darauf, Matrizen nur an bestimmten, öffentlich bekannten Stellen zu manipulieren, was in einem sicheren Mehrparteienprotokoll nur mit Protokollen, wie den in Abschnitt 3.1.2 beschriebenen, möglich ist. Diese Protokolle, insbesondere bei wiederholter Anwendung auf Matrizen, sind jedoch sehr teuer. Zudem hängt die Größe mancher Matrizen von der Größe der aktiven Menge ab. Das Rechnen mit einer solchen Matrix würde ein weiteres Informationsleck bedeuten. Deswegen muss hier die Unterscheidung zwischen Hinzunahme und Entfernen einer Nebenbedingung entfallen.

5.6.7 Die Suchrichtung z und die Schrittlänge

Es wird zunächst ein Lemma benötigt

Lemma 1. *Sei $i \in W_k$ und $n_i^t = a_i$ die entsprechende Zeile von A. Dann gilt für alle $p \in \{1, \ldots, m\} \backslash W_k$*

$$n_i^t H n_p = 0$$

Beweis:

$$\begin{aligned} n_i^t H n_p &= n_i^t G^{-1}(Id - NN^*)n_p \\ &= n_i^t G^{-1} n_p - n_i^t G^{-1} N (N^t G^{-1} N)^{-1} N^t G^{-1} n_p \\ &= n_i^t G^{-1} n_p - v_i^t (N^t G^{-1} N)^{-1} N^t G^{-1} n_p \end{aligned}$$

Dabei ist v_i^t die $i-$te Zeile von $N^t G^{-1} N$. Man erhält also:

$$\begin{aligned} &= n_i^t G^{-1} n_p - \begin{pmatrix} 0 & \ldots & 0 & 1_i & 0 & \ldots & 0 \end{pmatrix} \cdot N^t G^{-1} n_p \\ &= n_i^t G^{-1} n_p - n_i^t G^{-1} n_p \\ &= 0 \end{aligned}$$

□

Nun ist es möglich, eine Aussage über die Suchrichtung zu treffen. Dazu benötigt wird die Identität

$$GHn^+ = (Id - NN^*)n^+ = n^+ - Nr = N^+ \cdot \begin{pmatrix} -r \\ 1 \end{pmatrix}. \tag{5.84}$$

Dabei sei n^+ die neu hinzugenommene Nebenbedingung und

$$r := N^* n^+. \tag{5.85}$$

Satz 3. *Sei (x, K, p) ein V-Tripel. Dann wird das quadratische Optimierungsproblem auf $\mathcal{M}^+ = \{x \in \mathbb{R}^n | a_i x = b_i \, \forall i \in K \cup \{p\}\}$ minimiert durch*

$$\bar{x} = x + t_2 z,$$

wobei

$$z = Hn^+ \quad \textit{und} \quad t_2 = -\frac{a_p^t x - b_p}{z^t n^+}. \tag{5.86}$$

Beweis: Zeige zunächst $\bar{x} \in \mathcal{M}^+$. Sei $i \in K$, dann gilt

$$\begin{aligned} a_i \bar{x} - b_i &= a_i x - b_i + t_2 a_i H n^+ \\ &= 0 + t_2 a_i H n^+ \\ &= 0. \end{aligned}$$

Die letzte Gleichheit mit 0 folgt aus Lemma 1. $a_p \bar{x} - b_p = 0$ folgt nach Definition (5.86). Zeige weiterhin $H^+ \nabla f(\bar{x}) = 0$. Sei zunächst $t > 0$ beliebig. Dann gilt mit (5.47), (5.84) und (5.85)

$$\begin{aligned} H^+ \nabla f(\bar{x}) &= H^+ \nabla f(x + tz) \\ &= H^+ \nabla f(x) + t H^+ G z \\ &= 0 + t H^+ N^+ \begin{pmatrix} -r \\ 1 \end{pmatrix} \\ &= 0 \end{aligned}$$

Dabei ist $H^+ \nabla f(x) = 0$, da (x, K, p) V-Tripel. Der 2. Summand ist $= 0$, da $HN = G^{-1}(Id - NN^*)N = G^{-1}(N - NN^*N) = G^{-1}(N - N) = 0$ und somit auch $H^+ N^+ = 0$. Nun folgt mit Folgerung 1 die Behauptung. □

Die duale Zulässigkeit der Iterationspunkte muss zu jedem Zeitpunkt gewahrt bleiben. Wird durch den Schritt $t_2 z$ die duale Zulässigkeit verletzt, muss er soweit verkürzt werden, dass kein Lagrange-Multiplikator u kleiner als 0 wird. Die Auswirkungen eines Schritts tz auf u - und somit auch Möglichkeiten zur Aktualisierung von u nach einem Schritt tz - sind in folgendem Satz beschrieben.

Satz 4. *Seien $(x, K, p), \bar{x}, z$ wie in Satz 3. Dann gilt*

$$u^+(\bar{x}) = (N^+)^* \nabla f(\bar{x}) = u^+(x) + t \begin{pmatrix} -r \\ 1 \end{pmatrix}$$

Beweis:

$$\begin{aligned} u^+(\bar{x}) &= (N^+)^* \nabla f(\bar{x}) \\ &= (N^+)^* (G(x + tz) + c) \\ &= (N^+)^* (Gx + c) + t(N^+)^* GHn^+ \\ &= u^+(x) + t(N^+)^* N^+ \cdot \begin{pmatrix} -r \\ 1 \end{pmatrix} \\ &= u^+(x) + t \cdot \begin{pmatrix} -r \\ 1 \end{pmatrix} \end{aligned}$$

□

Die Schrittlänge t muss also $\leq \frac{u_i^+(x)}{r_i}$ für alle i mit $r_i > 0$ erfüllen. Der Vektor r ist demnach ein Hilfsvektor, dessen Komponenten dazu verwendet werden die maximale Schrittlänge zu bestimmen, so dass der resultierende Punkt dual zulässig bleibt. Geht man nur einen solchen verkürzten Schritt, wird ein bisher positiver Lagrange-Multiplikator zu 0, es kann also dann auf die entsprechende Nebenbedingung verzichtet (5.8) werden. Sie wird fallengelassen.

5.7 Umsetzung als Mehrparteienberechnung

Die Darstellung der Implementierung folgt der Einteilung des Algorithmus in die oben angesprochenen Teilschritte. Auf die Unterscheidung *Gehe-zu Schritt 1/2a* wird verzichtet. Schritt 1, die Auswahl einer verletzten Nebenbedingung, wird also in jeder Iteration durchgeführt, unabhängig davon, ob der vorangegangene Schritt ganz oder halb ist.

Die Unterscheidungen $t = t_1, t_2$ und $t_2 \stackrel{?}{=} \infty$, die ein unterschiedliches, weiteres Vorgehen nach sich ziehen, werden mit Hilfe von Schaltervariablen implementiert (Abschnitt 1.3.1). Die äußere Schleife kann entweder als endlose **while**-Schleife implementiert werden, die bei Erreichen des optimalen Punkts bzw. bei Unlösbarkeit des Problems abgebrochen wird, oder nach einer festen Anzahl an Iterationen, wenn vorher kein Abbruchkriterium erfüllt wurde. Da letzteres eine Beendigung des Programms auch im Fall von Zyklen, die nicht ganz ausgeschlossen werden können, garantiert, wurde diese Variante gewählt. Die Teilnehmer können sich nach Durchlaufen aller Schleifen am Ende ausgeben lassen, ob noch ein Lagrange-Multiplikator < 0 ist, so erkennen, ob das Optimum gefunden wurde und das Programm ggf. fortsetzen bzw. mit einer höheren Iterationszahl neu starten. Diese Variante spielte jedoch bei keinem der durchgeführten Tests eine Rolle und ist in den Darstellungen - der Einfachheit halber - nicht berücksichtigt. Der gesamte Algorithmus ist in Protokoll 5.8 abgebildet.

Protokoll 5.8: $\left(\overrightarrow{[x]}, \overrightarrow{[u]}, \overrightarrow{[W]}\right) \leftarrow \mathsf{GI}\left([[G]], \overrightarrow{[c]}, \overrightarrow{[b]}, [[A]], m, n, l\right)$

1 $\left(\overrightarrow{[x]}, [[G_C]], \overrightarrow{[s]}, \overrightarrow{[S]}, [F]\right) \leftarrow \mathsf{Schritt0}\left([[G]], \overrightarrow{[c]}, m\right)$ //Protokoll 5.9
2 **For** $(i = 1, \ldots, l)$
3 **If** (1. Iteration)
4 $\left(\overrightarrow{[n_p]}, \overrightarrow{[n^l]}\right) \leftarrow \mathsf{Schritt1}\left([[A]], \overrightarrow{[s]}, \overrightarrow{[S]}, m, n\right)$ //Protokoll 5.10
5 $\left(\overrightarrow{[x]}, \overrightarrow{[u^+]}, \overrightarrow{[u]}, \overrightarrow{[W]}, [[P]], \overrightarrow{[S]}, [F]\right) \leftarrow \mathsf{Schritt2}\left(\overrightarrow{[x]}, [F]\right)$ //Protokoll 5.13
6 **Else**
7 $\left(\overrightarrow{[n_p]}, \overrightarrow{[n^l]}\right)_{neu} \leftarrow \mathsf{Schritt1}\left([[A]], \overrightarrow{[s]}, \overrightarrow{[S]}, m, n\right)$ //Protokoll 5.10
8 $\left(\overrightarrow{[x]}, \overrightarrow{[u^+]}, \overrightarrow{[u]}, \overrightarrow{[W]}, [[P]], \overrightarrow{[S]}, [f]\right) \leftarrow \mathsf{Schritt2}\left(\overrightarrow{[x]}, \overrightarrow{[u]}, \overrightarrow{[u^+]}, \overrightarrow{[W]}, [[P]], [f]\right)$ //P. 5.13

Abbildung 5.8: Implementierung des Algorithmus von Goldfarb und Idnani

5.7.1 Vorbemerkung

Sind insgesamt $m \geq n$ Nebenbedingungen vorhanden, so können unter der Voraussetzung (8) immer die n aktiven in einer $n \times n$-Matrix zusammengefasst werden, indem mit Protokoll 3.8 (Kompaktifiziere) die aktiven Nebenbedingungen in die ersten n Zeilen - unter Beibehaltung der Reihenfolge - permutiert werden. Analog kann bei dem Vektor vorgegangen werden, der die aktive Menge binär kodiert sowie bei anderen Vektoren und Matrizen, die durch die aktive Menge indiziert werden. Für - im Vergleich zu n - großes m ist so eine dramatische Verbesserung der Komplexität möglich. Eine solche Darstellung wird im Folgenden auch als *kurze Form* des Vektors bzw. der Matrix bezeichnet. Die Schreibweise für einen so verkürzten Vektor $\overrightarrow{[v]}$ ist $\overrightarrow{[v^C]}$ und für eine Matrix $[[A]]$ $[[A^C]]$. Die Nicht-Degeneriertheit der Nebenbedingungen wird von nun an - wie in Abschnitt 5.5.5 - vorausgesetzt.

5.7.2 Berechnung der verletzten Nebenbedingung

Im Gegensatz zu dem in (5.2) beschriebenen Vorgehen wird in Schritt 2 ein neuer Iterationspunkt nicht am Anfang einer jeden Iteration auf Gültigkeit, sondern am Ende, sobald er vorliegt (so wie in Schritt 1 (Abb. 5.10)), überprüft. Im Anschluss daran können dann die verletzten Nebenbedingungen berechnet werden. Dies erspart es, die Matrizen B, Q, R und J neu berechnen zu müssen, wenn das Optimum erreicht ist. Geht es nur um die Lösung eines quadratischen Optimierungsproblems, ist der dadurch resultierende Effizienzgewinn bei einer zunehmenden Anzahl an Iterationen vernachlässigbar. Anders

Protokoll 5.9: $\left(\overrightarrow{[x]}, [[G_C]], \overrightarrow{[s]}, \overrightarrow{[S]}, [F]\right) \leftarrow$ **Schritt0** $\left([[G]], \overrightarrow{[c]}, [[A]], \overrightarrow{[b]}, m\right)$

1 $\left([[G_C]], \overrightarrow{[x]}\right) \leftarrow \mathsf{CholeskySolve}\left([[G]], \overrightarrow{[-c]}\right)$ //Protokoll 4.12
2 $[F] \leftarrow \mathsf{Inner}([c], [x])$ //1 Mul, 1 Rnd F_q
3 $[F] \leftarrow \mathsf{TruncPr}([F], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
4 $[F] \leftarrow \mathsf{DivKonst}([F], 2)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
5 $\overrightarrow{[s]} \leftarrow [[A]] \cdot \overrightarrow{[x]}$// m Mul, 1 Rnd $\mathbb{F}_q$
6 $\overrightarrow{[s]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[s]}, k, f\right)$ //m Mul, 1 Rnd $\mathbb{F}_q$, $m(f+1)$ Mul $\mathbb{F}_{q_1}$
7 $\overrightarrow{[s]} \leftarrow \overrightarrow{[s]} - \overrightarrow{[b]}$
8 **For**$(i = 1, \ldots, m)$ **do parallel**
9 $[S(i)] \leftarrow \mathsf{LTZappr}([s(i)], \varepsilon, k)$ //vgl. Tabelle 2.1
10 $[V] \leftarrow \sum_{i=1}^{m} [S(i)]$
11 $v \leftarrow \mathsf{EQZPub}([V], k)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
12 **If**$(v = 1)$
13 Optimaler Punkt erreicht. ENDE
14 **Return**$[x]$
15 **Return** $\left(\overrightarrow{[x]}, [[G_C]], \overrightarrow{[s]}, \overrightarrow{[S]}, [F]\right)$

Abbildung 5.9: Implementierung von Schritt 0

kann es jedoch aussehen, wenn im Rahmen der Sequentiellen Quadratischen Programmierung (Abschnitt 6.1) mehrere quadratische Optimierungsprobleme gelöst werden müssen.

5.7.3 Schritt 0

$[x]$ (Zeile 1 in Abb. 5.9) wird mit Hilfe der Cholesky-Zerlegung (Abb. 4.12) bestimmt, die sowieso für die (implizite) Berechnung von $[[N^*]]$ und $[[H]]$ benötigt wird. Zusätzlich bestimmt wird noch der Wert der Zielfunktion (Schritte 2-4), der Vektor $\overrightarrow{[s]} = \overrightarrow{[Ax - b]}$ (Schritte 5-7) sowie der Vektor $\overrightarrow{[S]}$, der die verletzten Nebenbedingungen indiziert (Schritte 8-9). Für den Fall, dass das globale Minimum zulässig ist, wird dies daran erkannt, dass keine Nebenbedingungen verletzt sind (Schritte 10-11). In diesem Fall wird der Algorithmus bereits hier abgebrochen (Schritte 12-14). Zurückgegeben wird der Punkt $\overrightarrow{[x]}$, die Vektoren $\overrightarrow{[s]}$ und $\overrightarrow{[S]}$ und zusätzlich die zerlegte Matrix $[[G_C]]$.

Protokoll 5.10: $\left(\overrightarrow{[n_p]}, \overrightarrow{[n^I]}\right) \leftarrow$ **Schritt1** $\left([[A]], \overrightarrow{[b]}, \overrightarrow{[x]}, \overrightarrow{[s]}, \overrightarrow{[S]}, m, n\right)$

1 $\overrightarrow{[n^I]} \leftarrow$ NBAuswahl //Protokoll 5.11 oder Protokoll 5.12

2 $\overrightarrow{[n_p]} \leftarrow$ SecReadRow $\left([[A]], \overrightarrow{[n^I]}\right)$ //n Mul $\mathbb{F}_q$, 1 Rnd (vgl. Abschnitt 3.1.2)

3 **Return** $\left(\overrightarrow{[n_p]}, \overrightarrow{[n^I]}\right)$

Abbildung 5.10: Implementierung von Schritt 1

5.7.4 Schritt 1

In Schritt 1 wird eine der verletzten Nebenbedingungen ausgewählt (vgl. Abschnitt 5.7.4.1). Die diesbezügliche Routine NBAuswahl erhält die Matrix $[[A]]$, den Vektor $\overrightarrow{[b]}$, den aktuellen Iterationspunkt $[x]$, die Indizes der verletzen Nebenbedingungen $\overrightarrow{[S]}$ und ggf. die Cholesky-Zerlegung $[[G_C]]$ der Matrix $[[G]]$. Sie liefert einen binären Vektor $\overrightarrow{[n^I]}$ zurück, der die Zeile der ausgewählten Nebenbedingung binär kodiert. Mit dessen Hilfe wird dann der konkrete Vektor $\overrightarrow{[n_p]}$ aus der Matrix $[[A]]$ ausgelesen. Zurückgegeben werden $\overrightarrow{[n_p]}$ und $\overrightarrow{[n^I]}$.

5.7.4.1 Auswahl der verletzten Nebenbedingung

Für das Funktionieren des Algorithmus ist es irrelevant, *welche* verletzte Nebenbedingung in Schritt 1 ausgewählt wird. Es besteht jedoch die Möglichkeit, die Anzahl der Iterationen durch geschickte Wahl der verletzten Nebenbedingungen zu minimieren. In [GI83] wird vorgeschlagen, diejenige Nebenbedingung zu wählen, die am stärksten verletzt ist, d.h. diejenige, bei der der Betrag des Residuums

$$a_i^t x - b_i \tag{5.87}$$

am größten ist (*Residuumsstrategie*). Begründet wird dies damit, dass auf diese Weise durch numerische Ungenauigkeiten bedingte Zyklen verhindert werden können. Korn ([Kor89]) weist jedoch darauf hin, dass dies nicht unter allen Umständen richtig ist und schlägt anstatt dessen vor, die Nebenbedingung zu wählen, die den größten Abstand bzgl. der von G induzierten Norm zum aktuellen Iterationspunkt besitzt (*G-Norm-Strategie*). Dabei definiert er den Abstand eines Punktes $\tilde{x}$ von einer Nebenbedingung $a_i^t x \geq b_i$ wie folgt:

$$d_G(\tilde{x}, a_i^t x \geq b_i) = \begin{cases} \min\limits_{a_i^t x = b_i} \|x - \tilde{x}\|_G, \text{ falls } a_i^t \tilde{x} < b_i \\ 0, \text{ falls } a_i^t \tilde{x} \geq b_i \end{cases} \tag{5.88}$$

Im ersten Fall lässt sich dies vereinfachen ([Kor89]) zu

$$\min_{a_i^t x = b_i} \|x - \tilde{x}\|_G = \frac{|b_i - a_i^t \tilde{x}|}{\|a_i\|_{G^{-1}}} \tag{5.89}$$

Diese Definition wird in der hier vorgestellten Implementierung (Abb. 5.11) der G-Norm-Strategie verwendet. Dabei ist (wie üblich)

$$\|a_i\|_{G^{-1}} := \sqrt{a_i^t G^{-1} a_i}. \tag{5.90}$$

Alternativ gibt es noch die Möglichkeit, die Nebenbedingung zu wählen, die den größten euklidischen Abstand (ersetze dazu G in (5.88) durch die Einheitsmatrix) zu x^* besitzt bzw. einfach die lexikalisch erste. Welche Strategie am günstigsten ist, lässt sich nicht allgemein angeben. Die Strategie die Nebenbedingung, die den größten Abstand zum aktuellen Iterationspunkt bzgl. der Norm von G besitzt, zahlt sich nur bei einer dramatischen Reduktion der Zahl der Iterationen aus, da für die Berechnung des Nenners für jede Nebenbedingung c das Gleichungssystem $Gx = c$ gelöst werden muss. In weiteren Iterationen können die so gewonnenen Werte x zwar wiederverwertet werden, aber bei einer größeren Anzahl an verletzten Nebenbedingungen können die Kosten bereits prohibitiv sein. Zwar ist eine Cholesky-Zerlegung von G bekannt, doch sind die verbleibenden Operationen immer noch sehr teuer (vgl. Tabelle 4.2). Die G-Norm-Strategie sowie die Euklidische-Norm-Strategie scheinen am ehesten geeignet, Zyklen zu vermeiden ([Kor89]).

Protokoll 5.11 implementiert die G-Norm-Strategie. Für den Zähler und Nenner werden nicht Zähler und Nenner wie in (5.89) berechnet, sondern deren Quadrat. Der Index des Maximums wird dadurch nicht beeinträchtigt, es kann aber auf die Berechnung der Wurzel im Nenner und die des Betrags im Zähler verzichtet werden. Um den Zähler einer jeden Komponente zu erhalten, wird einfach der jeweilige übergebene Wert $[s(i)]$ (Schritt 2-3) quadriert. Der Nenner wird durch Vorwärts- und Rückwärtssubstitution mit der Zerlegung von G (Schritt 5-9) berechnet, die bestehenden Cholesky-Zerlegung $[[G_C]]$ der Matrix G kann dazu verwendet werden. Beachte, dass der Nenner nicht vom Iterationspunkt abhängt, also nur einmal berechnet werden muss. Bevor der Index des Minimums $\frac{\overrightarrow{[Z]}}{\overrightarrow{[N]}}$ bestimmt wird (Schritt 11), werden Komponenten, die zu bereits erfüllten Nebenbedingungen korrespondieren, durch $-\infty$ ($\triangleq -2^{k-1} + 1$) im Zähler und 1 im Nenner (Schritte 9-10) ersetzt. Bei der

Protokoll 5.11: $\left(\overrightarrow{[n_I]}, \overrightarrow{[N]}\right) \leftarrow$ **NB_G_Norm** $\left([[G_C]], \overrightarrow{[b]}, \overrightarrow{[x]}, \overrightarrow{[S]}, \overrightarrow{[s]}, \overrightarrow{[N]}\right)$

For $(i = 1, \ldots, m)$ **do parallel**
  $[Z(i)] \leftarrow [s(i)] \cdot [s(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
  $[Z(i)] \leftarrow \mathsf{TruncPr}([Z(i)], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul, 2 Rnd $\mathbb{F}_{q_1}$
  If (1. Iteration)
    $\overrightarrow{[HV]} \leftarrow \mathsf{VWSubs}(\overrightarrow{[A(i,*)]}, [[G_C]], n)$ //Protokoll 4.4
    $\overrightarrow{[HV]} \leftarrow \mathsf{RWSubs}(\overrightarrow{[HV]}, [[G_C]], n)$ //Protokoll 4.5
    $[N(i)] \leftarrow \mathsf{Inner}(\overrightarrow{[A(i,*)]}, \overrightarrow{[HV]}, n)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
    $[N(i)] \leftarrow \mathsf{TruncPr}([N(i)], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
  $[Z(i)] \leftarrow [S(i)] \cdot [Z(i)] + (1 - [S(i)]) \cdot (-\infty)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
  $[KN(i)] \leftarrow [S(i)] \cdot [N(i)] + (1 - [S(i)]) \cdot (-2^f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
$\overrightarrow{[n_I]} \leftarrow \mathsf{MaxBruch}(\overrightarrow{[Z]}, \overrightarrow{[KN]})$ //Protokoll 3.5
Return $\left(\overrightarrow{[n_I]}, \overrightarrow{[N]}\right)$

Abbildung 5.11: G-Norm-Strategie

Protokoll 5.12: $\overrightarrow{[n_I]} \leftarrow$ **NB_Residuum** $\left(\overrightarrow{[s]}, m, k\right)$

$\overrightarrow{[n_I]} \leftarrow \mathsf{MinVektor}\left(\overrightarrow{[s]}, m, k\right)$ //Protokoll 3.3
Return $\overrightarrow{[n_I]}$

Abbildung 5.12: Auswahl der verletzten Nebenbedingung

Residuumsstrategie (Protokoll 5.12) braucht die aktive Menge nicht berücksichtigt werden, da der Wert $a_i^t x - b_i$ für $i \in W_k$ ohnehin größer als Null ist.

5.7.5 Schritt 2

In der Implementierung von Schritt 2 (Protokoll 5.13) wird zwischen der 1. Iteration und allen weiteren (vgl. Protokoll 5.8) unterschieden. Zu Beginn ist H gegeben als G^{-1} (vgl. Abschnitt 5.6.6) und die aktive Menge ist leer; aus diesem Grund kann die Berechnung von z vereinfacht werden (5.67) bzw. die von r und t_1 auch entfallen (Nach Konstruktion werden diese nur benötigt, wenn die aktive Menge nicht leer ist!). Daraus ergeben sich wieder Vereinfachungen in Schritt 2c bei der Berechnung von u^+ und u (Satz 4) und der Neuberechnung der aktiven Menge. Aus Gründen der Einfachheit wird bei der Darstellung davon ausgegangen, dass die Matrix $[[A]]$, der Vektor $\overrightarrow{[b]}$ und die Cholesky-Zerlegung $[[G_C]]$ der Matrix $[[G]]$ sowie die Dimension m und die Anzahl

der Nebenbedingungen als globale Variablen bekannt sind, beim Aufruf von Schritt2 und den Unterfunktionen also nicht übergeben werden müssen.

In der 1. Iteration ist H gegeben durch G^{-1}. Der Schrittvektor z kann also mit Hilfe von Vorwärts- und Rückwärtssubstitution von n^+ und der Cholesky-Zerlegung der Matrix G bestimmt werden (Schritte 1-3). Ansonsten wird die Funktion zrNeu (Schritte 4-5) verwendet. Im Anschluss daran wird die Schrittweite t (Schritt 6) berechnet und es wird überprüft, ob diese gleich ∞ ist (Schritt 7). Ist dies der Fall, gibt es keinen zulässigen Bereich und das Programm wird abgebrochen (Schritt 8). Aus t und anderen Rückgabewerten der Funktion Schrittweite werden Sharings abgeleitet, die kodieren, ob $t = t_1$, $t = t_2$ und $t_2 = \infty$ (Schritte 9 und 10). Beachte, dass der Wert M, zurückgegeben von Schrittweite genau dann $[1]$ ist, wenn $t = t_1$. Dies lässt jedoch - im Prinzip - nicht darauf schließen, dass in diesem Fall $t \neq t_2$! Die vorliegende Implementierung geht trotzdem davon aus (Zu den Gründen siehe Abschnitt 5.7.9).

Mit Hilfe von t wird der neue Iterationspunkt berechnet (Schritt 11) und auf primale Zulässigkeit überprüft (Schritte 20-26). Zudem können der Wert der Zielfunktion f (Schritt 12) und der Lagrange-Multiplikatoren u (Schritt 13) aktualisiert werden. Die Information, ob $t = t_1$ oder $t = t_2$ wird dazu verwendet die aktive Menge W (Schritte 15,17-18) und ihre Größe q (Schritte 15,19) zu aktualisieren. Da in der 1. Iteration $t = t_2$ gelten muss, kann die Berechnung in diesem Fall vereinfacht werden. Ergibt die Prüfung auf primale Zulässigkeit ein Optimum, so kann der Algorithmus abgebrochen werden und der optimale Punkt x^*, der Wert der Zielfunktion f und die Lagrange-Multiplikatoren u zurückgegeben werden (Schritt 27). Als Vorbereitung der Aktualisierung von J (Schritt 29) muss nun nur noch die neue aktive Menge kompaktifiziert werden (Schritt 28).

Beachte, dass viele der Operationen parallel ausgeführt werden können, so z.B. jeweils die Schritte 9-10 und 11-19. Schritt 28 kann parallel zu den Schritten 19-26 ausgeführt werden.

5.7.6 Berechnung von z und r

Der Algorithmus zur Berechnung von z und r stellt sich wie in Protokoll 5.14 beschrieben dar und implementiert die Formeln (5.81,5.82 und 5.83). Die Matrizen $[[J]]$ und $[[B]]$ seien als gegeben vorausgesetzt. Beachte, dass in der Matrix $[[B]]$ die Matrix $[[R]]$ gespeichert ist (vgl. Protokoll 5.19). Mehr zur Berechnung von z und r in Abschnitt 5.7.12.

Protokoll 5.13: $\left(\overrightarrow{[x]}, \overrightarrow{[u^+]}, \overrightarrow{[u]}, \overrightarrow{[W]}, \overrightarrow{[S]}, [f]\right) \leftarrow$ **Schritt2** $\left(\overrightarrow{[x]}, \overrightarrow{[u]}, \overrightarrow{[u^+]}, \overrightarrow{[n^+]}, \overrightarrow{[n^I]}, \overrightarrow{[W]}, [[P]], [f]\right)$

If (1.Iteration)
  $\overrightarrow{[y]} \leftarrow \mathsf{VWSubs}\left([[G_C]], \overrightarrow{[n_p]}, n\right)$ //Protokoll 4.4
  $\overrightarrow{[z]} \leftarrow \mathsf{RWSubs}\left([[G_C]], \overrightarrow{[y]}, n\right)$ //Protokoll 4.5
Else
  $\left(\overrightarrow{[z]}, \overrightarrow{[r]}, \overrightarrow{[r_c]}\right) \leftarrow \mathsf{zrNeu}\left([[J]], [[B]], \overrightarrow{[W^C]}, [[P]], \overrightarrow{[n^+]}\right)$ //Protokoll 5.14
$\left([t], [M], [zn], [N_z], \left[t^I_{1,g}\right]\right) \leftarrow \mathsf{Schrittweite}\left([[P]], \overrightarrow{[W]}, \overrightarrow{[z]}, \overrightarrow{[r^C]}, \overrightarrow{[n^+]}, It.\right)$//P. 5.15
$C \leftarrow \mathsf{EQPub}([t], \infty, k)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
If (C=1): **Return** $\left(\overrightarrow{[x]}, \overrightarrow{[u]}, [f]\right)$ //Kein zulässiger Bereich. ENDE.
$\left[t_{2 \stackrel{?}{=} \infty}\right] \leftarrow \mathsf{EQ}([t_2], \infty, k)$ //vgl. Tabelle 2.1
$\left[t_{\stackrel{?}{=} t_1}\right] \leftarrow [M]$, $\left[t_{\stackrel{?}{=} t_2}\right] \leftarrow 1 - [M]$
$\overrightarrow{[x]} \leftarrow \mathsf{xNeu}\left(\overrightarrow{[z]}, [t], \left[t_2 \stackrel{?}{=} \infty\right], m, n\right)$ //Protokoll 5.16
$[f] \leftarrow \mathsf{fNeu}\left(\overrightarrow{[n^I]}, \overrightarrow{[u^+]}, [t], [zn], \left[t_{2_{\stackrel{?}{=}\infty}}\right], n\right)$ //Protokoll 5.18
$\left(\overrightarrow{[u]}, \overrightarrow{[u^+]}\right) \leftarrow \mathsf{uNeu}\left(\overrightarrow{[r]}, \overrightarrow{[u]}, \overrightarrow{[u^+]}, [n^I], [t], n\right)$ //Protokoll 5.17
If (1.Iteration)
  $\overrightarrow{[W]} \leftarrow \overrightarrow{[n^I]}$, $[q] \leftarrow 1$
Else
  $\overrightarrow{[W]} \leftarrow \overrightarrow{[W]} \cdot \left(1 - \left[t_{\stackrel{?}{=} t_2}\right]\right) + \left[t_{\stackrel{?}{=} t_2}\right] \cdot \left(\overrightarrow{[n^I]} + \overrightarrow{[W]}\right)$ //$2n$ Mul $\mathbb{F}_q$
  $\overrightarrow{[W]} \leftarrow \overrightarrow{[W]} \cdot \left(1 - \left[t_{\stackrel{?}{=} t_1}\right]\right) + \left[t_{\stackrel{?}{=} t_1}\right] \left(\overrightarrow{[W]} - \overrightarrow{[t^I_1]}\right)$ //$2n$ Mul F_q
  $[q] \leftarrow ([q] + 1) \cdot \left[t_{\stackrel{?}{=} t_2}\right] + ([q] - 1) \cdot \left(1 - \left[t_{\stackrel{?}{=} t_2}\right]\right)$ //2 Mul $\mathbb{F}_q$
$\overrightarrow{[s]} \leftarrow [[A]] \cdot \overrightarrow{[x]}$ //m Mul, 1 Rnd $\mathbb{F}_q$
$\overrightarrow{[s]} \leftarrow \mathsf{TruncPr}(\overrightarrow{[s]}, k, f)$ //m Mul, 1 Rnd $\mathbb{F}_q$, $f + 1$ Mul $\mathbb{F}_{q_1}$
$\overrightarrow{[s]} \leftarrow \overrightarrow{[s]} - \overrightarrow{[b]}$
For $(i = 1, \ldots, m)$ **do parallel**
  $[S(i)] \leftarrow \mathsf{LTZappr}([s(i)], \epsilon, k)$ //vgl. Tabelle 2.1
$[V] \leftarrow \sum_{i=1}^{m} [S(i)]$
$v \leftarrow \mathsf{EQZPub}([V], k)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
If $(v = 1)$: **Return** $\left(\overrightarrow{[x]}, \overrightarrow{[u]}, [f]\right)$ // Optimaler Punkt erreicht
$\left(\overrightarrow{[W^C]}, [[P]]\right) \leftarrow \mathsf{KompkatifiziereBinär}\left(\overrightarrow{[W]}, \overrightarrow{[W]}, m\right)$ //Protokoll 3.8
$([[J]], [[B]]) \leftarrow \mathsf{J}\left([[B]], [[L]], [[P]], \overrightarrow{[W^C]}, \overrightarrow{[n_p]}, \overrightarrow{[Num]}, \left[t_{\stackrel{?}{=} t_1}\right], \left[t_{\stackrel{?}{=} t_2}\right], [NNB], It., m, n\right)$//P.5.19
Return $\left(\overrightarrow{[x]}, \overrightarrow{[u^+]}, \overrightarrow{[u]}, \overrightarrow{[W^+]}, [[P]], \overrightarrow{[S]}, [f]\right)$

Abbildung 5.13: Implementierung von Schritt 2

Protokoll 5.14: $\left(\overrightarrow{[z]}, \overrightarrow{[r]}\right) \leftarrow$ **zrNeu** $\left([[J]], [[B]], \overrightarrow{[W_c]}, [[P]], \overrightarrow{[n^+]}\right)$

1 $\overrightarrow{[d]} \leftarrow [[J^t]] \cdot \overrightarrow{[n^+]}$ //n Mul $\mathbb{F}_q$, 1 Rnd
2 $\overrightarrow{[d]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[d]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $n(f+1)$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3 $\overrightarrow{[d_1]} \leftarrow \mathsf{MultipliziereZeilenweise}\left(\overrightarrow{[d]}, \overrightarrow{[W_c]}\right)$ //n Mul $\mathbb{F}_q$, 1 Rnd
4 $[[J_1]] \leftarrow \mathsf{MultipliziereSpaltenweise}\left([[J]], \overrightarrow{[W_c]}\right)$ //n^2 Mul $\mathbb{F}_q$
5 $[[J_2]] \leftarrow [[J]] - [[J_1]]$
6 $\overrightarrow{[z]} \leftarrow [[J_2]] \cdot \overrightarrow{[d]}$ //n Mul $\mathbb{F}_q$, 1 Rnd
7 $\overrightarrow{[z]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[z]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $n(f+1)$ Mul $\mathbb{F}_{q_1}$
8 $\overrightarrow{[r_c]} \leftarrow \mathsf{RWSubs}\left([[B]], \overrightarrow{[d_1]}\right)$ //Protokoll 4.5
9 $\overrightarrow{[r]} \leftarrow \begin{pmatrix} [\overrightarrow{r'}] \\ 0 \end{pmatrix}$
10 $\overrightarrow{[r]} \leftarrow [[P^t]] \cdot \overrightarrow{[r]}$ //m Mul, 1 Rnd $\mathbb{F}_q$
11 **Return** $\left(\overrightarrow{[z]}, \overrightarrow{[r]}, \overrightarrow{[r_c]}\right)$

Abbildung 5.14: Berechnung von z und r

5.7.7 Berechnung der Schrittweite

Übergeben werden die Variablen $\left([[P]], \overrightarrow{[W]}, \overrightarrow{[z]}, \overrightarrow{[r^C]}, \overrightarrow{[n^+]}, Iteration\right)$. In $\overrightarrow{[r_C]}$ werden die - maximal n - zu den aktiven Nebenbedingungen korrespondierenden Werte von $\overrightarrow{r'}$ übergeben. Sie werden von der Funktion zrNeu bereits in einem Vektor der Länge n zusammengeschoben (ggf. aufgefüllt mit Null-Elementen am Ende). Zu Beginn wird überprüft, ob $z \stackrel{?}{=} 0$ (Schritt 1) und das Skalarprodukt $\langle z, n^+ \rangle$ berechnet (Schritte 2-3), das für die Berechnung von t_2 und die Aktualisierung von t_2 benötigt wird. Für die Berechnung von $[t_1]$ wird wie in [GI83] zwischen der 1. Iteration - in der $[q]$ zwangsweise 0 ist - und allen weiteren unterschieden. Ist die Iterationszahl größer als 0, so wird $[q]$ (die Größe der aktiven Menge) auf Gleichheit mit Null überprüft (Schritt 7) und ob mindestens einer der Einträge von $[\overrightarrow{r'}]$ größer als 0 ist (Schritte 8-10). Das Ergebnis der Komponentenüberprüfung wird in den Variablen $\left[r_{\stackrel{?}{>0_V}}(i)\right]$ gespeichert (Schritt 9) und das Ergebnis der Überprüfung, ob mindestens *eine* Komponente größer als null ist, im Vektor $\left[r_{\stackrel{?}{>0}}\right]$ (Schritt 10). Die Entscheidung, ob die Bedingungen für $t_1 < \infty$ erfüllt sind ($q \neq 0$ und mindestens eine Komponente von r größer 0), wird in $[S_{t_1}]$ gespeichert (Schritt 11).

Protokoll 5.15: $\left([t],[t_2],[M],[zn],[N_z],\left[t^I_{1,g}\right]\right) \leftarrow$ **Schrittweite** $\left([[P]],\overrightarrow{[W]},\overrightarrow{[z]},\overrightarrow{[r^C]},\overrightarrow{[n^+]},\overrightarrow{[s]},\overrightarrow{[n^I]},It.\right)$

1 $[N_z] \leftarrow \mathsf{EQZappr}(\overrightarrow{[z]},n,k)$ //Protokoll 3.2
2 $[zn] \leftarrow \mathsf{Inner}\left(\overrightarrow{[z]},\overrightarrow{[n^+]}\right)$ //1 Mul $\mathbb{F}_q$
3 $[zn] \leftarrow \mathsf{TruncPr}([zn],k,f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
4 **If** (1. Iteration)
5 $\quad [t_1] \leftarrow \infty$
6 **Else**
7 $\quad \left[q_{\stackrel{?}{=}0}\right] \leftarrow \mathsf{EQZ}([q],k)$ //vgl. Tabelle 2.1
8 $\quad$ **For** $(i=1,\ldots,n)$ **do parallel**
9 $\quad\quad \left[r_{\stackrel{?}{>}0_V}(i)\right] \leftarrow \mathsf{GTZappr}([r(i)],\varepsilon,k)$ //vgl. Tabelle 2.1
10 $\quad \left[r_{\stackrel{?}{>}0}\right] \leftarrow \mathsf{OR}^n_{i=1}\left[r_{\stackrel{?}{>}0'_V}(i)\right]$ //n Mul, n Rnd $\mathbb{F}_q$
11 $\quad [S_{t_1}] \leftarrow \left[r_{\stackrel{?}{>}0}\right] \mathsf{AND} \left(1-\left[q_{\stackrel{?}{=}0}\right]\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
12 $\quad \begin{pmatrix}\overrightarrow{[t_{1_V},k]}\\ 0\end{pmatrix} \leftarrow [[P]]\cdot\overrightarrow{[u^+]}$ //m Mul, 1 Rnd $\mathbb{F}_q$
13 $\quad$ **For** $(i=1,\ldots,n)$ **do parallel**
14 $\quad\quad [K(i)] \leftarrow [W(i)]$ AND $\left[r_{\stackrel{?}{>}0_V}(i)\right]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
15 $\quad\quad \left[t_{1_{V,k}}(i)\right] \leftarrow \left[t_{1_{V,k}}(i)\right]\cdot[K(i)]+(1-[K(i)])\cdot\infty$ //1 Mul $\mathbb{F}_q$
16 $\quad\quad [\tilde{r}(i)] \leftarrow [r(i)]\cdot[K(i)]+(1-[K(i)])\cdot 2^f$ //1 Mul $\mathbb{F}_q$
17 $\quad \overrightarrow{[t^I_1]} \leftarrow \mathsf{MinBruch}\left(\overrightarrow{[t_{1_V}]},\overrightarrow{[\tilde{r}]},n\right)$ //Protokoll 3.4
18 $\quad [t_1] \leftarrow \mathsf{SecRead}\left(\overrightarrow{[t^I_i]},\left[\overrightarrow{t_{1_V}}\right],n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
19 $\quad [r_N] \leftarrow \mathsf{SecRead}\left(\overrightarrow{[t^I_i]},\left[\overrightarrow{\tilde{r}}\right],n\right)$ //1 Mul $\mathbb{F}_q$
20 $\quad [t_1] \leftarrow \mathsf{FPDiv}([t_1],[r_N],k,f)$ //vgl. Tabelle 2.1
21 $\quad [t_1] \leftarrow [S_{t_1}]\cdot[t_1]+(1-[S_{t_1}])\cdot\infty$ //m Mul, 1 Rnd $\mathbb{F}_q$
22 $\quad \left(\overrightarrow{[t^I_{1,g}]} = \begin{pmatrix}\overrightarrow{[t^I_1]}\\ 0_{n+1}\\ \vdots\\ 0_m\end{pmatrix}\right) \hookleftarrow \overrightarrow{[t^I_1]}$
23 $\quad \overrightarrow{[t^I_{1,g}]} \leftarrow [[P^t]]\cdot\overrightarrow{[t^I_{1,g}]}$ //m Mul $\mathbb{F}_q$
24 $[s_p] \leftarrow \mathsf{Inner}\left(\overrightarrow{[s]},\overrightarrow{[n^I]},m\right)$ //1 Mul $\mathbb{F}_q$
25 $[s_p] \leftarrow \text{-}\mathsf{FPDiv}([s_p],[zn],k,f)$ //vgl. Tabelle 2.1
26 $[t_2] \leftarrow [N_z]\cdot\infty+(1-[N_z])\cdot[s_p]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
27 $([t],[M]) \leftarrow \mathsf{Min}([t_1],[t_2],k,f)$ //Protokoll 2.1
28 **Return** $\left([t],[M],[zn],[N_z],\left[t^I_{1,g}\right],\overrightarrow{[r_g]}\right)$

Abbildung 5.15: Berechnung der Schrittweite

Protokoll 5.16: $\overrightarrow{[x]} \leftarrow \mathsf{xNeu}\left(\overrightarrow{[z]}, [t], \left[t_{2_{\stackrel{?}{=}\infty}}\right], m, n\right)$

1 $\overrightarrow{[tz]} \leftarrow [t] \cdot \overrightarrow{[z]}$ //n Mul, 1 Rnd $\mathbb{F}_q$

2 $\overrightarrow{[tz]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[tz]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$

3 $\overrightarrow{[x]} \leftarrow \left(\overrightarrow{[x]} + \overrightarrow{[tz]}\right) \cdot \left(1 - \left[t_{2_{\stackrel{?}{=}\infty}}\right]\right) + \left[t_{2_{\stackrel{?}{=}\infty}}\right] \cdot \overrightarrow{[x]}$ //$2n$ Mul, 1 Rnd $\mathbb{F}_q$

4 **Return** $\overrightarrow{[x]}$

Abbildung 5.16: Neuberechnung von x

In Schritt 12 werden die Nicht-Null-Komponenten des Vektors u^+ (die zur aktiven Menge korrespondieren), in einem Vektor $\overrightarrow{[t_{1,V}]}$ der Länge n gespeichert. Anschließend (Schritte 13-16) werden die Komponenten von $\overrightarrow{[t_{1,V}]}$ bzw. $\overrightarrow{[r_{\stackrel{?}{=}0_V}]}$, die nicht zur aktiven Menge korrespondieren und für die die entsprechende Komponente von r nicht größer 0 ist, durch ∞ bzw. 2^{-f} ersetzt. $[t_1]$ wird im Anschluss mit Hilfe der Funktion MinBruch in den Schritten 17-20 berechnet und, falls die Bedingung für t_1 (Schritt 11) nicht erfüllt ist, durch ∞ ersetzt (Schritt 21). Die Position von t_1, die für die Aktualisierung der aktiven Menge benötigt wird, wird in den Schritten 22 - 23 bestimmt. Dazu wird die Permutation, mit der der Indexvektor der aktiven Menge kompaktifiziert wurde (und die ebenso auf r und u^+ angewandt wurde), rückgängig gemacht (Schritt 23).

In den Schritten 24-26 wird schließlich $[t_2]$ berechnet - einschließlich der Überprüfung, ob $[t_2]$ wegen Gleichheit mit 0 von $[z]$ auf ∞ gesetzt wird. Die Wahl zwischen $[t_1]$ und $[t_2]$ geschieht im letzten Schritt 28. Zurückgegeben werden t, die Variable $[M]$, in der kodiert ist, ob $t = t_2$ oder $t = t_1$, N_z, das implizit angibt, ob $t_2 = \infty$, $\left[z^t n^+\right]$ sowie der Index $\overrightarrow{\left[t^I_{1,g}\right]}$ von $[t_1]$ in binärer Form. Die Schritte 1-3 und 7 sowie alle GTZappr-Berechnungen (Schritt 9) können parallelisiert werden. Gleiches gilt für die Schritte 11-12 sowie 13-16. Die Berechnung von $[t_2]$ (Schritte 25-27) kann parallel zu allen vorhergehenden erfolgen.

5.7.8 Neuberechnung von x

Für die Neuberechnung von x (Protokoll 5.16) muss unterschieden werden, ob $[t_2] = \infty$; in diesem Fall wird nur ein Schritt in den dualen Variablen ausgeführt. Die Entscheidung darüber ist in der Variablen $\left[t_{2_{\stackrel{?}{=}\infty}}\right]$ gespeichert.

5.7.9 Der Fall $t_1 = t_2$

In den meisten nicht-sicheren Implementierungen des Algorithmus von Goldfarb und Idnani ([GI83],[Kor89],[Pow85]) wird dem Fall $t_1 = t_2$ keine Beachtung geschenkt. Dies ist wenig verwunderlich, denn der Fall ist sehr unwahrscheinlich! Tritt er dennoch ein, bedeutet dies, dass bei der Schrittlänge t eine neue Nebenbedingung aktiv wird und *gleichzeitig* der Lagrange-Multiplikator einer bestehenden zu Null wird. Es muss also eine Nebenbedingung zur aktiven Menge hinzugefügt werden und eine andere aus ihr entfernt werden. In diesem Fall versagt die Strategie, nur eine Spalte von B zu aktualisieren (Abschnitt 5.7.12). Möchte man dies bei der hier vorliegenden sicheren Implementierung ausschließen, muss die Aktualisierung von J in jedem Schritt durch eine vollständige Neuberechnung ersetzt werden bzw. hintereinander stets die Addition einer Spalte (ggf. der Nullspalte) und anschließend das Entfernen (ggf. der Nullspalte) einer Spalte durchgeführt werden. Die Schritte 10 und 11 von Protokoll 5.13 sowie die Rückgabe der Funktion Schrittweite müssen dann ebenfalls geändert werden. Da in der Literatur aber dem Fall $t_1 = t_2$ keine Beachtung geschenkt und auch ein praktisches Auftreten nicht erwähnt wird, wird er hier ebenfalls ignoriert.

Auch denkbar ist, dass *gleichzeitig* zwei oder mehr Nebenbedingungen auf einmal aktiv bzw. inaktiv werden. Dies ist jedoch noch unwahrscheinlicher und wird weder von den nicht-sicheren Implementierungen noch hier berücksichtigt.

5.7.10 Neuberechnung von $[u]$ und $[u^+]$

Protokoll 5.17 implementiert die Aktualisierung von $[u]$ und $[u^+]$ mittels der Formel $u^+ \leftarrow u^+ + r \begin{pmatrix} -r \\ 1 \end{pmatrix}$ (Satz 4). In Schritt 5 wird $\overrightarrow{[u]}$ aktualisiert, wenn $t = t_2$, d.h. wenn ein voller und nicht nur ein partieller Schritt durchgeführt wurde.

5.7.11 Aktualisierung des Werts der Zielfunktion

Die Neuberechnung von $[f]$ (Protokoll 5.18) implementiert die Formel

$$f \leftarrow f + tz^t n^+ \cdot \left(\frac{t}{2} + u^+_{q+1} \right). \tag{5.91}$$

5.7.12 Berechnung von J

In diesem Abschnitt soll die Idee der sicheren Implementierung der Neuberechnung von $[[J]]$ (vgl. Abschnitt 5.6.6) beschrieben werden. Insbesondere soll

Protokoll 5.17: $\left(\overrightarrow{[u]}, \overrightarrow{[u^+]}\right) \leftarrow \mathsf{uNeu}\left(\overrightarrow{[r]}, \overrightarrow{[u]}, \overrightarrow{[u^+]}, \overrightarrow{[n^I]}, [t], \left[t_{\stackrel{?}{=}t_2}\right], n\right)$

1. $\overrightarrow{[u^+]} \leftarrow \overrightarrow{[u^+]} + \overrightarrow{[n^I]}$
2. $\overrightarrow{[r]} \leftarrow [t] \cdot \overrightarrow{[r]}$ //m Mul, 1 Rnd $\mathbb{F}_q$
3. $\overrightarrow{[r]} \leftarrow \mathsf{TruncPr}\left(\overrightarrow{[r]}, k, f\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
4. $\overrightarrow{[u^+]} \leftarrow \overrightarrow{[u^+]} - \overrightarrow{[r]}$
5. $\overrightarrow{[u]} \leftarrow \overrightarrow{[u^+]} \cdot \left[t_{\stackrel{?}{=}t_2}\right] + \overrightarrow{[u]} \cdot \left(1 - \left[t_{\stackrel{?}{=}t_2}\right]\right)$ //$2m$ Mul, 1 Rnd $\mathbb{F}_q$
6. **Return** $\left(\overrightarrow{[u]}, \overrightarrow{[u^+]}\right)$

Abbildung 5.17: Neuberechnung von u

Protokoll 5.18: $[f] \leftarrow \mathsf{fNeu}\left(\overrightarrow{[n^I]}, \overrightarrow{[u^+]}, [t], [zn], \left[t_{2_{\stackrel{?}{=}\infty}}\right], n\right)$

1. $[zn] \leftarrow [zn] \cdot [t]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
2. $[zn] \leftarrow \mathsf{TruncPr}([zn], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
3. $\left[u_{q+1}\right] \leftarrow \mathsf{SecRead}\left(\overrightarrow{[u^+]}, \overrightarrow{[n^I]}, n\right)$ //n Mul, 1 Rnd $\mathbb{F}_q$
4. $[H] \leftarrow 2\left[u_{q+1}\right] + [t]$
5. $[H] \leftarrow \mathsf{DivKonst}([H], 2, k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
6. $[zn] \leftarrow [zn] \cdot [H]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
7. $[zn] \leftarrow \mathsf{TruncPr}([zn], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
8. $[F] \leftarrow ([F] + [zn]) \cdot \left(1 - \left[t_{2_{\stackrel{?}{=}\infty}}\right]\right) + \left[t_{2_{\stackrel{?}{=}\infty}}\right] \cdot [F]$ //2 Mul, 1 Rnd $\mathbb{F}_q$
9. **Return** $[f]$

Abbildung 5.18: Neuberechnung von f

gezeigt werden, wie diese durchgeführt werden kann, *ohne* die Größe der aktiven Menge über die Größe der beteiligten Matrizen preiszugeben.

Die Matrix N besitzt wegen LICQ (Definition 8) nicht mehr als n Spalten, die Normalen der der aktiven Menge zuzuordnenden Nebenbedingungen. N wird nun eingebettet in eine $n \times n$-Matrix $\tilde{N}$, d.d.

$$\tilde{N} = \begin{pmatrix} N & 0 \end{pmatrix}. \tag{5.92}$$

Dann gilt

$$\tilde{B} := L^{-1}\tilde{N} = \begin{pmatrix} L^{-1}N & 0 \end{pmatrix} = \begin{pmatrix} B & 0 \end{pmatrix}. \tag{5.93}$$

Verfahren	Multiplikationen	Runden	Körper
SchrittweiteGI	$19n+8\theta+23$	$2n+4\lceil\log_2 n\rceil+2\theta+22$	$\mathbb{F}_q$
	$\mathcal{O}(nk+\theta f)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(k\log_2 k+nk)$	$\lceil\log_2 k\rceil(n+5+\log_2 n)+n+3$	$\mathbb{F}_{2^8}$
zrNeu	$n^2+17n+4\theta n$	$2n(\theta+6)+2(k+1)$	$\mathbb{F}_q$
	$2\theta n(2f+1)+4n(2k+1)$	2	$\mathbb{F}_{q_1}$
	$3n(k-1)+1,5nk\lceil\log_2 k\rceil$	$4,5n\lceil\log_2 k\rceil+2n$	$\mathbb{F}_{2^8}$
xNeu	$4n$	3	$\mathbb{F}_q$
	$n(f+1)$	2	$\mathbb{F}_{q_1}$
uNeu	$4m$	3	$\mathbb{F}_q$
	$m(f+1)$	2	$\mathbb{F}_{q_1}$
fNeu	$n+7$	7	$\mathbb{F}_q$
	$3(f+1)$	2	$\mathbb{F}_{q_1}$
J	$\mathcal{O}(n^3+n^2\theta)$	$\mathcal{O}(n^2+\Xi n)$	$\mathbb{F}_q$
	$\mathcal{O}(n^3f+n^2\theta f+n^2k+\Xi kn)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(n^2k\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$
Schritt0	$\mathcal{O}(n^3+\Xi n)$	$\mathcal{O}(n^3+\Xi n)$	$\mathbb{F}_q$
	$\mathcal{O}(n^3f+\Xi kn)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(nk\log_2 k)$	$\mathcal{O}(n\log_2 k)$	$\mathbb{F}_{2^8}$
NB_G_Norm (It.>1)	$24n-3$	$4\log_2 m+5$	$\mathbb{F}_q$
	$2n(f+1)+12nk-4k$	$2\log_2 m+\log_2 k+2$	$\mathbb{F}_{q_1}$
	$8nk-6n-4k+3$	$\log_2 m\log_2 k+\log_2 m$	$\mathbb{F}_{2^8}$
NB_G_Norm (It.=1)	$\mathcal{O}(n^2\theta)$	$\mathcal{O}(k\log_2 n)$	$\mathbb{F}_q$
	$\mathcal{O}(n^2\theta f+n^2k)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(n^2k\log_2 k)$	$3(n+1)\lceil\log_2 k\rceil+2n$	$\mathbb{F}_{2^8}$
Schritt2	$\mathcal{O}(n^3+n^2\theta)$	$\mathcal{O}(n^2+\Xi n+k)$	$\mathbb{F}_q$
	$\mathcal{O}(n^3f+n^2\theta f+n^2k+\Xi kn)$	2	$\mathbb{F}_{q_1}$
	$\mathcal{O}(n^2k\log_2 k)$	$\mathcal{O}(n\log_2 k+k)$	$\mathbb{F}_{2^8}$

Tabelle 5.3: Komplexität der beim Algorithmus von Goldfarb und Idnani auftretenden Protokolle. θ bezeichne hier die Anzahl der Goldschmidt-Iterationen im Protokoll FPDiv und Ξ die Anzahl der Newton-Raphson-Iterationen im Protokoll DivNR.

Es folgt dann

$$\tilde{B} = \begin{pmatrix} B & 0 \end{pmatrix} = Q \cdot \begin{pmatrix} R & 0 \\ 0 & 0 \end{pmatrix}. \tag{5.94}$$

Somit ergibt sich für J und d keine Veränderung zur nicht-verteilten Implementierung, denn dort ist $J := L^{-1}Q$. Weiterhin kann der Schrittvektor z durch die

Formel

$$\begin{pmatrix} 0 & J_2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ d_2 \end{pmatrix} = J_2 \cdot d_2 = z \tag{5.95}$$

berechnet werden, also indem man in dem Ausdruck $J \cdot d$ $J_1 = 0$ und $d_1 = 0$ setzt. r erhält man als Lösung des Gleichungssystems $R \cdot r = d_1$ (Abschnitt 5.6.6), aber auch in Form der ersten q Einträge der Lösung des Gleichungssystems

$$\begin{pmatrix} R & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ 0 \end{pmatrix} = \begin{pmatrix} d_1 \\ 0 \end{pmatrix}. \tag{5.96}$$

Dieses lässt sich einfach durch Rückwärtssubstitution (vgl. Protokoll 4.5) lösen. Beachte dabei, dass $\mathsf{FPDiv}([0],[0],k)=[0]$.

Es ist also möglich, die Matrix J sowie die Vektoren z und r sicher zu berechnen, *ohne* die Größe der Matrizen J_1, J_2 und R explizit zu kennen!

5.7.12.1 Details der sicheren Implementierung mittels teilweiser Aktualisierung von $[[B]]$

Für eine sichere Implementierung der Berechnung von $[[J]]$ bestehen zwei Möglichkeiten: Einerseits kann $[[J]]$ in jeder Iteration neu berechnet werden. Andererseits kann das bereits vorhandene $[[J]]$ aktualisiert werden. Dabei kann jedoch nicht so vorgegangen werden, wie in [GI83] beschrieben, denn dazu müsste die Möglichkeit bestehen, einzelne Elemente der Matrizen $[[B]]$, $[[Q]]$ und $[[R]]$ explizit anzusprechen. Zudem hinge ein solches Vorgehen zur Laufzeit immer von der Größe der aktiven Menge ab, die nicht preisgegeben werden darf.

Bei der sicheren Implementierung hängen die Größen der Matrizen $[[B]]$ und $[[R]]$ hingegen nur eingeschränkt von der Größe der aktiven Menge q ab: Allenfalls in den ersten $l < n$ Iterationen, in denen die aktive Menge nicht mehr als l Elemente umfassen *kann*, wird $[[B]]$ entsprechend auf l Elemente verkleinert. Da allerdings, insbesondere bei kleinen Dimensionen, oft schon wenige Iterationen ausreichen, das Maximum zu bestimmen (vgl. Abschnitt 7), kann dies trotzdem nennenswerte Effizienzgewinne bedeuten. Im Normalfall sind diese Matrizen quadratisch mit der Dimension n, was der maximalen Größe der aktiven Menge entspricht (bei nicht-degenerierten Nebenbedingungen).

Die hier vorliegende Implementierung geht einen Mittelweg zwischen einer vollkommenen Neuberechnung von $[[J]]$ in jeder Iteration und der effizienten, nicht-sicher implementierbaren Implementierung aus [GI83]: In jeder Iteration wird genau eine Spalte von $[[B]]$ mittels Vorwärtssubstitution neu berechnet.

Protokoll 5.19: $([[J]],[[M]]) \leftarrow \mathsf{J}\left([[B]],[[L]],[[P]],\overrightarrow{[W^C]},\overrightarrow{[n_p]},\overrightarrow{[Num]},\left[t_{\stackrel{?}{=}t_1}\right],\left[t_{\stackrel{?}{=}t_2}\right],[NNB],It.,m,n\right)$

$l := \min(Iteration, n)$
If (1. Iteration)
  $\overrightarrow{[B(*,1)]} \leftarrow \mathsf{VWSubs}\left([[L]],\overrightarrow{[n_p]},n\right)$ //Protokoll 4.4
  $[[B_H]] \leftarrow \overrightarrow{[B(*,1)]}$
Else
  $[S] \leftarrow [1]$
  $\overrightarrow{[n^{\pm}]} \leftarrow [t_{\stackrel{?}{=}t_2}] \cdot \overrightarrow{[n_p]} + [t_{\stackrel{?}{=}t_1}] \cdot \overrightarrow{[0]}$ //n Mul $\mathbb{F}_q$
  $\overrightarrow{[HV]} \leftarrow \mathsf{VWSubs}\left([[L]],\overrightarrow{[n^{\pm}]},n\right)$ //Protokoll 4.4
  For$(i = 1,\ldots,l)$
    $[NGN] \leftarrow \mathsf{EQZ}([Num(i)],k)$ //vgl. Tabelle 2.1
    $[IG] \leftarrow \mathsf{EQ}([Num(i)],[NNB],k)$ //vgl. Tabelle 2.1
    $[NKN] \leftarrow \mathsf{LT}([NNB],[Num(i)],k)$ //vgl. Tabelle 2.1
    $[H2] \leftarrow [S] \cdot [NGN]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
    $[H] \leftarrow [NKN]\mathsf{OR}[H2]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
    $\overrightarrow{[B(*,i)]} \leftarrow [H] \cdot \overrightarrow{[B(*,i)]} + (1-[H]) \cdot \overrightarrow{[HV]}$ //$2n$ Mul, 1 Rnd $\mathbb{F}_q$
    $[H2] \leftarrow [Num(i)]$
    $[Num(i)] \leftarrow [NNB] \cdot [H] + (1-[H]) \cdot [Num(i)]$ //2 Mul, 1 Rnd $\mathbb{F}_q$
    $[NNB] \leftarrow [H] \cdot [H2] \cdot (1-[IG]) + [NNB] \cdot (1-[H]) + [IG] \cdot (m+1)$ //3 M, 1 R $\mathbb{F}_q$
    $\overrightarrow{[HV]} \leftarrow [H] \cdot \overrightarrow{[B(*,i)]} + (1-[H]) \cdot \overrightarrow{[HV]}$ //$2n$ Mul, 1 Rnd $\mathbb{F}_q$
    $\overrightarrow{[HV]} \leftarrow (1-[IG]) \cdot \overrightarrow{[HV]}$ //n Mul, 1 Rnd $\mathbb{F}_q$
    $[S] \leftarrow [S] \cdot (1-[NGN])$ // 1 Mul, 1 Rnd $\mathbb{F}_q$
  $\left([[B_H]] \quad 0\right) \leftarrow \mathsf{KompaktifiziereSpaltenweiseBinär}\left([[B]],\overrightarrow{[W_C]},n,m,l\right)$ //s. 3.8
$\left([[M]],\overrightarrow{[w]},\overrightarrow{[\beta]}\right) \leftarrow \mathsf{QR}([[B_H]],n,l)$ //Protokoll 4.9
$[[Q]] \leftarrow \mathsf{ComputeQ}\left([[Q]],\overrightarrow{[w]},\overrightarrow{[\beta]},Iteration\right)$ //vgl. Tabelle 4.1
For $(i = 1,\ldots,n,)$ **do parallel**
  $\overrightarrow{[J(*,i)]} \leftarrow \mathsf{RWSubs}\left([[L^t]],\overrightarrow{[Q(*,i)]},n\right)$ //Protokoll 4.5
Return $([[J]],[[M]])$

Abbildung 5.19: Aktualisierung von J

Falls $T = t_2$, also eine neue Nebenbedingung aktiv wird, wird dazu die entsprechende Zeile der Matrix A verwendet, andernfalls, d.h. wenn eine Nebenbedingung fallengelassen wird, der Null-Vektor (von dem die Vorwärtssubstitution - zumindest bei Verwendung von Protokoll 4.4 - wieder Null ist!).

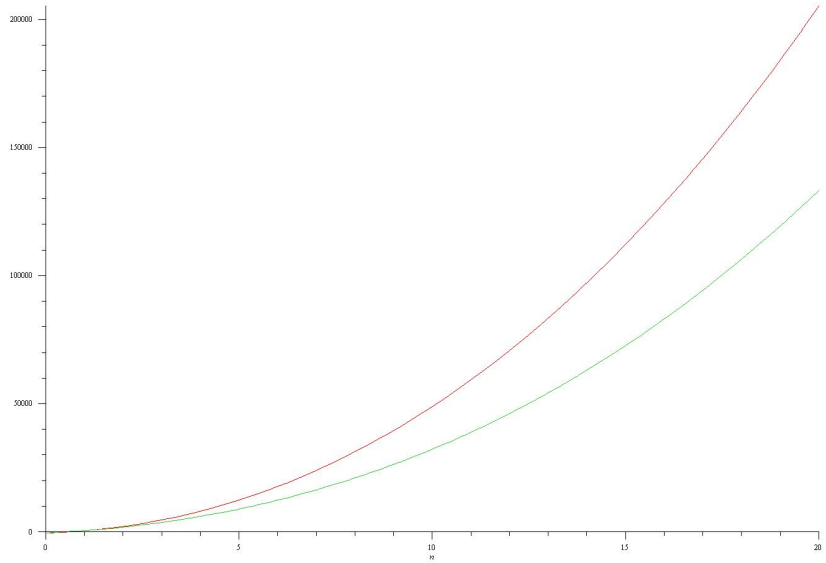

Abbildung 5.20: Kosten der Neuberechnung von B (rot) und der Aktualisierung von B (grün) in Abhängigkeit von der Anzahl der Variablen n

Bei der Speicherung der Vektoren $\overrightarrow{L^{-1}n_p}$ in B muss die *Reihenfolge* der Nebenbedingungen gewahrt bleiben! D.h. die Indizes der zu den Spalten in $\tilde{N}$ korrespondierenden Zeilen in A müssen zu jedem Zeitpunkt strikt aufsteigend sortiert sein. Andernfalls können in späteren Schritten Nebenbedingungen u.U. nicht korrekt entfernt werden. Dazu ist es nötig in einem Vektor $\overrightarrow{[Num]}$ die Indizes der aktiven Nebenbedingungen zu speichern und der Routine, die $[[B]]$ aktualisiert, zusätzlich den Index $[[NNB]]$ der neu hinzugenommenen Nebenbedingung zu übergeben, in $[[B]]$ ggf. einen Teil der Spalten nach rechts zu verschieben (dies geschieht iterativ), und $\overrightarrow{[Num]}$ zu aktualisieren. Im Detail wird iterativ der Index der neuen Nebenbedingung mit den zur aktiven Menge gehörigen Indizes verglichen und die Spalte $[[L^{-t}]]\overrightarrow{[n_p]}$ richtig in die Matrix $[[N]]$ eingeordnet. Danach wird sukzessive in jeder Iteration eine Spalte um eine Position nach rechts verschoben. Dies geschieht durch sichere Implementierung einiger **if**-Abfragen (Abschnitt 1.3). Ist die Aktualisierung der Spalten abgeschlossen wird die Matrix $[[B]]$ kompaktifiziert, die QR-Zerlegung berechnet und mit deren Hilfe $[[J]]$ ermittelt. Als weitere Modifikation wird die Matrix $\tilde{N}$

in den ersten $l \leq n$ Iterationen, in denen sie nicht mehr als l Spalten besitzen *kann* auf diese l Spalten beschränkt.

Im Detail kann eine sichere Aktualisierung von $[[J]]$ auf die folgende Weise erfolgen (Protokoll 5.19): In der ersten Iteration ist notwendigerweise $T = t_2$ und die Größe der aktiven Menge 0. Es kann also die aktivierte Nebenbedingung mit $[[L]]$ vorwärtssubstituiert und in die 1. Spalte von $[[B]]$ geschrieben werden (Schritte 2-4). In späteren Iterationen wird nur die neu aktivierte Nebenbedingung bzw. die Nullspalte (wenn eine Nebenbedingung entfernt wird), mit $[[L]]$ vorwärtssubstituiert (Schritte 7 und 8). Nun wird iterativ der Index $[NNB]$ der neu aktivierten Nebenbedingung mit 0 (Schritt 10) und in jeder Iteration mit dem nächsthöheren Index der aktiven Menge (diese sind im Vektor $\overrightarrow{[Num]}$ kodiert) verglichen (Schritte 11 und 12). Ist der neue Index gleich Null (d.h. es wird eine Nebenbedingung aus der aktiven Menge entfernt) oder echt kleiner als $Num(i)$, ändert sich nichts in $[[B]]$, ansonsten wird die Spalte i von $[[B]]$ durch die gespeicherte Spalte $\overrightarrow{[HV]}$ (diese ist in der 1. Iteration gleich $L^{-1}\overrightarrow{n_p}$) ersetzt (Schritt 15). Im Anschluss daran wird $\overrightarrow{[Num]}$ aktualisiert (Schritte 16 und 17) und ggf. $\overrightarrow{[HV]}$ durch $\overrightarrow{[B(*,i)]}$ ersetzt (Schritte 19 und 20). Gleichzeitig wird der Index $[NNB]$ (der mit dem Index der neuen Nebenbedingung $\overrightarrow{[n_p]}$ initialisiert ist) der Spalte $\overrightarrow{[HV]}$ aktualisiert (Schritt 18):[2] Wenn sich in $[[B]]$ nichts ändert, bleibt auch $[NNB]$ unverändert, ansonsten nimmt es den Index der soeben verdrängten Nebenbedingung an. Wenn die Spalte auf 0 gesetzt wurde, erhält $[NNB]$ den Wert $[n+1]$; dies sorgt dafür, dass in den folgenden Iterationen keine Veränderungen an $[[B]]$ mehr vorgenommen werden. Nach der Iteration werden die Nicht-Null-Spalten am linken Rand von $[[B]]$ zusammengeschoben (Schritt 22). Da wegen LICQ nie mehr als n Nebenbedingungen aktiv sind, hat die resultierende Matrix $[[B_H]]$ nicht mehr als l (vgl. Schritt 1) Spalten. Von $[[B_H]]$ wird dann die QR-Zerlegung berechnet (Schritt 23). Beachte dabei, dass falls $l \neq n$ $[[B_H]]$ nicht quadratisch ist, Protokoll 4.9 entsprechend angepasst werden muss. Zudem besitzt dann die Matrix $[[R]]$ genau $n - q$ Spalten, ist also ggf. nicht quadratisch. Wie oben bereits besprochen, stellt dies für die Rückwärtssubstitution jedoch kein Problem dar. $[[Q]]$ selbst wird in Schritt 24 berechnet und davon ausgehend die Matrix $[[J]]$ in den Schritten 25 und 26.

Die Komplexität des Verfahrens - im Fall $r = n$ und $m = 2n$ - ist in Tabelle 5.3 zu sehen. Schritte 6 und 7, 8 und 9 sowie die Rückwärtssubstitutionen (Schritte 15 und 16) können jeweils parallelisiert werden.

[2] In der protoypischen Implementierung werden Nebenbedingungen von $1, \ldots, n$ nummeriert i.G. zu den Spalten von $[[B]]$, deren Zählung mit 0 beginnt. Auf diese Weise kann das Fallenlassen einer Nebenbedingung mit [0] kodiert werden.

Protokoll 5.21: $\overrightarrow{[a_p]} \leftarrow$ **SecReadVektor_Sgn** $\left([[A]], \overrightarrow{[I]}, \overrightarrow{[Sgn]}, m, n, g\right)$

```
1  For (i = 1, ..., m)
2      For (j = 1, ..., n)
3          [a_p(j)] ← [a_p(j)] + [A(i,j)] · [I(i)]
4      If (i ≤ g)
5          [a_p(j)] ← [a_p(j)] · [−Sgn(j)]
6  Return [a_p]⃗
```

Abbildung 5.21: Modifikation des sicheren Lesens der Nebenbedingungen, so dass für ausgewählte Gleichheitsnebenbedingungen immer gilt $a_i^t x - b_i < 0$

5.7.12.2 Im Vergleich: Neuberechnung von $[[B]]$

Im Vergleich dazu kann eine vollständige Neuberechnung von $[[J]]$ durchgeführt werden. Im Unterschied zu Protokoll 5.19 wird dabei die Vorwärtssubstitution mit allen n Spalten von $[[N]]$ durchgeführt. Andererseits können die Zeilen 6-9 sowie 11 weggelassen werden. Ein Vergleich der Komplexitäten der Aktualisierung von B mit der vollständigen Neuberechnung ist in Abb. 5.20 zu sehen. Für 20 Gleichungen ergibt sich für $k = 128$ und $f = 64$ und 40 Nebenbedingungen ein Vorteil von ca. 48%!

5.7.13 Gleichheitsnebenbedingungen

Die Veränderungen am Algorithmus bei zusätzlich vorhandenen Gleichheitsnebenbedingungen ähneln denen beim primalen Algorithmus (Abschnitt 5.5.6): Die Überprüfung, ob $S \geq 0$ in Schritt 27 von Protokoll 5.13, wird ersetzt durch $S \stackrel{?}{=} 0$ für $i \in \mathcal{E}$. Ist $a_i^t x = b_i$ eine Gleichheitsbedingung, so wird sie durch die verletzte Ungleichheitsbedingung $a_i^t x < b$ bzw. $a_i^t x \geq b$ ausgedrückt. Dazu multipliziert man $s_i = a_i^t x_k - b_i$ für $(i \in \mathcal{E})$, a_i und b_i mit $-sgn(s_i)$. Die letzten beiden allerdings nur, wenn a_i in Schritt 1 ausgewählt wird. Dies geschieht beim sicheren Auslesen des Vektors $\overrightarrow{[A(i,*)]}$ bzw. des Werts $[b(i)]$ und kann mit Hilfe einer leichten Modifikation (Protokoll 5.21) von Protokoll SecReadVector aus [CdH10b] erreicht werden.

Übergeben werden an Protokoll 5.21 die Matrix $[[A]]$, der Vektor $\overrightarrow{[I]}$, der die auszuwählende Nebenbedingung kodiert sowie der Vektor $\overrightarrow{[Sgn]}$, der die Vorzeichen von $[s_i]$ beinhaltet und zusätzlich noch die Dimensionen der Matrix $[[A]]$, m und n sowie die Anzahl der Gleichheitsnebenbedingungen g. Die Tatsache, dass das Residuum einer nicht-erfüllten Gleichheitsnebenbedingung

immer mit einem negativen Vorzeichen dargestellt wird, stellt sicher, dass alle Gleichheitsnebenbedingungen bei ordnungsgemäßer Beendigung des Programms erfüllt sind.

Zusätzlich ergeben sich Einschränkungen bei der Anwendbarkeit der Algorithmen zur Auswahl der verletzten Nebenbedingung (Abschnitt 5.7.4.1). Die G-Norm-Strategie erscheint bei Hinzunahme von Gleichheitsnebenbedingungen nicht mehr wirtschaftlich: Da Gleichheitsnebenbedingungen mit verschiedenen Vorzeichen dargestellt werden können, kann die in Abschnitt 5.7.4.1 besprochene Vereinfachung, die Nenner $\|a_i^t\|_{G^{-1}}$ nur einmal zu berechnen, nicht durchgeführt werden. Die Rechnung müsste für $i \in \mathcal{E}$ bei jedem Aufruf von NB_G_Norm neu durchgeführt werden. Dies erscheint nicht sinnvoll. Es verbleiben noch die Euklidische-Norm-Strategie und die Residuumsstrategie. Bei diesen müssen die Vorzeichen von a_i, b_i, $i \in \mathcal{E}$ durch Multiplikation mit $-sgn(s_i)$ entsprechend angepasst werden.

5.8 Sicherheit

Bei den sicheren Implementierungen beider Optimierungsalgorithmen werden, außer den Abbruchkriterien, keine weiteren Daten bekannt. Gleiches gilt für die selbstentwickelten Bausteine wie die Verfahren zum Lösen linearer Gleichungssysteme oder die Wurzelfunktion. Da diese jedoch wiederum hauptsächlich aus Protokollen der sicheren Fixpunktarithmetik bestehen, die teilweise nur statistische Sicherheit bieten, sind die Optimierungsalgorithmen statistisch, aber nicht vollkommen sicher. Da die Sicherheitsparameter (vgl. Definition 3), die den „Grad“ der statistischen Ununterscheidbarkeit festlegen frei, gewählt werden können, ist dies praktisch jedoch keine allzu große Einschränkung.

5.9 Grenzen der Parallelisierung am Beispiel TruncPr

An der Funktion TruncPr (vgl. Tabelle 2.1) lässt sich exemplarisch veranschaulichen, dass Funktionen bei begrenzten Ressourcen, die in der Praxis immer gegeben sind, sich nicht unbegrenzt parallelisieren lassen (vgl. Abschnitt 1.4): In Protokoll TruncPr aus [CdH10b] können im Prinzip *alle* sicheren Multiplikationen über dem Körper $\mathbb{F}_{q_1}$ in *allen* aufgerufenen Protokollen parallel ausgeführt bzw. vorausberechnet werden. Aber selbst bei den in Kapitel 7 vorgestellten - vergleichsweise kleinen - Testbeispielen können leicht mehrere tausend oder zehntausend solcher Multiplikationen auftreten. Diese lassen sich praktisch nicht parallelisieren. Dieses Beispiel unterstreicht den Gedankengang aus Abschnitt 1.4.2, die Rundenkomplexität eines Protokolls sehr kritisch zu sehen.

6 Anwendungen

In diesem Kapitel werden zwei Anwendungen der Quadratischen Programmierung vorgestellt und Möglichkeiten zu ihrer sicheren Implementierung erläutert. Einerseits ist dies die Sequentielle Quadratische Programmierung (SQP), die zur Lösung nicht-linearer und nicht-quadratischer Optimierungsprobleme verwendet (Abschnitt 6.1) wird. Es werden Ansätze zu einer sicheren Implementierung erörtert. Die andere Anwendung, auf der der Fokus dieses Kapitels liegt, ist die Klassifikation von Datensätzen mit Hilfe von Support Vector Machines (SVM, Abschnitt 6.2). Deren sichere Implementierung ermöglicht eine datenschutzfreundliche Klassifizierung in horizontal partitionierten Datenbanken. Ein Beispiel dafür ist finanzinstitutübergreifendes Credit-Scoring.

6.1 Sequentielle Quadratische Programmierung

Sequentielle quadratische Programmierung (SQP) ist eines der wichtigsten Werkzeuge der nicht-linearen Optimierung. Eine gute Einführung findet sich z.B in [NW99]. Die wichtigsten Elemente und ihre (potentielle) Anwendbarkeit im Zusammenhang mit den vorgestellten sicheren Implementierungen von Algorithmen zur Lösung quadratischer Optimierungsprobleme sollen hier kurz vorgestellt werden. Sei

$$f(x) \tag{6.1}$$

eine nicht-lineare, zweifach differenzierbare Zielfunktion, die unter den ebenfalls nicht-linearen Nebenbedingungen

$$c_i(x) = 0 \quad i \in \mathcal{E} \tag{6.2}$$
$$c_i(x) \geq 0 \quad i \in \mathcal{I} \tag{6.3}$$

minimiert werden soll. Die Idee ist, die Zielfunktion durch eine quadratische Funktion und die Nebenbedingungen durch lineare Ausdrücke zu approximieren, das resultierende quadratische Optimierungsproblem zu lösen und mit dem neuen Iterationspunkt eine neue quadratische Approximation (mit linear approximierten Nebenbedingungen) zu berechnen. Analog zu den Algorithmen aus Kapitel 5 wird auch hier nicht explizit nach den Lösungen x_k der

quadratischen Optimierungsprobleme gesucht, sondern nur nach dem Korrekturschritt p von x_k nach x_{k+1}. [1] Die Darstellung in diesem Abschnitt orientiert sich weitgehend an [NW99].

6.1.1 Optimierung Nicht-Linearer Funktionen unter Gleichheitsbedingungen

Man betrachte zunächst den Fall, dass nur Gleichheitsbedingungen gegeben sind. Die meisten Verfahren sind Aktive-Mengen-Strategien und in jedem Schritt äquivalent zu einem Newton-Verfahren: Bei temporärer Außerachtlassung der Ungleichheitsbedingungen ist die Lagrange-Funktion gegeben durch

$$\mathcal{L}(x, \lambda) = f(x) - \lambda^t c(x) \tag{6.4}$$

und die Matrix $A(x)^t = \begin{pmatrix} \nabla c_1(x) & \dots & \nabla c_m(x) \end{pmatrix}$ ist die Jacobi-Matrix der Nebenbedingungen. Die KKT-Bedingungen (vgl. (5.4)-(5.8)) lassen sich dann in dem Gleichungssystem

$$F(x, \lambda) = \begin{pmatrix} \nabla f(x) - A(x)^t \lambda \\ c(x) \end{pmatrix} = 0 \tag{6.5}$$

ausdrücken. Ein Lösungsansatz ist das Newton-Verfahren: Die Jacobi-Matrix von (6.5) ist dabei gegeben durch[2]

$$M := \begin{pmatrix} W(x, \lambda) & -A(x)^t \\ A(x) & 0 \end{pmatrix}. \tag{6.6}$$

Dabei ist $W(x, \lambda)$ die Hessematrix der Lagrangefunktion[3]

$$W(x, \lambda) = H(\mathcal{L}(x, \lambda)). \tag{6.7}$$

M ist die Matrix, die in Schritt k das KKT-System

$$M \cdot \begin{pmatrix} p_k \\ p_\lambda \end{pmatrix} = \begin{pmatrix} -\nabla f_k + A_k^t \lambda_k \\ -c_k \end{pmatrix} \tag{6.8}$$

löst. Es seien die folgenden beiden Annahmen erfüllt:

- Die Jacobi-Matrix A_k habe vollen Zeilenrang.

[1] Es sei dabei zunächst angenommen, dass keine Probleme aufgrund falscher Skalierung der Koeffizienten der Zielfunktion bzw. der Nebenbedingungen auftreten. Ggf. können die einzelnen quadratischen Teilprobleme skaliert werden. Vgl. Abschnitt 5.2.2

[2] Beachte die Ähnlichkeit zu (5.35).

[3] Diese kann numerisch bestimmt werden; z.B. mit Hilfe der BFGS-Methode([NW99]); diese lässt sich - da für festgewählte Parameter explizit gegeben - als sichere Mehrparteienberechnung implementieren.

- Die Matrix W_k ist positiv definit auf dem Tangentialraum der Nebenbedingungen, d.h. $d^t W_k d > 0$ für alle $d \neq 0$, derart dass $A_k d = 0$.

Dann kann die oben besprochene Newton-Iteration auch betrachtet werden als das *quadratische Minimierungsproblem*

$$\min_p \frac{1}{2} p^t W_k p + \nabla f_k^t p \tag{6.9}$$

unter

$$A_k p + c_k = 0. \tag{6.10}$$

Man sieht leicht ([NW99]), dass unter den obigen Annahmen dies eine Lösung liefert, die zu der Lösung, die durch das Newton-Verfahren (6.5,6.6) gewonnen wird, äquivalent ist.

6.1.2 Ungleichheitsbedingungen

Sind zusätzlich Ungleichheitsbedingungen gegeben, so stellt sich das in Schritt k zu lösende Teilproblem wie folgt dar ([NW99]):

$$\min \frac{1}{2} p^t W_k p + \nabla f_k^t p \tag{6.11}$$

unter

$$\begin{aligned} \nabla c_i(x_k)^t p + c_i(x_k) &= 0 && (6.12) \\ \nabla c_i(x_k)^t p + c_i(x_k) &\geq 0 && (6.13) \end{aligned}$$

6.1.3 Lösungsansätze als Sichere Mehrparteienberechnungen

Die beschriebenen quadratischen Optimierungsprobleme lassen sich mit den bekannten Methoden - darunter den in Kapitel 5 vorgestellten - lösen. Besonders der Algorithmus von Goldfarb und Idnani (Abschnitt 5.6) wird jedoch häufig empfohlen und angewandt ([Pow85],[Sch02],[TK91],[MW93]). Von großem Vorteil ist dabei das folgende

Theorem 1. *Sei x^* eine Lösung von (6.1)-(6.3) und seien die beiden Annahmen aus Abschnitt 6.1.1 und Gleichung (5.8) erfüllt. Ist dann (x_k, λ_k) hinreichend nahe an (x^*, λ^*), gibt es eine lokale Lösung von Teilproblem (6.11)-(6.13), dessen aktive Menge W_k identisch ist mit der an x^*.*

Dies bedeutet, dass für die iterierten quadratischen Optimierungsprobleme ein sogenannter *Warmstart* möglich ist: Die aktive Menge an der optimalen Lösung des vorherigen Problems kann als Schätzwert für die erste Iteration des neuen Problems verwendet werden. Dies kann die Anzahl der notwendigen Iterationen des Algorithmus von Goldfarb und Idnani, die zur Lösung des Teilproblems benötigt werden, drastisch reduzieren ([Pow85]).

6.2 Institutübergreifendes Credit-Scoring mit Hilfe sicherer, verteilter Support Vector Machines

In diesem Abschnitt soll ein Modell für ein *finanzinstitutübergreifendes Credit-Scoring-System* basierend auf Support Vector Machines konstruiert werden.

Ein *Credit-Scoring-System* ist ein mathematisches Modell, das ein Kreditinstitut dabei unterstützt, die Bonität eines Kunden einzuschätzen. Ein derartiges Modell basiert u.a. auf den persönlichen Daten des Kunden, d.h. Alter, Einkommen, Art des Einkommens, Kredithistorie, etc.. Es können aber auch auf den ersten Blick als nicht relevant erscheinende Daten, wie z.B. der Wohnort, eine Rolle spielen. Anhand der „Ähnlichkeit“ der Daten eines neuen Kunden zu denen von Kunden, die einen Kredit bedient bzw. nicht bedient haben, wird nun eine Einschätzung über die Bonität des neuen Kunden abgegeben. Die Vorteile, die Kredithistorien von Kunden mehrerer Kreditinstitute als Datenbasis für ein derartiges Modell heranzuziehen, liegen auf der Hand: Eine breitere Datenbasis sollte zu einer verbesserten Bonitätseinschätzung von potentiellen Kreditnehmern führen. Einerseits kann sich auf diese Weise zeigen, dass einem Kunden, dessen Bonität auf Grund der schmaleren Datenbasis „seines“ Kreditinstituts schlecht bzw. zweifelhaft eingeschätzt wurde, durch Vergleich mit den Kundendaten anderer Banken doch kreditwürdig erscheint. Andererseits kann ein Kunde schlechter Bonität eindeutiger als nicht kreditwürdig eingestuft werden. Problematisch ist dabei allerdings die Tatsache, dass Kundendaten hochsensibel sind und i.d.R. von der besitzenden Bank nicht an Dritte weitergegeben werden dürfen. Einen Ausweg bieten sichere Mehrparteienberechnungen: Im Folgenden soll gezeigt werden, wie ein Klassifizierer für das Credit-Scoring auf Basis von Kundendaten von mehreren Kreditinstituten berechnet werden kann, ohne die Vertraulichkeit der Daten zu gefährden! Er wird als Support Vector Machine umgesetzt. Ein allgemeiner Überblick über die Theorie des Credit-Scoring findet sich z.B. in [And07].

6.2.1 Lineare Support Vector Machines

Support Vector Machines sind eines der wichtigsten Werkzeuge des maschinellen Lernens. Ziel ist, zu bestimmen, ob ein bestimmter Datensatz zu ei-

ner vorher definierten Klasse gehört oder nicht. Es handelt sich dabei um ein zweistufiges Verfahren: Im ersten Schritt wird anhand von *Trainingsdaten* (dargestellt jeweils als Vektor), deren Klassenzugehörigkeit bekannt ist, eine Hyperebene konstruiert, die die Trainingsdaten möglichst klar trennt. Im zweiten Schritt werden die zu klassifizierende Testdaten in die Klassen eingeordnet, je nachdem auf welcher Seite der Hyperebene sie sich befinden. Ist eine solche Trennung der Trainingsdaten eindeutig möglich, spricht man von *Hard Margin Support Vector Machines*, andernfalls von *Soft Margin Support Vector Machines*. Ist die Trennfläche ein affin linearer Unterraum, spricht man von *Linearen Support Vector Machines*, andernfalls von *Nichtlinearen Support Vector Machines*.

Die folgenden Abschnitte sind ein kurzer Abriss über die Theorie der Support Vector Machines. Ausführlichere Darstellungen finden sich z.B. in [BGV92], [Bak07] oder [CST00].

Es sei ein Datensatz von Trainingspunkten $\mathcal{M} = \{(x_1, \kappa_1), \ldots, (x_N, \kappa_N)\}$ gegeben. Dabei sei $x_i \in \mathbb{R}^m$, $1 \leq i \leq N$ und $\kappa_i \in \{-1, 1\}$. Der Wert von κ_i besagt, ob der Punkt x_i zu der zu klassifizierenden Menge M gehört oder nicht. Die folgende Definition ist aus [Bak07].

Definition 13 (Lineare Separierbarkeit)**.** *Die Klassen M und $\bar{M}$ heißen bzgl. $\mathcal{M}$ genau dann linear separierbar, wenn eine Hyperebene der Dimension $m-1$ existiert, so dass alle Datensätze mit $\kappa = 1$ sich auf der einen Seite von $\mathcal{L}$ befinden und alle Datensätze mit $\kappa = -1$ auf der anderen.*

Existiert eine solche Ebene, so lässt sie sich durch eine Gleichung der Form

$$H : g(x) = w_0 + \langle x, w \rangle = 0 \tag{6.14}$$

beschreiben. Dabei ist $w = \begin{pmatrix} w_1 & \ldots & w_m \end{pmatrix}^t$ ein Vektor und $w_0 \in \mathbb{R}$. Es gelte o.E. für alle i mit $\kappa_i = 1$, $g(x_i) > 0$ und für alle i mit $\kappa_i < 1$ $g(x_i) < 0$. Man normiert w und w_0 so, dass im ersten Fall $g(x_i) \geq 1$ und im zweiten Fall $g(x_i) \leq -1$. Die Punkte, für die Gleichheit gilt, bezeichnet man als *Support-Vektoren*. Die Ungleichungen lassen sich zusammenfassen zu

$$\kappa_j \cdot g(x_j) = \kappa_j \cdot (w_0 + \langle x_j, w \rangle) \geq 1 \quad 1 \leq j \leq m. \tag{6.15}$$

Betrachtet man die Support-Vektoren x_j, so liegen diese auf den Hyperebenen

$$H_M : w_0 + \langle x, w \rangle = 1 \text{ bzw. } H_{\bar{M}} : w_0 + \langle x, w \rangle = -1. \tag{6.16}$$

Der Abstand zwischen H und H_M bzw. H und $H_{\bar{M}}$ ist offensichtlich gegeben durch

$$\rho_{\mathcal{M}}(w, w_0) = \frac{1}{\|w\|}. \tag{6.17}$$

Man bezeichnet $\rho_{\mathcal{M}}(w, w_0)$ auch als *Trennbreite* von $\mathcal{M}$ bzgl. w und w_0.

Bemerkungen:

1. Gibt es (für eine Seite) *weniger* als m Support-Vektoren, so ist die Ebene H_M (bzw. $H_{\bar{M}}$) nicht eindeutig.

2. Ist die trennende Hyperebene nicht eindeutig festgelegt, so definiert man die *optimale Hyperebene* als die, die die Trennbreite (6.17) maximiert. Durch deren Verwendung gewinnt das Verfahren an Robustheit ([Bak07]).

Um die Trennbreite $\rho_{\mathcal{M}}(w, w_0) = \frac{1}{\|w\|}$ zu maximieren, muss das quadratische Optimierungsproblem

$$\text{Minimiere } \frac{1}{2}\|w\|^2 \tag{6.18}$$

unter den Nebenbedingungen

$$\kappa_j \cdot (w_0 + \langle x_j, w\rangle) \geq 1 \quad \forall\, 1 \leq j \leq N \tag{6.19}$$

gelöst werden. Aus Gründen auf die weiter unten näher eingegangen wird, ist es jedoch häufig besser, das *duale Problem* zu lösen. Dazu bildet man die zu (6.18) und (6.19) gehörige Lagrangefunktion

$$\mathcal{L}(w, w_0, \lambda) := \frac{1}{2}\|w\|^2 + \sum_i \lambda_i \cdot (1 - \kappa_i(\langle w, x_i\rangle + w_0)). \tag{6.20}$$

Aus (5.4) folgt, dass an einem Extremum $\hat{w}$ gelten muss

$$\hat{w} = \sum_i \lambda_i \kappa_i x_i \text{ und } 0 = \frac{\partial \mathcal{L}}{\partial w_0} = \sum_i \lambda_i \kappa_i. \tag{6.21}$$

Mit Hilfe dieser Gleichungen lässt sich (6.20) transformieren zu

$$\mathcal{L}(\lambda) = \sum_i \lambda_i - \frac{1}{2} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_N \end{pmatrix}^t \cdot diag(\kappa_i) \cdot G \cdot diag(\kappa_i) \cdot \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_N \end{pmatrix}. \tag{6.22}$$

Dabei ist G die Gram-Matrix definiert durch $G_{ij} = \langle x_i, x_j\rangle$. Man erhält also das Optimierungsproblem ([PTVF07])

$$\text{Maximiere } \sum_i \lambda_i - \frac{1}{2}\lambda^t \cdot diag(\kappa) \cdot G \cdot diag(\kappa) \cdot \lambda \tag{6.23}$$

unter

$$\lambda_j > 0 \quad \forall j \text{ und } \lambda_j \cdot \kappa_j = 0 \quad (vgl.\, 6.21.) \tag{6.24}$$

Mit Hilfe von (6.21) lässt sich die Lösung des primalen Problems aus der des dualen ableiten. w_0 kann dann mit Hilfe der Gleichung

$$w_0 = -\frac{\max_{\kappa_i=-1}(\langle w, x_i\rangle) + \min_{\kappa_i=1}(\langle w, x_i\rangle)}{2} \tag{6.25}$$

bestimmt werden.[4]. Die Gründe, wegen denen es vorteilhaft ist, das duale anstelle des primalen Problems zu betrachten, sind:

1. Die Lösung des primalen Problems kann numerisch schwierig sein ([Bak07], [CM07],[Joa00]).
2. Das Problem transformiert sich von einem, dessen Dimension mit der Größe der Datenpunkte wächst, zu einem, das mit der Anzahl der Datenpunkte wächst!

6.2.2 Lineare Soft-Margin Support Vector Machines

In den meisten Fällen ist es unrealistisch anzunehmen, dass sich die Klassen linear separieren lassen. Man definiert deswegen für jeden Punkt eine Schlupfvariable ξ_i. Ist $\xi_i = 0$, so liegt der zugehörige Punkt x_i in der „richtigen" Halbebene. Ist $\xi > 0$, lässt sich der Punkt nicht linear separieren. Die Größe von ξ gibt die Abweichung an. Um die Abweichungen zu minimieren, wird die Summe der ξ's als Strafterm zur Zielfunktion hinzugefügt. Man erhält das primale quadratische Optimierungsproblem ([PTVF07]).

$$\text{Minimiere } \frac{1}{2}\|w\|^2 + \alpha \cdot \sum_i \xi_i \tag{6.26}$$

unter den Nebenbedingungen

$$\xi_i \geq 0 \tag{6.27}$$

$$\kappa_i(\langle w, x_i\rangle + w_0) \geq 1 - \xi_i \tag{6.28}$$

Die Größe des frei wählbaren Operators α gibt an, ob der Fokus eher auf einer großen Trennbreite (kleines α, mit einer ggf. höheren Anzahl falsch klassifizierter Punkte) oder auf einer möglichst guten Klassifikation der Trainingsdaten (großes α, mit einer kleinen Trennbreite) liegt. Das zugehörige duale Problem hat die folgende Gestalt ([PTVF07])

$$\text{Maximiere } \sum_i \lambda - \frac{1}{2}\lambda^t \cdot diag(\kappa) \cdot G \cdot diag(\kappa) \cdot \lambda \tag{6.29}$$

unter

$$0 \leq \lambda_i \leq \alpha \tag{6.30}$$

$$\lambda \cdot \kappa = 0 \tag{6.31}$$

[4]Die Gleichung gilt, da für Support Vektoren $g(x_i) = 1$ oder $g(x_i) = -1$.

6.2.3 Nicht-Lineare Support Vector Machines

Um linear nicht separierbare Klassen zu trennen, verwendet man den sogenannten *Kernel-Trick*. Durch eine nicht-lineare Transformation φ werden die Punkte x_i in einem höher-dimensionalen Raum V abgebildet, in dem die Bilder linear separierbar sind. Die in V definierte Hyperebene ist dann

$$f(x) = \langle W, \varphi(x) \rangle + W_0 \tag{6.32}$$

und das primale Optimierungsproblem (vgl. (6.26))

$$\text{Minimiere } \frac{1}{2}\|W\|^2 + \alpha \sum_i \Xi_i \tag{6.33}$$

unter

$$\begin{aligned} \Xi_i &\geq 0 && (6.34) \\ \kappa_i(\langle W, \varphi(x_i) \rangle + W_0) &\geq 1 - \Xi_i. && (6.35) \end{aligned}$$

Das duale Problem hat allerdings die (wesentlich einfacher zu handhabende Gestalt)

$$\text{Minimiere } \frac{1}{2}\lambda^t \cdot diag(\kappa) \cdot K \cdot diag(\kappa) \cdot \lambda - \sum_i \lambda_i \tag{6.36}$$

unter

$$\begin{aligned} 0 &\leq \lambda_i \leq \alpha && (6.37) \\ \lambda \cdot \kappa &= 0. && (6.38) \end{aligned}$$

Dabei ist die *Kernel-Matrix* K gegeben durch $K_{ij} = \langle \varphi(x_i), \varphi(x_j) \rangle$. Die genaue Kenntnis von φ ist also gar nicht nötig und die Größe des Problems hängt nur von der Anzahl der Punkte x_i ab und nicht von der Dimension von V! Grundlegende Eigenschaften eines jeden Kernels sind ([PTVF07])

- K ist symmetrisch und positiv semidefinit
- Multinomiale Kombinationen von Kerneln sind wieder Kernel
- $K(\sigma(x_i), \sigma(x_j))$ ist ein Kernel für jede Abbildung σ, wenn $K(\cdot, \cdot)$ einer ist
- $K(x_i, x_j) = g(x_i)g(x_j)$ ist ein Kernel für jede (skalare) Funktion g

Offensichtlich ist für positiv definites K die Matrix $diag(\kappa) \cdot K \cdot diag(\kappa)$ wieder positiv definit. Einige bekannte Kernel sind

- Linear: $K_{ij} = \langle x_i, x_j \rangle$
- Exponentiell: $K_{ij} = \langle x_i, x_j \rangle^d$
- Polynomial: $K_{ij} = (a\langle x_i, x_j \rangle + b)^d$
- Sigmoid: $K_{ij} = \tanh(a\langle x_i, x_j \rangle + b)$
- Gauß-Radial-Basis: $K_{ij} = \exp\left(-\frac{1}{2}|x_i - x_j|^2/\sigma^2\right)$
- Coulomb: $(\|x - y\|^2 + \varepsilon)^{-\delta}$ (vgl. [HMO02])

Zur Wahl der Koeffizienten siehe z.B. [PTVF07].
Möchte man einen Testdatensatz klassifizieren, so ist noch die Entscheidungsregel anzugeben, die die Testdaten den einzelnen Klassen zuordnet. Ist x ein zu klassifizierender Vektor, so ist diese gegeben durch

$$f(x) = sgn\left(\sum_i \hat{\lambda}_i \kappa_i K(x_i, x) + \hat{b}\right). \tag{6.39}$$

Dabei ist $\hat{b}$ die Translation der berechneten Hyperebene und kann berechnet werden als gewichteter Mittelwert ([PTVF07])

$$\hat{b} = \sum_i \hat{\lambda}_i(\alpha - \hat{\lambda}_i)\left(\kappa_i - \sum_j \hat{\lambda}_j \kappa_j K(x_i, x_j)\right) / \sum_i \hat{\lambda}_i(\alpha - \hat{\lambda}_i). \tag{6.40}$$

6.2.4 Lösung des Quadratischen Optimierungsproblems

Zur Lösung des Quadratischen Optimierungsproblems (6.36) unter (6.37) und (6.38) können sowohl allgemeine - wie die in Kapitel 5 vorgestellten Methoden - als auch spezialisiertere, auf die Lösung von Support Vector Machines zugeschnittene, Verfahren verwendet werden. Deren Einsatz ist v.a. dann vorteilhaft, wenn die Anzahl der Datenpunkte sehr hoch ist, so dass die Größe der Kernel-Matrix den verfügbaren Arbeitsspeicher übersteigt. In diesem Fall kann das Optimierungsproblem so in Teilprobleme zerlegt werden, dass immer nur eine Teilmenge der Nebenbedingungen (im Extremfall nur 2) betrachtet werden muss. Ein (frei verfügbares) Software-Paket, das dieses Vorgehen umsetzt, ist [CL11]. Es beruht u.a. auf den Ideen aus [REPHCJJ05] über eine dafür geeignete Zerlegung der Nebenbedingungen. Eine weitergehende Übersicht über Lösungsmethoden für Support Vector Machines findet sich in [CM07].

Für „kleinere“ Probleme sind die in Kapitel 5 beschriebenen Lösungsverfahren aber durchaus ausreichend. Insbesondere der duale Algorithmus erzielt sehr

gute Ergebnisse (Abschnitt 7.8). Um auch „größere“ Klassifizierungsprobleme sicher mit Support Vector Machines bearbeiten zu können, müssten aber auch einige der oben erwähnten, speziellen Algorithmen sicher implementiert werden. Da es sich dabei ebenfalls um Aktive Mengen Strategien handelt, die eine ähnliche Gestalt besitzen wie die primale, die in Kapitel 5 vorgestellte, erscheint dies durchaus möglich. Leider überstieg dies jedoch den Umfang dieser Arbeit.

6.2.5 Anwendung auf Credit Scoring

Die Verwendung von Support Vector Machines zum Credit-Scoring ist vieldiskutiert. Grundlegend werden die Trainingsdaten danach klassifiziert, ob ein gewährter Kredit vollständig und vertragsgemäß getilgt wurde. Es werden dann davon ausgehend - und nach Konstruktion der trennenden Hyperebene - die Testdaten als kreditwürdig bzw. kreditunwürdig eingestuft.

Eine Einführung findet sich z.B. in [SS05] oder [BC09]. Auch die Deutsche Bundesbank zieht Support Vector Machines als unterstützendes Werkzeug zu Rate, um die Kreditwürdigkeit von Firmen zu beurteilen ([Deu10]). Für die Vorhersage über die künftige Solvenz von Firmen sind sie ebenfalls geeignet, wie eine Untersuchung von 150.000 Firmen über eine Zeitraum von 5 Jahren zeigt ([AM08]): Dabei waren Support Vector Machines anderen Werkzeugen zur Beurteilung der Kreditwürdigkeit, wie z.B. Logische Regression (LR), Linearer Diskriminanten Analyse (LDA) oder k-Nearest Neighbors (kNN) gleichwertig bzw. leicht überlegend ([BC09]). Eine Anwendung auf reale Kreditfälle findet sich in [SS05]. Insbesondere in Fällen, in denen Zweifel bestehen, können SVMs gute Entscheidungshilfen sein. Eine Anwendung auf das Rating von Banken auf der Grundlage finanzieller Kenndaten und dem (bekannten) Rating anderer Banken ist in [VGBGVD03] beschrieben. Auch hier sind Support Vector Machines anderen Verfahren überlegen. Ein weiterer Vergleich von Support Vector Machines und anderen Modellen des Credit Scoring ([SS03]) legt nahe, dass diese die besten Ergebnisse liefern, auch wenn man zusätzlich Opportunitätskosten betrachtet, d.h. entgangene Einnahmen aus einem nicht-gewährten Kredit (die i.d.R. niedriger sind als die, die auftreten, wenn ein Kredit teilweise oder ganz ausfällt). Allen Modellen gemeinsam ist u.a. die moderate Anzahl an Variablen (nicht mehr als 40) und die Tatsache, dass sich die Fehlerquote (d.h. die Summe beider Fehler) nicht unter 20% bewegt. Dies liegt u.a. daran, dass einige der verwendeten Kriterien einen nicht-monotonen Einfluss auf die Wahrscheinlichkeit, dass ein Kredit bedient wird, haben können: Zum Beispiel ist es denkbar, dass Personen mit hohem oder niedrigem Einkommen eine höhere Kreditausfallwahrscheinlichkeit tragen als solche mit mittlerem ([SS05]). Dies bedeutet aber auch, dass Support Vector Machines nur als unterstützendes Werkzeug für eine Kreditwürdigkeitsbewertung zum Einsatz kommen können.

6.2.6 Wahl der Parameter

Die Wahl der Parameter, insbesondere der des Multiplikators α für die Strafterme Ξ_i, sind stark vom Problem abhängig und müssen anhand der Trainingsdaten ermittelt werden. Optimale Werte für α bei Verwendung eines linearen Kernels schwanken zwischen 0.001 ([BC09]) und 100 ([SS05]).

6.2.7 Wahl des Kernels

Die Wahl der Kernelfunktion trägt maßgeblich zum Erfolg der SVM bei. Häufig ([SS06],[SS05]) wird ein Gaußscher- oder Coulomb-Kernel eingesetzt, aber auch die wesentlich leichter handhabbaren linearen und polynomialen Kernel haben bereits sehr gute Ergebnisse geliefert ([BC09],[SS05]), insbesondere bei Verwendung zur Kreditwürdigkeitsbewertung.

6.2.8 Vor- und Nachteile von Support Vector Machines als Klassifikationswerkzeug

Gegenüber anderen Klassifikationsmethoden besitzen Support Vector Machines einige Vorteile: Sie kommen gut mit nicht-regulären Daten zurecht und sind durch die Möglichkeiten, den Kernel zu variieren sowie den Parameter α (6.37) zu verändern, sehr flexibel. Auch können Parameter, die bei anderen Verfahren Probleme bereiten und deswegen ausgeschlossen werden, problemlos integriert werden ([AM08]).

Auf der anderen Seite sind die Ergebnisse von Support Vector Machines nicht transparent. Der relative Beitrag - insbesondere bei nicht-linearen Kernelfunktionen - einer einzelnen Komponente zur resultierenden Trennebene, ist nicht klar erkennbar. Es ist jedoch möglich, die entscheidenden Merkmale zu identifizieren, indem man die Gewichte der berechneten Hyperebene betrachtet und diejenigen auswählt, die eine bestimmte Größe überschreiten ([BC09]).

6.2.9 Allgemeines zur Verwendung zum datenschutzfreundlichen Credit-Scoring

Die Tatsache, dass linearer und quadratischer Kernel, die leicht sicher implementierbar sind, bei der Kreditwürdigkeitsbewertung sehr gute Ergebnisse liefern (Abschnitt 6.2.7), macht Support Vector Machines für eine sichere verteilte Kreditwürdigkeitsbewertung interessant. Ein weiteres Argument ist die Tatsache, dass sie nur mit der Anzahl der Datenpunkte skalieren und so die Größe der zu optimierenden Fragestellung flexibel angepasst werden kann. Es erscheint z.B. sinnvoller Klassifizierer für Firmen, die gewisse Eigenschaften (z.B. Branche, Größe, Umsatz, etc.) gemeinsam haben, zu erstellen, als einen, der

Daten von Kleinstunternehmen wie von internationalen Konzernen beinhaltet. Auf diese Weise können institutübergreifend Klassifizierer für bestimmte Wirtschaftssegmente erstellt werden, für die die Datenbasis nur *eines* Finanzunternehmens zu klein wäre. So kann zumindest teilweise kompensiert werden, dass - bei identischen Rechenressourcen - eine sicher verteilte Implementierung immer wesentlich ineffizienter ist als eine nicht-sichere und deswegen nur im Vergleich kleinere Problemstellungen bearbeitet werden können.

Leider war eine experimentelle Überprüfung des in den vorherigen Abschnitten beschriebenen Credit-Scoring-Modells beruhend auf Support Vector Machines an Echtdaten nicht möglich. Echtdaten sind nur äußerst schwer zu beschaffen und ihre Verwendung ist nur unter großen datenschützenden Auflagen möglich. Es war jedoch möglich, das Modell anhand von medizinischen Daten zu testen. Dies stellt einerseits eine gute Alternative dar, um die Funktionsfähigkeit des Programms zu verifizieren und Einschätzungen über seine Leistungsfähigkeit zu gewinnen. Andererseits stellt sich aber auch in diesem Themenbereich ein eigenständiges Datenschutzproblem: Wie kann die Vertraulichkeit von Patientendaten bei der Klassifizierung von Krebsarten (o.ä.) aufgrund von Trainingsdaten, die bei verschiedenen Krankenhäusern lagern, gewahrt bleiben? Die Kennzahlen bestehen häufig aus mehreren tausend Werten und können u.U., auch wenn sie anonymisiert sind, doch dazu verwendet werden, einzelne Patienten zu identifizieren. Eine Klassifizierung neuer Daten aufgrund datenbankübergreifender Testdaten ist dann auf klassischem Wege nicht ohne Weiteres möglich. Die Ergebnisse der im Rahmen dieser Arbeit vorgenommenen Tests an medizinischen Daten finden sich in Abschnitt 7.8.

6.2.10 Implementierung

Zur sicheren Implementierung der Support Vector Machines waren einige zusätzliche Protokolle nötig. Diese sollen hier vorgestellt werden. Zusammengefasst sind dies:

1. Erstellung der Kernel-Matrix (für lineare und polynomiale Kernel)
2. Erstellung der Matrix G des Quadratischen Optimierungsproblems
3. Implementierung der Entscheidungsregeln für Testdaten
4. Berechnung der Translation b der trennenden Hyperebene

Die Erstellung der Kernel-Matrix ist einfach und wird hier nicht extra beschrieben. Gleiches gilt für die Matrix G des quadratischen Optimierungsproblems. Um diese zu erhalten, muss einfach jeder Eintrag (i, j) der Kernel-Matrix mit $\kappa_i \cdot \kappa_j$ multipliziert werden. Die Implementierung der Entscheidungsregel für

Protokoll 6.1: $[B] \leftarrow$ **Entscheidungsregel** $\left([[A]], \overrightarrow{[x]}, \overrightarrow{[\lambda]}, \overrightarrow{[\kappa]}, [\hat{b}], m, n\right)$

1 $[B] \leftarrow \mathsf{PRSZ}$
2 **For**$(i = 1, \ldots, m)$
3 $\quad [Kx(i)] \leftarrow \mathsf{Inner}\left(\overrightarrow{[A(i,*)]}, \overrightarrow{[x]}, n\right)$ //1 Mul, 1 Rnd $\mathbb{F}_q$
4 $\quad [Kx(i)] \leftarrow \mathsf{TruncPr}([Kx(i)], k, f)$//1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul, 2 Rnd $\mathbb{F}_{q_1}$
5 $\quad [B] \leftarrow [B] + [\kappa(i)] \cdot [Kx(i)] \cdot [\hat{\lambda}(i)]$ //2 Mul, 2 Rnd $\mathbb{F}_q$
6 $[B] \leftarrow \mathsf{TruncPr}([B], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f + 1$ Mul $\mathbb{F}_{q_1}$
7 $[B] \leftarrow [B] + [\hat{b}]$
8 $[B] \leftarrow \mathsf{Signum}([B], k)$
9 **Return** $[B]$

Abbildung 6.1: Entscheidungsregel für einen Datensatz x anhand einer linearen Support Vector Machine S

den linearen Kernel (für eine andere Kernelfunktion müssen nur die Schritte 2 und 3 entsprechend ersetzt werden) ist in Protokoll 6.1 zu sehen.

Bei der Berechnung der Translation $[b]$ (Protokoll 6.2) werden in den Zeilen 4-6 zunächst die Support-Vektoren durch die Anwendung von GTZappr auf die Lagrange-Multiplikatoren bestimmt. Support-Vektoren erfüllen die entsprechende Nebenbedingung exakt und somit sind ihre Lagrange-Multiplikatoren (echt) größer als Null. In Zeile 7 wird jedes $[\lambda(i)]$ mit $[1]$ multipliziert, wenn es sich um einen Support Vektor handelt und mit $[0]$ sonst. So wird vermieden, dass Lagrange-Multiplikatoren, die aufgrund von Rundungsfehlern nahe Null sind, einen Einfluss auf die Berechnung von $[b]$ besitzen. Im Anschluss wird in den Zeilen 8-11 der Nenner von (6.40) berechnet. Darauf aufbauend, und die Summanden des Nenners verwendend, wird in den Zeilen 12-20 der Zähler berechnet. Beachte, dass das Produkt in Zeile 14, das einen Summanden der inneren Summe des Zählers darstellt, trotz der 3 Faktoren nur Länge $2k$-Bits besitzt, da $[\kappa(i)]$ *keine* Fixpunktzahl ist, sondern nur die Klassenzugehörigkeit von Datensatz i angibt. Im vorletzten Schritt (21) wird $[b]$ dann tatsächlich berechnet. Die Zeilen 8 und 9 können für alle i parallel ausgeführt werden. Gleiches gilt für die innere Schleife bei der Berechnung der Komponenten des Zählers und die Bestimmung der Support-Vektoren am Anfang.

Bemerkung: Bei der in diesem Abschnitt vorgestellten Implementierung werden die Support-Vektoren, die die trennende Hyperebene bestimmen, nicht bekannt gemacht und liegen nur als Shares vor. Die Klassifikation eines neuen Datensatzes erfolgt wieder in einem interaktiven Protokoll. Veröffentlicht man die trennende Hyperebene, können Mitspieler unter Zuhilfenahme der Kenntnis der eigenen Daten Erkenntnisse über die Lage der Daten der *anderen* Teilnehmer gewinnen.

Protokoll 6.2: $[b] \leftarrow$ **Translation** $\left([[A]], [[K]], \overrightarrow{[\lambda]}, \overrightarrow{[\kappa]}, m, n\right)$

1 $[M] \leftarrow$PRSZ
2 $[Y] \leftarrow$PRSZ
3 **For**$(i = 1, \ldots, m)$
4 $\quad [SV(i)] \leftarrow [\lambda(i)] \cdot [\lambda(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
5 $\quad [SV(i)] \leftarrow$TruncPr$([SV(i)], k, f)$//1 Mul, 1 Rnd $\mathbb{F}_q$, f+1 Mul, 2 Rnd $\mathbb{F}_{q_1}$
6 $\quad [SV(i)] \leftarrow$GTZappr$([SV(i)], k)$ //vgl. Tabelle 2.1
7 $\quad [\lambda(i)] \leftarrow [\lambda(i)] \cdot [SV(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
8 $\quad [N(i)] \leftarrow [\alpha] - [\lambda(i)]$
9 $\quad [N(i)] \leftarrow [N(i)] \cdot [\lambda(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
10 $\quad [N(i)] \leftarrow$TruncPr$([N(i)], k, f)$//1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul, 2 Rnd $\mathbb{F}_{q_1}$
11 $\quad [M] \leftarrow [M] + [N(i)]$
12 $\quad [Z(i)] \leftarrow$PRSZ
13 $\quad$ **For**$(j = 1, \ldots, m)$
14 $\quad\quad [z] \leftarrow [\lambda(j)] \cdot [\kappa(j)] \cdot [K(i,j)]$ //2 Mul, 2 Rnd $\mathbb{F}_q$
15 $\quad\quad [z] \leftarrow$TruncPr$([z], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
16 $\quad\quad [Z(i)] \leftarrow [Z(i)] + [z]$
17 $\quad [Z(i)] \leftarrow [\kappa(i)] - [Z(i)]$
18 $\quad [Z(i)] \leftarrow [Z(i)] \cdot [N(i)]$ //1 Mul, 1 Rnd $\mathbb{F}_q$
19 $\quad [Z(i)] \leftarrow$TruncPr$([Z(i)], k, f)$ //1 Mul, 1 Rnd $\mathbb{F}_q$, $f+1$ Mul $\mathbb{F}_{q_1}$
20 $\quad [Y] \leftarrow [Y] + [Z(i)]$
21 $[b] \leftarrow$FPDiv$([Y], [M], k, f)$ //s. Tabelle 2.1
22 **Return** $[b]$

Abbildung 6.2: Sichere Berechnung der Translation $[b]$

6.2.11 Implementierung mit Hilfe der primalen Aktiven-Mengen-Strategie

Die primale Implementierung setzt voraus, dass von beiden Klassen mindestens je ein Vertreter vorliegt (dies erscheint nicht allzu restriktiv); z.B. der erste Datensatz in der positiven und der letzte in der negativen Klasse. Dann ist ein zulässiger Startwert gegeben durch $x_0 = \left(\frac{\alpha}{2} \quad , \cdots , \quad \frac{\alpha}{2}\right)^t$[5]. An diesem Punkt aktiv sind die Gleichheitsnebenbedingung sowie die $m-2$ Nebenbedingungen $x_2 \geq 0, \ldots, x_{m-1} \geq 0$. Auf das Suchen eines zulässigen Startwerts (Phase I) kann also verzichtet werden. Ansonsten verfährt der Algorithmus so wie in Abschnitt 5.5.6 über die Beteiligung von Gleichheitnebenbedingungen beschrieben.

[5] Der Nullvektor wäre auch zulässig, erfüllt aber nicht LICQ!

6.2.12 Implementierung mit Hilfe des dualen Algorithmus

Die Implementierung mit Hilfe des dualen Algorithmus erfolgt wie in Abschnitt 5.7.13 beschrieben.

7 Theoretische Vergleiche und experimentelle Ergebnisse

In diesem Kapitel sollen die vorgestellten Algorithmen anhand von Testbeispielen evaluiert und - wo möglich - miteinander verglichen werden. Konkret werden alle vorgestellten Verfahren zum sicheren Lösen linearer Gleichungssysteme theoretisch, sowie Schnelligkeit und Genauigkeit der sicheren Implementierung der LR- und QR-Zerlegung bei zufälligen Matrizen und rechten Seiten, praktisch verglichen. Die sichere Berechnung der Wurzel wird ebenfalls an zufälligen Werten bzgl. Geschwindigkeit und Genauigkeit gemessen. Die verschiedenen Protokolle zum sicheren Lösen quadratischer Optimierungsprobleme und zur Berechnung von Support Vektor Maschinen werden anhand von Testbeispielen miteinander verglichen.

7.1 Technische Daten

Alle Test wurden durchgeführt auf einem Rechner ausgestattet mit einem Intel Core i7-2600 3,4GHz Prozessor und 8GB RAM. Das verwendete Betriebssystem war Windows 7 Professional 64. Die Algorithmen wurden in C++ implementiert. Alle Spieler wurden auf *einem* Rechner simuliert, d.h. echte Kommunikation fand nicht statt. Die Rechnungen mit großen Zahlen wurden durch Einsatz einer gepatchten Version der MPIR 2.5.1 ([MPI]; der Windows-Version der gmp [GMP]) ermöglicht.

7.2 Das allgemeine Mehrparteienszenario

Alle Algorithmen wurden in einem Szenario von fünf Spielern und Secret Sharing Polynomen vom Grad zwei getestet. Die Bitlänge des für das Secret Sharing verwendeten Modulus q war 1024. Des weiteren kamen für elementare Operationen auch Secret Sharing Schemes über den Körpern $\mathbb{F}_{q_1}$, bei einer Bitlänge von q_1 von 64 Bit und $\mathbb{F}_{2^8}$ zum Einsatz ([CS10],[CdH10a],[CdH10b]). Letzterer wurde als AES-Körper umgesetzt, für den gute Implementierungen öffentlich verfügbar sind. Die Länge der Fixpunktzahlen war in allen Versuchen, bis auf die, bei denen Effizienz und Genauigkeit der Wurzelfunktion getestet wurden (hier war $k = 110$ und $f = 80$, korrespondierend zu den Werten aus [Lie12]), $k = 128$, von denen $f = 64$ Nachkommastellen waren.

7.3 Komplexität von Algorithmen – Theoretisch gesehen

Die Komplexität eines Algorithmus wird durch die enthaltene Anzahl der sicheren Multiplikationen über den Körpern $\mathbb{F}_q$, $\mathbb{F}_{q_1}$ und $\mathbb{F}_{2^8}$ gemessen. Um Protokolle vergleichen zu können, die für alle drei Körper unterschiedliche Werte aufweisen, wird die Tatsache verwendet, dass die Kosten einer sicheren Multiplikation in Körpern $\mathbb{F}_p$ bei den verwendeten Multiplikationsprotokollen höchstens linear in der Bitlänge von p sind ([Lor09],[LW11]). Zwar lässt sich diese Aussage nicht ohne Weiteres auf den Körper $\mathbb{F}_{2^8}$ verallgemeinern, da einige Techniken, die in [Lor09],[LW11] verwendet werden, sich über $\mathbb{F}_{2^8}$ nicht anwenden lassen; jedoch sind die elementaren Rechenoperationen in diesem Körper auf Bit-Ebene und mit Hilfe von Tabellenaufrufen sehr effizient implementierbar, so dass die Kosten kaum ins Gewicht fallen sollten. Es wird deswegen *näherungsweise* angenommen, dass sich die Kosten auch hier linear in der Bitlänge verhalten. Konkret bedeutet dies, bei einer Länge von q von 1024 Bit und einer von q_1 von 64 Bit, dass eine Multiplikation in $\mathbb{F}_{q_1}$ $\frac{1}{16}$ sowie eine in $\mathbb{F}_{2^8}$ $\frac{1}{128}$ von einer in $\mathbb{F}_q$ kostet.

7.4 Komplexität von Algorithmen – Praktisch gesehen

Bis zu einem gewissen Grad konnte die These, dass sichere Multiplikationen über kleineren Körpern schneller durchführbar sind als in großen (siehe vorheriger Abschnitt), bestätigt werden: Eine sichere Multiplikation über dem Körper $\mathbb{F}_q$ benötigte im Schnitt (bei 100.000 Durchläufen) 0,000021s, eine über dem Körper $\mathbb{F}_{q_1}$ 0,000009s und eine über dem Körper $\mathbb{F}_{2^8}$ 0,000021s. Die Verwendung von $\mathbb{F}_{2^8}$ brachte also *keinen* Geschwindigkeitsvorteil! Über die Gründe kann hier nur spekuliert werden. Es ist denkbar, dass mit Hilfe der GMP die Multi-Core-Architektur des Prozessors besser genutzt werden kann, was die Berechnung der sicheren Multiplikationen über den Köpern $\mathbb{F}_q$ und $\mathbb{F}_{q_1}$ beschleunigen würde. Die Möglichkeit, dass eine schnellere Implementierung des Körpers $\mathbb{F}_{2^8}$ existiert, ist ebenfalls nicht von der Hand zu weisen. Auch über die Gründe, weshalb die Geschwindigkeit der sicheren Multiplikationen im Vergleich der Körper $\mathbb{F}_q$ und $\mathbb{F}_{q_1}$ nicht - wie vorhergesagt ([Lor09]) - linear wächst (d.h. hier um den Faktor 16), sondern nur um knapp 58% kann an dieser Stelle keine Aussage getroffen werden. Es ist aber denkbar, dass es ebenfalls in der Architektur der GMP begründet liegt oder systemspezifisch ist. Insbesondere bedeutet dies, dass die Abweichungen in einem anderen Umfeld auch wieder verschwinden könnten. Die Algorithmen wurden deshalb weitestgehend, d.h. insbesondere unter Verwendung der o.g. kleinen Körper für Bitoperationen, implementiert.

Eine weitergehende Analyse zeigt zusätzlich, dass die Metrik, die Komplexität eines sicheren, verteilten Algorithmus in der Anzahl der auftretenden sicheren Multiplikationen zu messen, im hier vorliegenden Fall, in dem alle Parteien auf einem Rechner innerhalb eines Programms dargestellt werden, nur begrenzt hilfreich ist. Sie beruht auf dem Gedanken, dass jeder sichere, verteilte Algorithmus im Kern aus sicheren Multiplikationen, für die eine Kommunikationsrunde zwischen den Spielern erforderlich ist und aus lokal durchführbaren Operationen besteht, deren Rechenzeit im Verhältnis zur Latenz des verwendeten Netzwerks nicht ins Gewicht fällt. Ist allerdings - wie hier - Kommunikation nur theoretisch, aber nicht *de-facto* gegeben, sind Vorhersagen über die Komplexität von Algorithmen auf dieser Grundlage nur näherungsweise möglich: Analysen der sicheren, verteilten Algorithmen mit Hilfe des Werkzeugs gprof ([GKM82]) zeigten, dass sichere Multiplikationen, Rekonstruktionen von Sharings und ähnliche Operationen, für die Kommunikation zwischen den Spielern erforderlich ist, nur wenige Prozent (der genaue Wert ist problemabhängig) der gesamten Rechenzeit ausmachten! Im Vergleich dazu benötigten die Methoden zum Umrechnen von Sharings zwischen den verschiedenen Körpern eine ähnliche Zeitspanne. Eine generelle Aussage über die Kosten eines Algorithmus bei Ausführung auf einem Rechner anhand der Zahl der sicheren Multiplikationen ist gleichwohl möglich: Häufig deutet eine hohe Rechenkomplexität auf ein generell komplexes Programm hin. Aus diesem Grund weichen tatsächliches und prognostiziertes Verhalten, insbesondere bei den Verfahren zum Lösen linearer Gleichungssysteme, teilweise erheblich voneinander ab. Dennoch hält die theoretische Aussage, dass die QR-Zerlegung bei zunehmender Größe des Gleichungssystems der LR-Zerlegung überlegen ist, einer praktischen Überprüfung stand. Es besteht jedoch Grund zu der Annahme, dass bei einer echten sicheren Mehrparteienberechnung, die auf mehreren Rechnern durchgeführt wird und bei der Latenzzeiten *nicht* null sind, sich die Algorithmen weit mehr wie prognostiziert verhalten, da die lokalen Operationen sehr schnell durchgeführt werden können und die Netzwerklatenz zeitkritisch wird.

x	a	b	c	d	e	f	g	h
abs. Fehler	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-83}$
rel. Fehler	$< 2^{-79}$	$< 2^{-79}$	$< 2^{-83}$	$< 2^{-83}$	$< 2^{-87}$	$< 2^{-87}$	$< 2^{-91}$	$< 2^{-95}$

Tabelle 7.1: Genauigkeit der Berechnung der Quadratwurzel

7.5 Der Wurzelalgorithmus

In den durchgeführten Experimenten wurden die acht Quadratwurzeln der Zahlen $a = 0.008585937, b = 0.146234375, c = 0.6326875, d = 11.19, e = 197.04, f = 3110.4, g = 489,291.776, h = 3,701,997.568$ berechnet. Da in dem vorgestellten Protokoll Eingabewerte zuerst normiert werden, sollte die Größe der Zahl weniger wichtig sein, als die Nähe von $2^{k-1} \leq s < 2^k, s \in \{a, b, c, d, e, f, g, h\}$ zu 2^k bzw. 2^{k-1}. Es wurde darauf geachtet, dass die Zahlen gleichmäßig verteilt waren, d.h. unabhängig von der Größe ist je eine Zahl in jedem Achtel des Intervalls $[2^{k-1}, 2^k[$ enthalten. Es wurden Fixpunktzahlen der Länge $k = 110$ Bit mit $f = 80$ Nachkommastellen verwendet. Der Betrag des absoluten Fehlers (die Differenz zwischen dem exakten Ergebnis und dem berechneten) war immer kleiner als 2^{-80}, d.h. exakt in Fixpunktdarstellung. Die Berechnungszeiten waren ca. $\approx 0,06s$ für alle Zahlen. Eine volle Goldschmidt-Iteration dauerte ca. 0,0035s und die verkürzte 0,0011s. Bemerkenswert ist, dass, obwohl keine Kommunikation stattfand, die Newton-Raphson Iteration am Schluss mit ca. 0,0055s ca. 60% teurer war! Dies rechtfertigt die Entscheidung, Goldschmidt-Iterationen für alle bis auf eine Iteration zu verwenden. Die Zahlen für die durchschnittliche Genauigkeit des Algorithmus, getestet an 1400 Zufallszahlen, finden sich in Tabelle 7.1.

7.6 Vergleich der Methoden zum Lösen von Gleichungssystemen

7.6.1 Allgemeine Verfahren – Theorie

Bei Wahl der übrigen Parameter wie in Abb. 7.1 und unter Einsatz eines unterliegenden $(5, 2)$-Shamir Secret Sharing Schemas ist unter den allgemein anwendbaren Verfahren in der Theorie ab ca. 15 Gleichungen der QR-Algorithmus dem Gaußverfahren überlegen. Dies liegt zum großen Teil begründet in der beim Gaußalgorithmus notwendigen Pivotisierung, die für jeden Eliminationsschritt die Bestimmung eines Maximums eines Vektors und eine Multiplikation der Matrix mit einer Permutationsmatrix erfordert, was bei zunehmender Größe der Matrizen teuer wird. Die Gesamtkosten hierfür liegen bei $\mathcal{O}(n^3)$ sicheren Multiplikationen über $\mathbb{F}_q$.

7.6.2 Verfahren für symmetrisch positiv definite Matrizen

Für symmetrisch positiv definite Matrizen ist vor allem das CG-Verfahren empfehlenswert. Im Gegensatz zu den anderen Verfahren besitzt die verteilte Implementierung nur quadratische und nicht kubische Komplexität (Tabelle

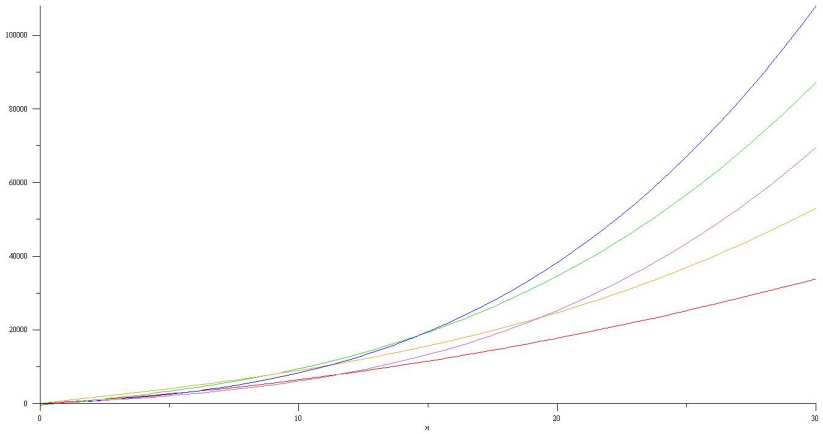

Abbildung 7.1: Vergleich der Verfahren zum Lösen linearer Gleichungssysteme für $k = 128$, $f = 64$, $\log_2 q \approx 1024$, $\log_2 q_1 \approx 64$. Der Graph des CG-Verfahrens ist rot, der des QR-Verfahrens grün, der der Cholesky-Zerlegung golden, der der LR-Zerlegung blau und der der LR-Zerlegung ohne Pivotisierung lila.

4.2). Die Gründe sind darin zu sehen, dass im Gegensatz zu den anderen Verfahren *keine* Matrix-Matrix-Multiplikationen notwendig sind und als „teure" Operationen pro Iteration nur jeweils 2 Divisionen anfallen. Von allen anderen Verfahren unterscheidet es sich jedoch auch dadurch, dass für eine neue rechte Seite der Algorithmus komplett neu gelöst werden muss.

Das Cholesky-Verfahren besitzt die selbe asymptotische Komplexität wie das QR-Verfahren (vgl. Tabelle 4.2). Für große n ist das QR- Verfahren zweimal so teuer wie das Cholesky-Verfahren (Abb. 7.2, es lässt sich aber auch - bei einer Bitlänge von q von 1024 Bit und einer von q_1 von 64 Bit unabhängig von k und f - analytisch zeigen). Analog kann man für dieselben Parameter zeigen, dass der Gaußalgorithmus für große n ca. dreimal so teuer ist wie das Cholesky-Verfahren (Abb. 7.2).

7.6.3 Experimentelle Ergebnisse

Die oben besprochenen Vergleiche von Gaußalgorithmus und QR-Verfahren wurden experimentell mit denselben Parametern überprüft. Ein experimenteller Vergleich von CG-Verfahren und Cholesky-Algorithmus unterblieb, da

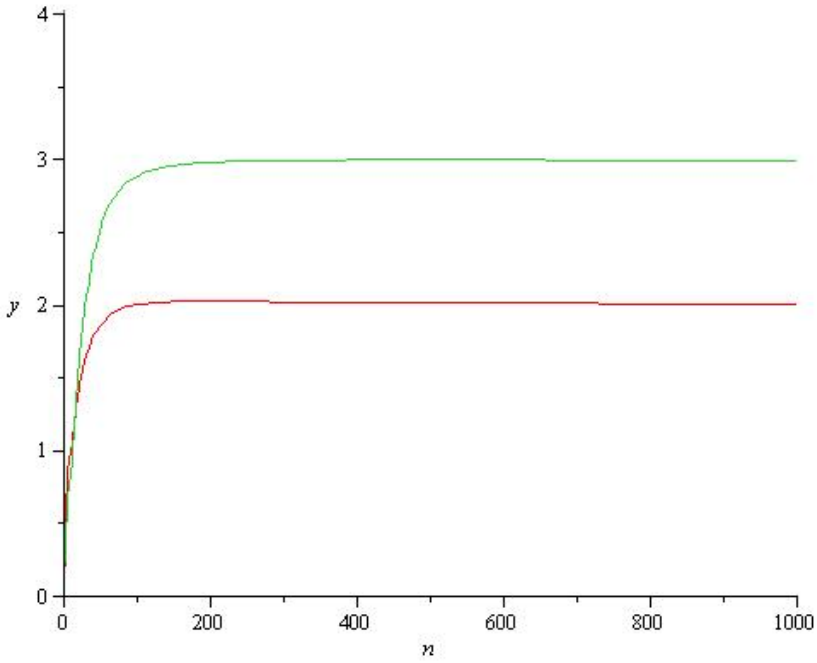

Abbildung 7.2: Relation der Komplexität des QR-Verfahrens zum Cholesky-Verfahren (rot) und des Gaußalgorithmus zum Cholesky-Verfahren (grün) für $k = 128$, $f = 64$, $\log_2 q_1 \approx \frac{1}{16} \log_2 q$.

für die in dieser Arbeit vorgestellten Algorithmen (fast)[1] keine Lösungen von linearen Gleichungssystemen beruhend auf symmetrisch positiv definiten Matrizen benötigt werden und die Erzeugung zufälliger, symmetrisch positiv definiter Matrizen, sehr aufwendig ist. Es wurden für jede Dimension jeweils 100 Gleichungssysteme mit zufälligen Koeffizienten zwischen -1024 und 1023 generiert und diese für ebenfalls zufällig erzeugte rechte Seiten b mit Werten im selben Bereich mit jeweils beiden Verfahren gelöst. Die Verfahren wurden bezüglich durchschnittlicher Rechenzeit und durchschnittlichem Fehler (d.h. Abweichung vom exakten Ergebnis, das zuvor mit einer nicht-sicheren Im-

[1] Im ersten Schritt des Algorithmus von Goldfarb und Idnani (Abb. 5.8) muss ein s.p.d Gleichungssystem gelöst werden. Allerdings muss die Cholesky-Zerlegung der entsprechenden Matrix auf jeden Fall durchgeführt werden, da die Faktoren später noch an anderer Stelle benötigt werden. Als zusätzliche Kosten fallen hier also nur Vorwärts- und Rückwärtssubstitution an. Diese sind für sich genommen billiger als die vollständige Ausführung des CG-Algorithmus.

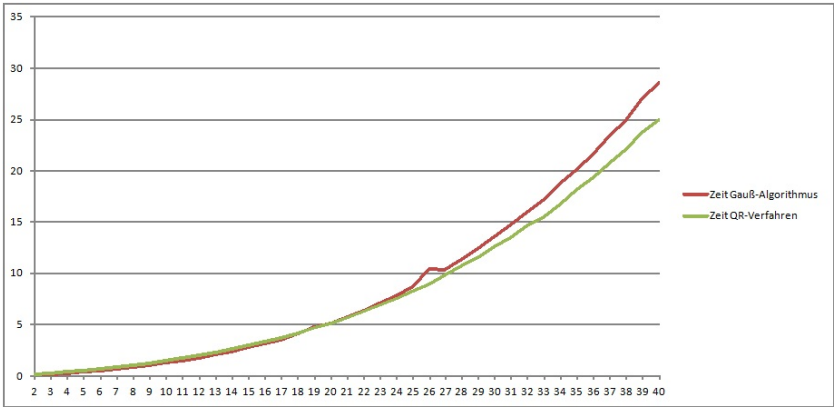

Abbildung 7.3: Durchschnittliche benötigte Zeit in s für das Lösen von linearen Gleichungssystemen mit Hilfe der LR- bzw. QR-Zerlegung in Abhängigkeit von der Dimension

plementierung des Gaußalgorithmus berechnet wurde) - berechnet mit Hilfe der Euklidischen Norm - verglichen. Um Verfälschung durch schlecht konditionierte Matrizen auszuschließen, wurden nur Matrizen verwendet, deren Frobenius-Kondition kleiner als 400 war (Auf diese Weise konnten auch singuläre Matrizen - bei zufälliger Belegung der Koeffizienten sowieso selten - ausgeschlossen werden). Die Ergebnisse finden sich in Abb. 7.3 und 7.4.

Auffällig ist, dass die LR-Zerlegung - im Gegensatz zu den theoretischen Betrachtungen - hinsichtlich Zeit und Effizienz für einen wesentlich größeren Dimensionsbereich überlegen bleibt. Erst ab einer Größe des Gleichungssystems von etwa 19 ist die QR-Zerlegung überlegen (Die genauen Werte finden sich in Anhang A).

Die Fehler, die bei der Lösung des Gleichungssystems mit Hilfe der LR-Zerlegung entstehen, befinden sich alle im Bereich von ca. 10^{-18} und die, die bei der Lösung mit Hilfe der QR-Zerlegung entstehen, im Bereich von 10^{-12}. Trotz des Unterschieds sind beide Fehlerraten sehr klein. Somit können beide Verfahren zur Lösung linearer Gleichungssysteme verwendet werden.

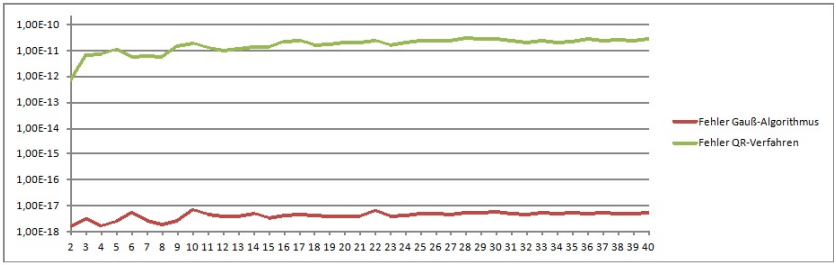

Abbildung 7.4: Durchschnittlicher Fehler (L^2-Norm) beim Lösen von linearen Gleichungssystemen mit Hilfe der LR- bzw. QR-Zerlegung in Abhängigkeit von der Dimension

7.7 Test der Algorithmen zur sicheren verteilten quadratischen Optimierung

7.7.1 Getestete Problemstellungen

Die getesteten Problemstellungen sind alle (bis auf die letzte; diese entstammt [NW99]) den Beispielsammlungen [SH81] und [Sch87] entnommen. Alle Probleme, die den Anforderungen der Algorithmen entsprechen (d.h. quadratische Optimierungsprobleme mit positiv definiter Matrix und linearen Nebenbedingungen), wurden ausgewählt. Der Reihe nach sind dies:

Problem 21:

$$f(x) = \frac{1}{2} x^t \begin{pmatrix} 0,02 & 0 \\ 0 & 2 \end{pmatrix} x \tag{7.1}$$

unter

$$\begin{aligned} 10x_1 - x_2 &\geq 10 \\ -x_1 &\geq -50 \\ x_1 &\geq 2 \\ -x_2 &\geq -50 \\ x_2 &\geq -50 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

$$f(x^*) = 0,04$$

Problem 35:

$$f(x) = \frac{1}{2}x^t \begin{pmatrix} 4 & 2 & 2 \\ 2 & 4 & 0 \\ 2 & 0 & 2 \end{pmatrix} x + x^t \begin{pmatrix} -8 \\ -6 \\ -4 \end{pmatrix} \tag{7.2}$$

unter

$$\begin{aligned} -x_1 - x_2 - 2x_3 &\geq -3 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ x_3 &\geq 0 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} \frac{4}{3} \\ \frac{7}{9} \\ \frac{4}{9} \end{pmatrix}$$

$$f(x^*) = \frac{1}{9}$$

Problem 76:

$$f(x) = \frac{1}{2}x^t \begin{pmatrix} 2 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} x + x^t \begin{pmatrix} -1 \\ -3 \\ 1 \\ -4 \end{pmatrix} \tag{7.3}$$

unter

$$\begin{aligned} -x_1 - 2x_2 - x_3 - x_4 &\geq -5 \\ -3x_1 - x_2 - 2x_3 + x_4 &\geq -4 \\ x_2 + 4x_3 &\geq 1,5 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ x_3 &\geq 0 \\ x_4 &\geq 0 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} \frac{3}{11} \\ \frac{23}{11} \\ -0,26 \cdot 10^{-10} \\ \frac{6}{11} \end{pmatrix}$$

$$f(x^*) = -4,681818181$$

Problem 224:

$$f(x) = \frac{1}{2} x^t \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix} x + x^t \begin{pmatrix} -48 \\ -40 \end{pmatrix} \tag{7.4}$$

unter

$$\begin{aligned} x_1 + 3x_2 &\geq 0 \\ -x_1 - 3x_2 &\geq -18 \\ x_1 + x_2 &\geq 0 \\ -x_1 - x_2 &\geq -8 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$$

$$f(x^*) = -304$$

Problem 268:

$$f(x) = \frac{1}{2} x^t \begin{pmatrix} 5098,5 & -6227 & -506,5 & 974 & 164,5 \\ -6227 & 10454 & -866,5 & -2457 & -93 \\ -506,5 & -866,5 & 877,5 & 544,5 & -87 \\ 974 & -2457 & 544,5 & 757,5 & -11 \\ 164,5 & -93 & -87 & -11 & 13,5 \end{pmatrix} x + x^t \begin{pmatrix} 18340 \\ -34198 \\ 4542 \\ 8672 \\ 86 \end{pmatrix} \tag{7.5}$$

unter

$$\begin{aligned} -x_1 - x_2 - x_3 - x_4 - x_5 &\geq -5 \\ 10x_1 + 10x_2 - 3x_3 + 5x_4 + 4x_5 &\geq 20 \\ -8x_1 + x_2 - 2x_3 - 5x_4 + 3x_5 &\geq -40 \\ 8x_1 - x_2 + 2x_3 + 5x_4 - 3x_5 &\geq 11 \\ -4x_1 - 2x_2 + 3x_3 - 5x_4 + x_5 &\geq -30 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} 1 \\ 2 \\ -1 \\ 3 \\ -4 \end{pmatrix}$$

$$f(x^*) = 0$$

Problem A:

$$\frac{1}{2} x^t \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} x + x^t \begin{pmatrix} -2 \\ 5 \end{pmatrix} \tag{7.6}$$

unter

$$\begin{aligned} x_1 - 2x_2 &\geq -2 \\ -x_1 - 2x_2 &\geq -6 \\ -x_1 + 2x_2 &\geq -2 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

Korrektes Ergebnis:

$$x^* = \begin{pmatrix} 1,4 \\ 1,7 \end{pmatrix}$$

$$f(x^*) = -6,45$$

7.7.2 Durchführung und Ergebnisse

Auf alle im vorigen Abschnitt genannten Problemstellungen wurde der primale sowie der duale Algorithmus angewandt. Da die Koeffizienten der Nebenbedingungen der Probleme alle dieselbe Größenordnung besitzen, wurde auf eine Skalierung verzichtet.

7.7.2.1 Ergebnisse des primalen Algorithmus

Bei der primalen Aktiven-Mengen-Strategie wurde das lineare Gleichungssystem (5.35) sowohl mit Hilfe der QR-Zerlegung, als auch mit Hilfe der LR-Zerlegung (Abschnitt 7) gelöst. Es bestätigt sich (Tabelle 7.2) das Ergebnis aus Abschnitt 7.6.3, dass bei den betrachteten Größen der Matrizen ($2n \times 2n \approx$

Problem	Simplex-It.	It.	Fehler ZF	Fehler x^*	t Phase 1	t Phase 2	t Gesamt
21	3	3	$< 10^{-19}$	$< 10^{-18}$	0,41s	1,28s	1,69s
35	0	8	$< 10^{-18}$	$< 10^{-18}$	0s	5,38s	5,38s
76	2	7	$< 10^{-9}$	$< 10^{-9}$	0,38s	7,48s	7,86s
224	0	6	$< 10^{-15}$	$< 10^{-17}$	0s	2,53s	2,54s
268	3	6	0	$< 10^{-14}$	0,60s	9,04s	9,64s
A	0	5	$< 10^{-15}$	$< 10^{-16}$	0s	2,21s	2,21s
21	3	3	$< 10^{-19}$	$< 10^{-18}$	0,39s	1,623s	2,028s
35	0	8	$< 10^{-18}$	$< 10^{-18}$	0s	6,67s	6,67s
76	2	7	$< 10^{-9}$	$< 10^{-9}$	0,37s	8,872s	9,242s
224	0	6	$< 10^{-15}$	$< 10^{-16}$	0s	3,167s	3,167s
268	3	9	0	$< 10^{-11}$	0,57s	15,17s	15,74s
A	0	5	$< 10^{-15}$	$< 10^{-16}$	0s	2,714s	2,714s

Tabelle 7.2: Iterationszahl, Genauigkeit (abs. Fehler) der Ergebnisse und Rechenzeit der Testprobleme bei Verwendung des primalen Algorithmus und Einsatz der LR-Zerlegung (oben) sowie der QR-Zerlegung (unten)

10×10 in den meisten Fällen; vgl. Abschnitt 5.5.5.2) der Gauß-Algorithmus deutlich schneller und effizienter ist. Die vergleichsweise schlechte Qualität des Ergebnisses von Problem 268 lässt sich durch die schlechte Frobenius-Kondition von mehr als 10^6 der Matrix G (5.13) erklären, genauso wie die um drei höhere Iterationsanzahl bei Verwendung des QR-Verfahrens im Vergleich zum Gaußalgorithmus. Das Ergebnis nach sechs Iterationen ist offensichtlich noch so schlecht, dass es vom Algorithmus verworfen wird. Erst der nach drei weiteren Iterationen erreichte Wert erfüllt die im Algorithmus gestellten Anforderungen an ein Optimum. Die aktive Menge am Optimum wird bei beiden Verfahren korrekt als leer erkannt. Von dieser Ausnahme abgesehen beeinflusst die Wahl des zum Lösen des Gleichungssystems verwendeten Algorithmus die Qualität der Lösungen nicht. Ein etwaiger Vorteil bei Einsatz der LR-Zerlegung verschwindet nach Durchführung der anderen Rechnungen vollständig.

7.7.2.2 Ergebnisse des dualen Algorithmus

Die Ergebnisse für den dualen Algorithmus (7.3) stützen die These, dass die Euklidische-Norm-Strategie bei wenigen Iterationen deutlich billiger ist als die G-Norm-Strategie (Abschnitt 5.7.4.1). Nur bei einem einzigen Problem (Problem 224) konnten so die Iterationen reduziert werden und das Optimierungsproblem effizienter gelöst werden, bei allen anderen Problemen war der Algorithmus mit der Euklidischen-Norm-Strategie nennenswert schneller. Die Ge-

nauigkeit der Ergebnisse war mit Fehlern $< 10^{-12}$ (zur Messung der Genauigkeit von x^* wurde die euklidische Norm verwendet) in allen Fällen zufriedenstellend. Eine kleine Ausnahme bildet Problem 76: Der relativ hohe Fehler in x^* ist fast vollständig auf einen Fehler in Höhe von $\approx 10^{-10}$ in der dritten Komponente zurückzuführen ($\approx 10^{-10}$), die sehr nahe bei 0 liegt (die anderen Fehler sind $\approx 10^{-19}$; dasselbe Phänomen tritt beim primalen Algorithmus auf). Dies wiederum kann auf die Tatsache zurückgeführt werden, dass Fixpunktarithmetik bei der Darstellung betragsmäßig sehr kleiner Zahlen zunehmend ungenau wird (Abschnitte 2.1.3 und 2.2).

Insgesamt zeigt die Auswertung der Ergebnisse für den primalen Algorithmus (Tabelle 7.2) eine i.d.R. deutlich höhere Rechenzeit für fast alle Testprobleme als für den dualen Algorithmus (Tabelle 7.3). Dies ist weniger auf das Finden eines zulässigen Startwerts zurückzuführen als vielmehr darauf, dass dieser häufig „weit“ vom Optimum entfernt liegt, was bis zu neun primale Iterationsschritte, wie z.B. bei Problem 268, nötig macht. An dem in Phase 1 ermittelten Ausgangspunkt $x_0 = \begin{pmatrix} 5 & 0 & 0 & 0 & 0 \end{pmatrix}^t$ sind zwei Nebenbedingungen aktiv, am optimalen Punkt x^* jedoch keine, d.h. der optimale Punkt ist gleichzeitig das globale Minimum der Zielfunktion. Auf dem Weg von x_0 zu x^* werden jedoch mehrfach Nebenbedingungen hinzugenommen und wieder fallengelassen, anstelle den direkten Weg, die an x_0 aktive Nebenbedingung fallenzulassen, zu nehmen. Ein weiteres Testproblem, bei dem ein ähnlicher Effekt zu beobachten ist, ist Problem 76.

Allgemein erscheint der duale Algorithmus besser für die Lösung sicherer, verteilter Optimierungsprobleme geeignet, als der primale, hauptsächlich weil er nur wenige Iterationen benötigt, um das Optimum zu erreichen, insbesondere dann, wenn im Optimum nur wenige Nebenbedingungen aktiv sind.

7.8 Test der sicheren verteilen Implementierung der Support Vektor Maschinen

7.8.1 Testbeschreibung

Der Algorithmus wurde anhand medizinischer Daten (Die Parameter der drei verwendeten Datensätze sind in Tabelle 7.4 abgebildet), die auf der `libsvm`[2]-Website (http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/) bereitgestellt sind, getestet. Ausgewählt wurden die Datensätze, die sich in zwei Klassen einteilen lassen und bei denen die Anzahl der Datensätze „klein“ war. Ziel war es nicht, für jeden Datensatz die optimale Kernelfunktion unter den optimalen Parametern zu bestimmen, sondern die sichere Implementierung

[2] eine freie Implementierung von Support Vektor Maschinen

Problem	Iterationen	Fehler ZF	Fehler x^*	Zeit
21	2	$< 10^{-18}$	$< 10^{-17}$	1,580s
35	2	$< 10^{-17}$	$< 10^{-18}$	2,025s
76	3	$< 10^{-9}$	$< 10^{-9}$	5,455s
224	2	$< 10^{-15}$	$< 10^{-18}$	1,380s
268	1	$< 10^{-12}$	$< 10^{-11}$	0,870s
A	2	$< 10^{-15}$	$< 10^{-16}$	1,610s
21	2	$< 10^{-18}$	$< 10^{-17}$	0,702s
35	2	$< 10^{-17}$	$< 10^{-18}$	0,920s
76	3	$< 10^{-9}$	$< 10^{-9}$	2,933s
224	4	$< 10^{-15}$	$< 10^{-17}$	2,137s
268	1	$< 10^{-12}$	$< 10^{-11}$	0,858s
A	2	$< 10^{-15}$	$< 10^{-16}$	0,717s

Tabelle 7.3: Iterationszahl, Genauigkeit (abs. Fehler) der Ergebnisse und Rechenzeit der Testprobleme bei Verwendung des Algorithmus von Goldfarb/Idnani und Einsatz der G-Norm Strategie (oben) sowie der Euklidischen-Norm-Strategie (unten)

anhand der als korrekt vorausgesetzten Ergebnisse, die `libsvm` für dieselben Datensätze unter denselben Parametern lieferte, zu validieren und so die Möglichkeit einer sicheren, verteilten Berechnung von Support-Vektor-Maschinen zu demonstrieren. Zusätzlich wurde noch die Präzision der Ergebnisse mit denen von `libsvm` gelieferten verglichen.

Für die hier vorgestellten Tests wurden nicht die im Internet bereitgestellten Rohdaten verwendet, sondern die mit dem im `libsvm`-Paket enthaltenen Skalierungstool `svm-scale` bearbeiteten. Dieses dient dazu, alle Spalten der gegebenen feature-Matrix auf ein festes Intervall (normalerweise $[-1, 1]$) zu skalieren. Es wäre möglich, dies sicher verteilt durchzuführen: Dafür müsste -

Problem	Anzahl Trainingsdaten	Anzahl Testdaten	Dimension
`leu`	38	34	7129
`colon`	30	32	2000
`duke`	38	4	7129

Tabelle 7.4: Parameter der Datensätze, die zum Testen der Support Vektor Maschine verwendet wurden

für die hier vorliegenden Testdaten - für alle 2000 bzw. 7129 Spalten der Matrix, der betragsmäßig größte Wert ermittelt werden (z.B. mit Protokoll 3.3). Im Anschluss daran müsste jeder Wert jeder Spalte durch den für die Spalte ermittelten Wert dividiert werden. Dies bedeutet für jedes Problem mehrere tausend Ausführungen von Protokoll 3.3 gefolgt von mehreren tausend Divisionen. Dieses Vorgehen - obwohl durchaus möglich - erschien zu aufwendig, zumal ja nur die Durchführbarkeit der sicheren verteilten Berechnung einer Support Vektor Maschine demonstriert werden sollte. Dabei steht die Skalierung, obwohl wichtig, nicht im Vordergrund. Bei einem Einsatz mit Echtdaten müsste das skizzierte Vorgehen (z.B. indem alle Attribute auf dieselbe Größenordnung skaliert werden; vgl. z.B. Protokoll 5.1) gleichwohl umgesetzt werden, was aber nur einen erhöhten, aber nicht prohibitiv großen Rechen- und Zeitaufwand bedeuten würde.

7.8.2 Testergebnisse

Die Support-Vektor-Maschinen, d.h. die trennende Hyperebene, wurde in allen Fällen mit beiden vorgestellten quadratischen Optimierungsmethoden, mit linearem und quadratischem Kernel, berechnet. Im linearen Fall wurde zum Lösen des Gleichungssystems das QR-Verfahren angewandt, da es für Glei-

Problem	`leu`	`colon`	`duke`
Zeit SV	688s	352s	548s
Zeit Verif.	34,1s	7,238s	4,173s
Anzahl Trainingsdaten	38	30	38
Anzahl Testdaten	34	32	4
Anzahl SV	29	23	31
Korrekte Zuordnung	88,24%	84,375%	75%
Korrekte Zuordnung libsvm	88,235%	84,375%	75%
Abs. Fehler x	$1,37 \cdot 10^{-6}$	$8.90 \cdot 10^{-5}$	$2,37 \cdot 10^{-6}$
Rel. Fehler x	$3,64 \cdot 10^{-4}$	$9.73 \cdot 10^{-3}$	$2,82 \cdot 10^{-4}$
Abs. Fehler f	$2,11 \cdot 10^{-7}$	$4.77 \cdot 10^{-7}$	$7,42 \cdot 10^{-9}$
Rel. Fehler f	$2,68 \cdot 10^{-5}$	$2.54 \cdot 10^{-5}$	$3,90 \cdot 10^{-7}$
Abs. Fehler $\hat{b}$	$1,26 \cdot 10^{-4}$	$2,32 \cdot 10^{-5}$	$1,60 \cdot 10^{-4}$
Rel. Fehler $\hat{b}$	$2,83 \cdot 10^{-4}$	$6,67 \cdot 10^{-5}$	$7,92 \cdot 10^{-4}$
α	0,5	0,5	0,5
Iterationszahl	11	9	9

Tabelle 7.5: Testergebnisse der SVM berechnet mit Hilfe des dualen Algorithmus mit linearem Kernel

Problem	`leu`	`colon`	`duke`
Zeit SV	11169s	3683s	10892s
Zeit Verif.	33,32s	6,423s	4,35s
Anzahl Trainingsdaten	38	30	38
Anzahl Testdaten	34	32	4
Anzahl SV	29	23	31
Korrekte Zuordnung	88,235%	84,375%	75%
Korrekte Zuordnung libsvm	88,235%	84,375%	75%
Abs. Fehler x	$1,37 \cdot 10^{-6}$	$8,90 \cdot 10^{-5}$	$2,37 \cdot 10^{-6}$
Rel. Fehler x	$3,64 \cdot 10^{-4}$	$9,73 \cdot 10^{-3}$	$2,82 \cdot 10^{-4}$
Abs. Fehler f	$2,12 \cdot 10^{-7}$	$4,76 \cdot 10^{-7}$	$7,42 \cdot 10^{-9}$
Rel. Fehler f	$2,68 \cdot 10^{-5}$	$2,54 \cdot 10^{-5}$	$3,90 \cdot 10^{-7}$
Abs. Fehler $\hat{b}$	$1,27 \cdot 10^{-4}$	$2,32 \cdot 10^{-5}$	$1,60 \cdot 10^{-4}$
Rel. Fehler $\hat{b}$	$2,83 \cdot 10^{-4}$	$6,67 \cdot 10^{-5}$	$7,92 \cdot 10^{-4}$
α	0,5	0,5	0,5
Iterationszahl	62	47	60

Tabelle 7.6: Testergebnisse der SVM berechnet mit Hilfe des primalen Algorithmus mit linearem Kernel

chungssysteme der auftretenden Größenordnung performanter ist (vgl. Abschnitt 7.6). Die Tests an linearen Gleichungssystemen (Abschnitt 7.6) legen nahe, dass eine Lösung mit Hilfe der LR-Zerlegung bessere Ergebnisse liefert als eine mit Hilfe der QR-Zerlegung. Jedoch waren die Ergebnisse - bei Einsatz der LR-Zerlegung und des linearen Kernels - nahezu identisch (und wurden deswegen hier nicht extra aufgeführt) zu denen der QR-Zerlegung, allerdings war die Berechnungszeit bis zu 40% länger! Der Einsatz der LR-Zerlegung liefert in diesem Zusammenhang also keine Vorteile.

Für den dualen Algorithmus wurde die Euklidische-Norm-Strategie gewählt, da die G-Norm-Strategie in Problemen mit Gleichheitsbedingungen nicht zur Verfügung steht (Abschnitt 5.7.13). Die Parameter - α im linearen Fall und zusätzlich a und b im quadratischen Fall - wurden mit `libsvm` so bestimmt, dass sie möglichst gute Ergebnisse liefern.

In den Erläuterungen zu den Beispielen ([CL11]) wird bereits darauf hingewiesen, dass die hier betrachteten mit dem linearen Kernel die geringste Fehlerquote besitzen. Dies konnte in den hier durchgeführten Experimenten im Prinzip bestätigt werden. Zudem zeigt sich - mehr noch als bei den Testbeispielen zur quadratischen Optimierung - die Überlegenheit des dualen Algorithmus: Er ist wesentlich schneller als der primale, bei vergleichbarer Qualität der Ergebnisse.

Problem	`leu`	`colon`	`duke`
Zeit SV	700s	120s	164s
Zeit Verif.	35,96s	8,26s	4,27s
Anzahl Trainingsdaten	38	30	38
Anzahl Testdaten	34	32	4
Anzahl SV	29	28	36
Korrekte Zuordnung	88,235%	84,375%	75%
Korrekte Zuordnung libsvm	88,235%	84,375%	75%
Abs. Fehler x	$9,89 \cdot 10^{-5}$	$6,93 \cdot 10^{-5}$	$8,08 \cdot 10^{-10}$
Rel. Fehler x	$5,26 \cdot 10^{-4}$	$5,44 \cdot 10^{-4}$	$2,79 \cdot 10^{-4}$
Abs. Fehler f	$3,00 \cdot 10^{-6}$	$2,03 \cdot 10^{-7}$	$1,04 \cdot 10^{-7}$
Rel. Fehler f	$7,60 \cdot 10^{-6}$	$6,77 \cdot 10^{-7}$	$1,48 \cdot 10^{-2}$
Abs. Fehler $\hat{b}$	$3,03 \cdot 10^{-5}$	$5,77 \cdot 10^{-5}$	$5,71 \cdot 10^{-5}$
Rel. Fehler $\hat{b}$	$6,79 \cdot 10^{-5}$	$1,91 \cdot 10^{-4}$	$1,96 \cdot 10^{-4}$
α	1,0	1,0	1,0
a	0,0001	0,01	1
b	100	1	1
Iterationszahl	11	4	4

Tabelle 7.7: Testergebnisse der SVM berechnet mit Hilfe des dualen Algorithmus mit quadratischem Kernel

Problem	`leu`	`colon`	`duke`
Zeit SV	9026s	4755s	11257s
Zeit Verif.	35,373s	8,40s	4,09s
Anzahl Trainingsdaten	38	30	38
Anzahl Testdaten	34	32	4
Anzahl SV	29	28	37
Korrekte Zuordnung	88,235%	84,375%	75%
Korrekte Zuordnung libsvm	88,235%	84,375%	75%
Abs. Fehler x	$9,89 \cdot 10^{-5}$	$6,93 \cdot 10^{-5}$	$2,20 \cdot 10^{-7}$
Rel. Fehler x	$5,26 \cdot 10^{-4}$	$5,44 \cdot 10^{-4}$	$7,59 \cdot 10^{-2}$
Abs. Fehler f	$3,00 \cdot 10^{-6}$	$2,03 \cdot 10^{-7}$	$2,46 \cdot 10^{-7}$
Rel. Fehler f	$7,60 \cdot 10^{-6}$	$6,77 \cdot 10^{-7}$	$3,52 \cdot 10^{-2}$
Abs. Fehler $\hat{b}$	$3,03 \cdot 10^{-5}$	$5,77 \cdot 10^{-5}$	0,234
Rel. Fehler $\hat{b}$	$6,79 \cdot 10^{-5}$	$1,91 \cdot 10^{-4}$	0,804
α	0,5	0,5	0,5
a	0,0001	0,01	1
b	100	1	1
Iterationszahl	62	57	74

Tabelle 7.8: Testergebnisse der SVM berechnet mit Hilfe des primalen Algorithmus mit quadratischem Kernel

Während der duale Algorithmus immer nach relativ kurzer Zeit (im Maximum nur knapp 12 min) beendet war, dauerte die Ausführung des primalen Algorithmus teils mehr als 3h! Zudem war bei Einsatz des quadratischen Kernels die Qualität der Ergebnisse wesentlich schlechter.

Bemerkenswert ist zudem, dass die Berechnung bei Einsatz des quadratischen Kernels bei Verwendung des dualen Algorithmus in zwei von drei Fällen mit einer Berechnungszeit von unter 200s nicht nur sehr schnell, sondern auch wesentlich schneller war als bei Einsatz des linearen Kernels!

Bemerkung: In `libsvm` ist die Translation b der berechneten Hyperebene mit dem umgekehrten Vorzeichen implementiert. Dies muss bei der Berechnung des Fehlers in der Translation b berücksichtigt werden.

Zusammenfassung und Ausblick

Die sicheren Implementierungen der quadratischen Optimierungsalgorithmen, die in dieser Arbeit vorgestellt wurden, zählen, wenn vielleicht auch nicht zu den kompliziertesten, so doch sicher zu den komplexesten bisher veröffentlichten Mehrparteienprotokollen. Für ihre Umsetzung war es nötig, grundlegende Methoden zum Rechnen mit Matrizen und Vektoren von Sharings wie auch andere Methoden zum sicheren Rechnen mit nicht-ganzen Zahlen völlig neu zu entwickeln. Dazu zählen vor allem die sichere Berechnung der Wurzelfunktion und das Lösen von linearen Gleichungssystemen bestehend aus geheimen Werten. Alle Protokolle zum Rechnen mit nicht-ganzen Zahlen beruhen auf der sicheren verteilten Fixpunktarithmetik von Catrina et al.. Es wurden zwei unterschiedlichartige quadratische Optimierungsalgorithmen implementiert. Einerseits handelte es sich dabei um eine primale Aktive Mengen Strategie, die, ausgehend von einem zulässigen Startwert, durch Anwendung der KKT-Gleichungen das Optimum bestimmt. Der Startwert wurde mit einer Varianten des sicheren Simplex-Algorithmus von Catrina berechnet. Weiterhin wurde der duale Algorithmus von Goldfarb und Idnani umgesetzt. Dieser erwies sich in praktischen Tests in der für eine Berechnung benötigten Zeit deutlich überlegen bei vergleichbarer oder besserer Qualität der Ergebnisse. Die beiden Algorithmen wurden verwendet für eine sichere Berechnung von Support Vektor Maschinen mit denen eine datenschutzfreundliche Klassifikation in horizontal partitionierten Datenbanken durchgeführt werden kann. Als konkretes Beispiel wurde ein finanzinstitutübergreifendes, datenschutzfreundliches Credit-Scoring konzeptioniert. Die versuchsweise Durchführung einer datenschutzfreundlichen Klassifikation gelang gut, insbesondere bei Verwendung des dualen Algorithmus.

Als vielversprechendste Fortsetzung der vorliegenden Arbeit erscheint die Weiterarbeit an der Entwicklung der sicheren Implementierungen der Support Vektor Maschinen. Um Probleme, die wesentlich mehr Datenpunkte als die hier behandelten besitzen, lösen zu können, zu denen also auch eine entsprechend größere Kernelmatrix gehört, die u.U. nicht mehr in den Arbeitsspeicher passt, müssen weitere Algorithmen sicher implementiert werden. Dazu zählen die in Kapitel 6 angesprochenen, mit denen es möglich ist, immer nur ein kleines Teilproblem beruhend auf einem 2×2-Ausschnitt der Kernelmatrix zu lösen.

Für die weitergehende Forschung interessant sind insbesondere auch die verwendeten Techniken, hauptsächlich das sichere Lösen linearer Gleichungssysteme und andere Techniken der Matrizen- und Vektorrechnung. Diese sind elementarer Bestandteil vieler anderer numerischer Probleme wie der kurz angeschnittenen Sequentiellen Quadratischen Programmierung oder der numerischen Nullstellenbestimmung. Es erscheint naheliegend, dass sich diese mit den hier verwendeten und ähnlichen Methoden sicher implementieren lassen.

Literaturverzeichnis

[ACS02] Algesheimer, Joy, Jan Camenisch und Victor Shoup: *Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products*. In: Yung, Moti (Herausgeber): *Advances in Cryptology – CRYPTO 2002*, Band 2442 der Reihe *Lecture Notes in Computer Science*, Seiten 417–432. Springer Berlin / Heidelberg, 2002.

[AF10] Atallah, Mikhail J. und Keith B. Frikken: *Securely outsourcing linear algebra computations*. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, Seiten 48–59, New York, NY, USA, 2010. ACM.

[AJW11] Asharov, Gilad, Abhishek Jain und Daniel Wichs: *Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE*. Cryptology ePrint Archive, Report 2011/613, 2011.

[AM08] Auria, Laura und Rouslan A. Moro: *Support Vector Machines (SVM) as a Technique for Solvency Analysis*. Discussion Papers of DIW Berlin 811, DIW Berlin, German Institute for Economic Research, August 2008.

[And07] Anderson, Raymond: *The Credit Scoring Toolkit: Theory and Practice for Retail Credit Risk Management and Decision Automation*. Oxford University Press, 2007.

[Bak07] Bakdi, Idir: *Benutzerauthetifizierung anhand des Tippverhaltens bei Verwendung fester Eingabetexte*. Universitätsverlag Regensburg, 2007.

[BC09] Bellotti, Tony und Jonathan Crook: *Support vector machines for credit scoring and discovery of significant features*. Expert Systems with Applications, 36(2, Part 2):3302 – 3308, 2009.

[BCD+09] Bogetoft, Peter, Dan Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøgaard, Janus Nielsen, Jesper Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach und Tomas Toft: *Secure Multiparty Computation*

Goes Live. In: Dingledine, Roger und Philippe Golle (Herausgeber): *Financial Cryptography and Data Security*, Band 5628 der Reihe *Lecture Notes in Computer Science*, Seiten 325–343. Springer Berlin / Heidelberg, 2009.

[BF01] Boneh, Dan und Matthew Franklin: *Efficient generation of shared RSA keys*. Journal of the ACM, 48:702–722, July 2001.

[BGV92] Boser, Bernhard E., Isabelle M. Guyon und Vladimir N. Vapnik: *A training algorithm for optimal margin classifiers*. In: *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, Seiten 144–152, New York, NY, USA, 1992. ACM.

[Bla11] Blanton, Marina: *Achieving Full Security in Privacy-Preserving Data Mining*. IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT'11), 2011.

[BMR90] Beaver, Donald, Silvio Micali und Philip Rogaway: *The round complexity of secure protocols*. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, Seiten 503–513, New York, NY, USA, 1990. ACM.

[BOGW88] Ben-Or, Michael, Shafi Goldwasser und Avi Wigderson: *Completeness theorems for non-cryptographic fault-tolerant distributed computation*. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, Seiten 1–10, New York, NY, USA, 1988. ACM.

[Can00] Canetti, Ran: *Security and Composition of Multiparty Cryptographic Protocols*. Journal of Cryptology, 13(1):143–202, 2000.

[Can01] Canetti, Ran: *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. In: *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, Seiten 136–145. IEEE Computer Society, 2001.

[CC06] Chen, Hao und Ronald Cramer: *Algebraic Geometric Secret Sharing Schemes and Secure Multi-Party Computations over Small Fields*. In: Dwork, Cynthia (Herausgeber): *Advances in Cryptology - CRYPTO 2006*, Band 4117 der Reihe *Lecture Notes in Computer Science*, Seiten 521–536. Springer Berlin / Heidelberg, 2006.

[CCG+07] Chen, Hao, Ronald Cramer, Shafi Goldwasser, Robbert de Haan und Vinod Vaikuntanathan: *Secure Computation from Random Error Correcting Codes*. In: Naor, Moni (Herausgeber):

Advances in Cryptology – EUROCRYPT 2007, Band 4515 der Reihe *Lecture Notes in Computer Science*, Seiten 291–310. Springer Berlin / Heidelberg, 2007.

[CD01] Cramer, Ronald und Ivan Damgård: *Secure Distributed Linear Algebra in a Constant Number of Rounds*. In: Kilian, Joe (Herausgeber): *Advances in Cryptology - CRYPTO 2001*, Band 2139 der Reihe *Lecture Notes in Computer Science*, Seiten 119–136. Springer Berlin / Heidelberg, 2001.

[CDD+04] Canetti, Ran, Ivan Damgård, Stefan Dziembowski, Yuval Ishai und Tal Malkin: *Adaptive versus Non-Adaptive Security of Multi-Party Protocols*. Journal of Cryptology, 17(3):153–207, 2004.

[CDdH07] Cramer, Ronald, Ivan Damgård und Robbert de Haan: *Atomic Secure Multi-party Multiplication with Low Communication*. In: Naor, Moni (Herausgeber): *Advances in Cryptology – EUROCRYPT 2007*, Band 4515 der Reihe *Lecture Notes in Computer Science*, Seiten 329–346. Springer Berlin / Heidelberg, 2007.

[CdH10a] Catrina, Octavian und Sebastiaan de Hoogh: *Improved Primitives for Secure Multiparty Integer Computation*. In: Garay, Juan und Roberto De Prisco (Herausgeber): *Security and Cryptography for Networks*, Band 6280 der Reihe *Lecture Notes in Computer Science*, Seiten 182–199. Springer Berlin / Heidelberg, 2010.

[CdH10b] Catrina, Octavian und Sebastiaan de Hoogh: *Secure Multiparty Linear Programming Using Fixed-Point Arithmetic*. In: Gritzalis, Dimitris, Bart Preneel und Marianthi Theoharidou (Herausgeber): *Computer Security - ESORICS 2010*, Band 6345 der Reihe *Lecture Notes in Computer Science*, Seiten 134–150. Springer Berlin / Heidelberg, 2010.

[CDI05] Cramer, Ronald, Ivan Damgård und Yuval Ishai: *Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation*. In: Kilian, Joe (Herausgeber): *Theory of Cryptography*, Band 3378 der Reihe *Lecture Notes in Computer Science*, Seiten 342–362. Springer Berlin / Heidelberg, 2005.

[CDM00] Cramer, Ronald, Ivan Damgård und Ueli Maurer: *General Secure Multi-party Computation from any Linear Secret-Sharing Scheme*. In: Preneel, Bart (Herausgeber): *Advances in Cryptology – EUROCRYPT 2000*, Band 1807 der Reihe *Lecture Notes in Computer Science*, Seiten 316–334. Springer Berlin / Heidelberg, 2000.

[CFS05] Cramer, Ronald, Serge Fehr und Martijn Stam: *Black-Box Secret Sharing from Primitive Sets in Algebraic Number Fields*. In: Shoup, Victor (Herausgeber): *Advances in Cryptology – CRYPTO 2005*, Band 3621 der Reihe *Lecture Notes in Computer Science*, Seiten 344–360. Springer Berlin / Heidelberg, 2005.

[CL11] Chang, Chih-Chung und Chih-Jen Lin: *LIBSVM: A library for support vector machines*. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[CM07] Cherkassky, Vladimir S. und Filip M. Mulier: *Learning from Data: Concepts, Theory, and Methods*. The Wiley bicentennial-knowledge for generations. John Wiley & Sons, 2007.

[CS10] Catrina, Octavian und Amitabh Saxena: *Secure Computation with Fixed-Point Numbers*. In: Sion, Radu (Herausgeber): *Financial Cryptography and Data Security*, Band 6052 der Reihe *Lecture Notes in Computer Science*, Seiten 35–50. Springer Berlin / Heidelberg, 2010.

[CST00] Cristianini, Nello und John Shawe-Taylor: *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[DA01] Du, Wenliang und Mikhail J. Atallah: *Privacy-Preserving Cooperative Scientific Computations*. In: *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Seiten 273–282, 2001.

[Dan63] Dantzig, George B.: *Linear programming and extensions*. Princeton University Press, Princeton, New Jersey, 1963.

[Deu10] Deutsche Bundesbank: *How the Deutsche Bundesbank assesses the credit standing of enterprises in the context of the refinancing of German credit institutions*. Technischer Bericht, Deutsche Bundesbank, May 2010.

[DM10] Damgård, Ivan und Gert Mikkelsen: *Efficient, Robust and Constant-Round Distributed RSA Key Generation*. In: Micciancio, Daniele (Herausgeber): *Theory of Cryptography*, Band 5978 der Reihe *Lecture Notes in Computer Science*, Seiten 183–200. Springer Berlin / Heidelberg, 2010.

[DN07] Damgård, Ivan und Jesper Nielsen: *Scalable and Unconditionally Secure Multiparty Computation*. In: Menezes, Alfred (Herausgeber): *Advances in Cryptology - CRYPTO 2007*, Band 4622 der Reihe

Lecture Notes in Computer Science, Seiten 572–590. Springer Berlin / Heidelberg, 2007.

[EG85] El Gamal, Taher: *A public key cryptosystem and a signature scheme based on discrete logarithms*. In: *Proceedings of CRYPTO 84 on Advances in cryptology*, Seiten 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[EL04] Ercegovac, Milos D. und Tomas Lang: *Digital arithmetic*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2nd Auflage, 2004.

[FDH+11] Franz, Martin, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser und Heike Schröder: *Towards Secure Bioinformatics Services (Short Paper)*. In: *Financial Cryptography*, Seiten 276–283. Springer Berlin / Heidelberg, 2011.

[FK11] Franz, Martin und Stefan Katzenbeisser: *Processing encrypted floating point signals*. In: *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security*, MM&Sec '11, Seiten 103–108, New York, NY, USA, 2011. ACM.

[Fle87] Fletcher, Roger: *Practical methods of optimization; (2nd ed.)*. Wiley-Interscience, New York, NY, USA, 1987.

[FSW03] Fouque, Pierre-Alain, Jacques Stern und Geert-Jan Wackers: *CryptoComputing with Rationals*. In: Blaze, Matt (Herausgeber): *Financial Cryptography*, Band 2357 der Reihe *Lecture Notes in Computer Science*, Seiten 136–146. Springer Berlin / Heidelberg, 2003.

[Gen09] Gentry, Craig: *Fully homomorphic encryption using ideal lattices*. In: *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, Seiten 169–178, New York, NY, USA, 2009. ACM.

[Gen11] Gentry, Craig: *Fully Homomorphic Encryption without Bootstrapping*. Security, 111(111):1–12, 2011.

[GI83] Goldfarb, Donald und Ashok Idnani: *A numerically stable dual method for solving strictly convex quadratic programs*. Mathematical Programming, 27:1–33, 1983.

[GKM82] Graham, Susan L., Peter B. Kessler und Marshall K. McKusick: *gprof: a call graph execution profiler (with retrospective)*. In:

McKinley, Kathryn S. (Herausgeber): *Best of PLDI*, Seiten 49–57. ACM, 1982.

[GL96] Golub, Gene H. und Charles F. Van Loan: *Matrix Computations*. The Johns Hopkins University Press, 3rd Auflage, 1996.

[GMP] *GNU Multiple Precision Arithmetic Library*. http://gmplib.org.

[GMW81] Gill, Philip E., Walter Murray und Margaret H. Wright: *Practical optimization*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1981.

[GMW87] Goldreich, O., S. Micali und A. Wigderson: *How to play ANY mental game*. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, Seiten 218–229, New York, NY, USA, 1987. ACM.

[Gol64] Goldschmidt, Robert Elliott: *Applications of Division by Convergence*. Diplomarbeit, M.I.T., 1964.

[Gol72] Goldfarb, Donald: *Extension of Newton's method and simplex methods for solving quadratic programs*. In: Lootsma, F. A. (Herausgeber): *Numerical methods for nonlinear optimization*. Academic Press, 1972.

[GRR98] Gennaro, Rosario, Michael O. Rabin und Tal Rabin: *Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography*. In: *PODC*, Seiten 101–111, 1998.

[HMO02] Hochreiter, Sepp, Michael Mozer und Klaus Obermayer: *Coulomb Classifiers: Generalizing Support Vector Machines via an Analogy to Electrostatic Systems*. In: Becker, Suzanna, Sebastian Thrun und Klaus Obermayer (Herausgeber): *NIPS*, Seiten 545–552. MIT Press, 2002.

[HS52] Hestenes, Magnus R. und Eduard Stiefel: *Methods of Conjugate Gradients for Solving Linear Systems*. Journal of Research of the National Bureau of Standards, 49:409–436, Dezember 1952.

[JF06] Jakobsen, Thomas und Strange L. From: *Secure Multi-Party Computation on Integers*. Diplomarbeit, Universität Århus, 2006.

[Joa00] Joachims, Thorsten: *The Maximum-Margin Approach to Learning Text Classifiers - Methods, Theory and Algorithms*. Doktorarbeit, Universität Trier, 2000.

[Kor89] Korn, Ralf: *Über die Algorithmen von Gill und Murray und von Goldfarb und Idnani zur quadratischen Optimierung*. Diplomarbeit, Johannes Gutenberg Universität Mainz, 1989.

[Lie12] Liedel, Manuel: *Secure Distributed Computation of the Square Root and Applications*. In: Ryan, Mark, Ben Smyth und Guilin Wang (Herausgeber): *Information Security Practice and Experience*, Band 7232 der Reihe *Lecture Notes in Computer Science*, Seiten 277–288. Springer Berlin / Heidelberg, 2012.

[Lor09] Lory, Peter: *Secure Distributed Multiplication of Two Polynomially Shared Values: Enhancing the Efficiency of the Protocol*. Emerging Security Information, Systems, and Technologies, The International Conference on, 0:286–291, 2009.

[LW11] Lory, Peter und Jürgen Wenzl: *A Note on Secure Multiparty Multiplication*. Working Paper, Universität Regensburg, Regensburg, März 2011.

[Mar04] Markstein, Peter: *Software Division and Square Root using Goldschmidt's Algorithms*. In: *In 6th Conference on Real Numbers and Computers*, Seiten 146–157, 2004.

[MPI] *MPIR – Multiple Precision Integers and Rationals 2.5.1*. http://www.mpir.org/.

[MW93] More, Jorge J. und Stephen J. Wright: *Optimization Software Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993.

[NW99] Nocedal, Jorge und Stephen J. Wright: *Numerical Optimization*. Springer-Verlag, 1st Auflage, 1999.

[Pai99] Paillier, Pascal: *Public-key cryptosystems based on composite degree residuosity classes*. In: *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'99, Seiten 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[Pow85] Powell, Michael James David: *On the quadratic programming algorithm of Goldfarb and Idnani*. In: Cottle, Richard W. (Herausgeber): *Mathematical Programming Essays in Honor of George B. Dantzig Part II*, Band 25 der Reihe *Mathematical Programming Studies*, Seiten 46–61. Springer Berlin Heidelberg, 1985.

[PTVF07] Press, William H., Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery: *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 Auflage, 2007.

[REPHCJJ05] Rong-En, Fan, Chen Pai-Hsuen, Lin Chih-Jen und Thorsten Joachims: *Working Set Selection Using Second Order Information for Training Support Vector Machines.* Journal of Machine Learning Research, 6(12):1889 – 1918, 2005.

[Ros60] Rosen, J. Ben: *The gradient projection method for nonlinear programming. Part I. Linear constraints*. Journal of the Society for Industrial and Applied Mathematics, 8(1):181–217, 1960.

[Ros04] Rosenberg, Gabriel D.: *Enumeration of all extreme equilibria of bimatrix games with integer pivoting and improved degeneracy check*. CDAM Res. Rep. LSE-CDAM-2005-18, London School of Economics, 2004.

[RSA78] Rivest, Ronald Linn, Adi Shamir und Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, 21:120–126, 1978.

[SB02] Stoer, Josef und Ronald Bulirsch: *Introduction to numerical analysis*. Texts in applied mathematics. Springer, Berlin/Heidelberg, 2002.

[Sch87] Schittkowski, Klaus: *More test examples for nonlinear programming codes*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[Sch02] Schittkowski, Klaus: *NLPQLP: A New Fortran Implementation of a Sequential Quadratic Programming Algorithm for Parallel Computing*. Technischer Bericht, Universität Bayreuth, 2002.

[Sec08] SecureSCM: *Technical Report D9.1: Secure Computation Models and Frameworks*. Technischer Bericht, SecureSCM, Juli 2008.

[Sec09] SecureSCM: *Technical Report D9.2: Secuity Analysis*. Technischer Bericht, SecureSCM, Juli 2009.

[Sec10] SecureSCM: *SecureSCM. Protocol Description V2. Deliverable D3.2, EU FP7 Project Secure Supply Chain Management*. Technischer Bericht, SecureSCM, Januar 2010.

[SH81] Schittkowski, Klaus und Willi Hock: *Test Examples for Nonlinear Programming Codes*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1981.

[Sha79] Shamir, Adi: *How to share a secret*. Commun. ACM, 22:612–613, November 1979.

[SS03] Schebesch, Klaus B. und Ralf Stecking: *Support Vector Machines for Credit Scoring: Comparing to and Combining With Some Traditional Classification Methods*. In: Schader, Martin, Wolfgang Gaul und Maurizio Vichi (Herausgeber): *Between Data Science and Applied Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, Seiten 604–612. Springer Berlin Heidelberg, 2003.

[SS05] Schebesch, Klaus B. und Ralf Stecking: *Support vector machines for classifying and describing credit applicants: detecting typical and critical regions*. Journal of the Operational Research Society, 56(9):1082–1088, 2005.

[SS06] Schebesch, Klaus B. und Ralf Stecking: *Comparing and Selecting SVM-Kernels for Credit Scoring*. In: Spiliopoulou, Myra, Rudolf Kruse, Christian Borgelt, Andreas Nürnberger und Wolfgang Gaul (Herausgeber): *From Data and Information Analysis to Knowledge Engineering*, Studies in Classification, Data Analysis, and Knowledge Organization, Seiten 542–549. Springer Berlin Heidelberg, 2006.

[SV09] Smart, Nigel P. und Frederik Vercauteren: *Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes*. Cryptology ePrint Archive, Report 2009/571, 2009.

[TK91] Tamura, M. und Y. Kobayashi: *Application of sequential quadratic programming software program to an actual problem*. Mathematical Programming, 52:19–27, 1991.

[Tof07] Toft, Tomas: *Primitives and Applications for Multi-party Computation*. Doktorarbeit, University of Aarhus, 2007.

[Tof09] Toft, Tomas: *Solving Linear Programs Using Multiparty Computation*. In: Dingledine, Roger und Philippe Golle (Herausgeber): *Financial Cryptography and Data Security*, Band 5628 der Reihe *Lecture Notes in Computer Science*, Seiten 90–107. Springer Berlin / Heidelberg, 2009.

[vDGHV10] Dijk, Marten van, Craig Gentry, Shai Halevi und Vinod Vaikuntanathan: *Fully Homomorphic Encryption over the Integers*. In: Gilbert, Henri (Herausgeber): *Advances in Cryptology – EUROCRYPT 2010*, Band 6110 der Reihe *Lecture Notes in Computer Science*, Seiten 24–43. Springer Berlin / Heidelberg, 2010.

[VGBGVD03] Van Gestel, Tony, Bart Baesens, Joao Garcia und Peter Van Dijcke: *A support vector machine approach to credit scoring*. Methodology, 1(1):73–82, 2003.

[Yao82] Yao, Andrew C.: *Protocols for secure computations*. In: *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, Seiten 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

Abbildungsverzeichnis

Tabellenverzeichnis

A Experimentelle Daten

Größe	Zeit LR	Zeit QR	Fehler LR	Fehler QR
2	0,12096	0,15631	1,55E-18	7,39E-13
3	0,20012	0,27582	2,98E-18	6,87E-12
4	0,30182	0,41247	1,61E-18	7,17E-12
5	0,41585	0,56357	2,35E-18	1,04E-11
6	0,54084	0,70654	5,35E-18	5,79E-12
7	0,69713	0,88423	2,65E-18	6,13E-12
8	0,86925	1,06877	1,98E-18	5,53E-12
9	1,04675	1,27636	2,59E-18	1,44E-11
10	1,29872	1,48993	7,08E-18	1,95E-11
11	1,52861	1,76226	4,56E-18	1,22E-11
12	1,81365	2,01322	3,76E-18	1,02E-11
13	2,08233	2,29605	3,80E-18	1,17E-11
14	2,39133	2,61205	5,04E-18	1,35E-11
15	2,82126	2,97724	3,44E-18	1,35E-11
16	3,20643	3,31998	4,28E-18	2,26E-11
17	3,57380	3,75212	4,76E-18	2,30E-11
18	4,04372	4,18611	4,11E-18	1,62E-11
19	4,76185	4,66133	3,81E-18	1,68E-11
20	5,12334	5,14709	3,97E-18	2,00E-11
21	5,70528	5,67557	3,97E-18	1,99E-11
22	6,34372	6,28356	6,20E-18	2,32E-11
23	7,06214	6,89646	3,96E-18	1,65E-11
24	7,78521	7,53565	4,28E-18	2,06E-11
25	8,65652	8,23093	4,81E-18	2,46E-11
26	10,39932	8,95314	4,94E-18	2,31E-11
27	10,35318	9,7642	4,74E-18	2,33E-11
28	11,30767	10,66004	5,23E-18	3,01E-11
29	12,37197	11,47443	5,33E-18	2,85E-11
30	13,47351	12,5193	5,93E-18	2,83E-11
31	14,63791	13,45407	4,93E-18	2,32E-11
32	15,99657	14,55316	4,71E-18	2,10E-11
33	17,22405	15,54790	5,18E-18	2,32E-11
34	18,77495	16,77509	4,95E-18	1,96E-11
35	20,15287	18,16674	5,18E-18	2,22E-11
36	21,64008	19,30721	4,77E-18	2,69E-11
37	23,38021	20,70200	5,20E-18	2,42E-11
38	24,92672	22,03828	4,87E-18	2,57E-11
39	27,04265	23,76684	5,08E-18	2,36E-11
40	28,57228	24,99961	5,40E-18	2,76E-11

Tabelle A.1: Rechenzeit und Fehler beim Lösen von linearen Gleichungssystemen der Größen 2-40 mit Hilfe der LR- und der QR-Zerlegung

B Prototypische Implementierung

In diesem Kapitel soll ein Überblick über die Details der prototypischen Implementierung gegeben werden. Zunächst wird die entwickelte Programmbibliothek beschrieben (Abschnitt B.1), auf deren Grundlage die sicheren, verteilten Optimierungsalgorithmen entwickelt wurden. Sie besteht aus den Implementierungen der sicheren verteilten Fixpunktarithmetik ([CS10][CdH10a][CdH10b]) und den beschriebenen Erweiterungen, insbesondere der sicheren Berechnung der Wurzel und der Protokolle zur sicheren Matrizen- und Vektorrechnung. In einem weiteren Abschnitt (Abschnitt B.2) wird auf die für die quadratischen Optimierungsprobleme und die zur Berechnung der Support Vektor Maschinen benötigten Methoden eingegangen.

B.1 Grundlegende Bausteine

Allgemeines

In der Implementierung werden alle Parteien, deren Anzahl vom Programm nicht beschränkt ist, auf einem Rechner simuliert. Die einzelnen Spieler werden dabei nicht als eigenständige Objekte modelliert, sondern jedes Sharing wird intern als Vektor dargestellt.

Da alle Komponenten eines Sharings eine große Länge - 1024 Bit ([CdH10b]) - besitzen, können sie nicht mit den internen Datentypen dargestellt werden. Zur Darstellung verwendet wird in der vorliegenden C++-Implementierung eine gepatchte MPIR 2.5.1 (Die Windows Version der GNU Multiple Precision Public Library). Diese wird zudem auch für andere Dinge, insbesondere die Erzeugung von Zufallszahlen, benötigt.

Die Klassen `Vektor`, `Share` und `Matrix`

Die eigentlichen Protokolle für sichere Mehrparteienberechnungen an einzelnen Sharings befinden sich in der von der Klasse `Vektor` (Anhang C.1) abgeleiteten Klasse `Share` (Anhang C.3). Eine Instanz von `Vektor` beinhaltet einen `mpz_t`-Vektor, in dem die den einzelnen Spielern P_i, $i = 1, \ldots, n$ zuzuordnenden Werte x_i in den Komponenten $i = 1, \ldots, n$ gespeichert sind (Die Komponente 0 ist nicht besetzt). Weiter sind in `Vektor` Implementierungen elemen-

tarer arithmetischer Operationen auf dem Vektor sowie dessen Besetzung mit zufälligen Werten enthalten.

Die Klasse `Share` beinhaltet eine Instanz der Klasse `Vektor` und alle zur Implementierung der sicheren verteilten Fixpunktarithmetik notwendigen Methoden sowie einige weitere (z.B. Ausgabe- und Debug-Funktionen). Darunter sind grundlegende Routinen wie `Erzeuge`, das aus einem übergebenen Wert ein ihn repräsentierendes Sharing erstellt, `Rekonstrukt`, das den Wert, den ein Sharing repräsentiert, rekonstruiert, aber auch alle vier Grundrechenarten sowie die Quadratwurzel, Vergleiche, die sichere verteile Erzeugung von Zufallsbits und einige weitere, insbesondere Hilfsprotokolle. Hervorzuheben sind in diesem Zusammenhang die von Peter Lory entwickelten und von Jürgen Wenzl programmierten Multiplikationsprotokolle ([Lor09][LW11]), die - zusammen mit `Mul_GRR` - als einzige Bestandteile der Bibliothek nicht selbst entwickelt wurden und essentieller Teil (fast) aller anderen Protokolle sind. Eine sechsstellige Anzahl von Aufrufen des Multiplikationsprotokolls auch in den - vergleichsweise kleinen Testbeispielen - ist nicht ungewöhnlich. Standardmäßig ist die Multiplikation als `MUL_Lory3` implementiert, aber die anderen Routinen `MUL_Lory2` und `Mul_GRR`, können, wenn dies auf Grund der Parameter ([Lor09][LW11]) sinnvoll erscheint, ohne größere Schwierigkeiten verwendet werden. An Variablen enthält `Share` nur eine Instanz von `Vektor` sowie die Spielerzahl n (entspricht der Vektorlänge), den Grad $t \leq \frac{n-1}{2}$ des zum Sharing verwendeten Polynoms sowie den Modulus `Modulus`, der den endlichen Körper, der für das betrachtete Sharing maßgeblich ist, definiert. Um die Klasse `Share`, von der im Programmablauf sehr viele Instanzen benötigt werden, handlich zu halten, wurden weder die Bitlänge k der Fixpunktzahlen noch die Anzahl der Nachkommastellen $f \leq k$ `Share` als Variablen hinzugefügt (f ist als globale Variable implementiert). An den Stellen, an denen diese Werte dann explizit benötigt werden, z.B. beim Abschneiden von Nachkommastellen nach sicheren Multiplikationen, werden sie den entsprechenden Routinen (z.B. `TruncPr`) explizit als Variable übergeben. Die Klasse `Matrix` (Anhang C.2) ist eine kleine Hilfsklasse, bestehend aus einer mpz_t-Matrix sowie Variablen, die deren Höhe und Breite angeben. Sie wird für das Multiplikationsprotokoll benötigt.

Die Klassen `Matrix_LA` und `Vektor_LA`

Auf der Klasse Share aufbauend existieren die Klassen `Vektor_LA` und `Matrix_LA` (Anhang C.4 und C.5).[1] Als Objekte enthalten sie Vektoren bzw. Matrizen von Shares, die jeweilige Größe, die Eigenschaften (Spielerzahl, Polynomgrad, Modulus) der unterliegenden Shares, die Fixpunktlänge k, die

[1] LA=Lineare Algebra

Anzahl der Nachkommastellen N (entspricht i.d.R. dem globalen f) sowie den kleinen Modulus q_1, der für die Erzeugung zufälliger Bits benötigt wird. `Vektor_LA` ist mit Hilfe der STL-Routine `std::<vector>` implementiert, während `Matrix_LA` die Matrix in Form eines dynamisch erzeugten Arrays darstellt. In der Klasse `Vektor_LA` kann auf Elemente der Klasse `Matrix_LA` zugegriffen werden, aber nicht umgekehrt (dies erzeugt Kompilier-Konflikte). Aus diesem Grund können Vektoren in der Klasse `Matrix_LA` nur in Form des enthaltenen STL-Vektors übergeben werden. Da die sonstigen Variablen sich auf die unterliegenden Shares beziehen und diese bei Vektor und Matrix identisch sind (außer es handelt sich bei der Matrix z.B. um eine Permutationsmatrix, für die wichtige Parameter wie Spielerzahl und Polynomgrad identisch und weniger wichtige wie k oder f irrelevant sind), stellt dies kein Problem dar. `Vektor_LA` und `Matrix_LA` enthalten die mit Vektoren und Matrizen möglichen arithmetischen Operationen wie Additionen oder Multiplikation mit einem Skalar. Enthalten sind außerdem die Methoden, die zur Implementierung aller vier beschriebenen Varianten zur Lösung linearer Gleichungssysteme benötigt werden sowie die zugehörigen Hilfsroutinen wie z.B. `house_neu`, das aus einem beliebigen Vektor (der $\neq 0$ vorausgesetzt wird) den zugehörigen Householder-Vektor erstellt, außerdem Ausgabe- und Debug-Funktionen.

Bei Operationen, für die Elemente mehrerer Klassen nötig sind, wurden die entsprechenden Methoden i.d.R. der Klasse zugeordnet, auf der die Methode operiert. So liegen z.B. sowohl Matrix-Vektor-Multiplikation (`MatrixMult`) als auch Vorwärts- und Rückwärtssubstitution (`VWSubs` und `RWSubs`) in der Klasse `Vektor_LA` wie auch das Skalarprodukt (`SkalarMult`).

Weitere Klassen sind `GF256` (Anhang C.11), das das Rechnen im AES-Körper $\mathbb{F}_{2^8}$ ermöglicht sowie die Klasse `Share_F2M` (Anhang C.7), die Sharings darüber ermöglicht. `Share_F2M` wird für Sub-Routinen der Klasse `Share`, z.B. für die Bitzerlegung und andere Methoden, insbesondere solche, die zur Berechnung der Division und der Wurzel nötig sind, benötigt. Beachte, dass das in `Share_F2M` verwendete Multiplikationsprotokoll im Prinzip das von [GRR98] ist, da sich auf Grund des verwendeten Körpers die Verbesserungen aus [Lor09] und [LW11] nur teilweise anwendbar sind (ein Multiplikationsprotokoll, das die Multiplikation von Sharings mit Hilfe der dividierten Differenzen durchführt, ist trotzdem implementiert, wird aus Effiziengünden standardmäßig aber nicht verwendet).

Die Klasse `Simplex_Startwert_Catrina`

Eine von `Matrix_LA` abgeleitete Klasse, ist die Klasse `Simplex_Startwert_Catrina` (Anhang C.6). Sie besitzt - zusätzlich zu einer Instanz der Klasse `Matrix_LA` - noch deren deskriptive Variablen (Breite, Höhe, Modulus, k, N,

q_1) als eigenständige Variable. Die enthaltene Instanz von `Matrix_LA` beinhaltet das Simplex-Tableau. In ihr ist der in Abschnitt 5.5.3 beschriebene Phase I-Algorithmus des primalen quadratischen Optimierungsalgorithmus implementiert.

Die Klasse `SVM`

Die Klasse `SVM` (Anhang C.8) enthält Methoden und Objekte, die zum Training von Support Vektor Maschinen bzw. der Zuordnung von Testdaten zu Klassen benötigt werden. Insbesondere sind dies eine Matrix `Daten`, die die zu klassifizierenden Daten speichert, die Kernel-Matrix `K`, die Matrix `G`, die Teil des quadratischen Optimierungsproblems ist, ein Vektor `Y`, der die Klassenzugehörigkeit der Testdaten anzeigt, ein Vektor `Alpha`, in dem nach erfolgtem Training der Lösungsvektor gespeichert wird, ein Sharing `b`, in dem die Translation der berechneten Hyperebene gespeichert ist, ein Vektor `SV`, der binär kodiert, ob ein Datensatz Support Vektor ist sowie ein Share `Lambda`, in dem der Soft-Margin-Toleranzparameter (vgl. (6.33)) gespeichert ist. Die Methoden der Klasse `SVM` sind die Berechnung der Kernel-Matrix (linear und quadratisch), die Berechnung der Matrix `G`, die Entscheidungsregel, die Berechnung des Translationsparameters `b` sowie verschiedene Debug-Funktionen. Das eigentliche Verfahren, mit dem die Berechnung der Hyperbene durchgeführt wird, ist Teil der (jeweiligen; für beide Methoden gibt es jeweils eine) `main`-Funktion.

Die Klasse `Vektor_mpf_t` und `Matrix_mpf_t`

Die Klassen `Vektor_mpf_t` (Anhang C.9) und `Matrix_mpf_t` (Anhang C.10) dienen zur Kontrolle der implementierten Mehrparteienalgorithmen. In ihnen sind nicht-verteilte Algorithmen (und jeweils ein Vektor bzw. eine Matrix) enthalten, die auf Vektoren bzw. Matrizen operieren, mit denen die Genauigkeit der Mehrparteienalgorithmen gemessen werden kann. Dazu zählt v.a. das Lösen linearer Gleichungssysteme. Es ist hier als der Gaußalgorithmus implementiert.

Funktionsweise

Eine explizite Eingabe der Parameter (wie Spielerzahl, Polynomgrad, Fixpunktlänge, etc.) ist nicht vorgesehen. Diese Werte müssen im Programm selbst geändert werden.

Öffentlich bekannte Konstante wie die Lagrange-Interpolationskoeffizienten, die z.B. beim Öffnen von Shares benötigt werden, die Tabellen mit deren Hilfe die Multiplikation in $\mathbb{F}_{2^8}$ implementiert wird oder die Koeffizienten, die

zur Umwandlung von Sharings über $\mathbb{F}_{q_1}$ nach $\mathbb{F}_q$ bzw. $\mathbb{F}_{2^8}$ nach $\mathbb{F}_q$ benötigt werden, werden zu Beginn des Programms als globale Variable berechnet und können dann von jeder Instanz, die sie benötigt, ohne weitere Berechnung verwendet werden. Andere globale Variablen sind z.B. die Koeffizienten der linearen Approximationsfunktionen zur sicheren Berechnung der Division oder der Wurzel.

Zusätzlich wird zu Programmbeginn eine Serie von Zufallszahlen initialisiert, die dann zur Laufzeit kostengünstig erzeugt werden können. Dies beschleunigt die Ausführung der Programme in beträchtlichem Maße. Zwar können auf diese Weise Wiederholungen der Zufallszahlen auftreten, jedoch würde die Initialisierung und Erzeugung von Zufallszahlen zur Laufzeit mehr als die Hälfte der Rechenzeit einnehmen! Verlässliche Aussagen über die eigentliche Rechenzeit sowie Laufzeitvergleiche wären dann nicht mehr möglich. Zudem wäre bei einer echten, sicheren Mehrparteienberechnung, die auf mehreren Rechnern stattfindet, die Erzeugung von Zufallszahlen (z.B. im Rahmen des Multiplikationsprotokolls) Sache der Teilnehmer. Diese können die Zufallszahlen regelmäßig neu initialisieren und so Wiederholungen vermeiden. Aufgrund der Latenz des Netzwerks und der Möglichkeit, Zufallszahlen parallel zu anderen Prozessen zu initialisieren, ist dies ohne Beeinträchtigung der Rechenzeit möglich.

B.2 Spezifische Bausteine

In allen Algorithmen wurden die verwendeten Testbeispiele/-daten in `#ifdefine/#endif` Umgebungen im main-Dokument eingefügt. Die Auswahl eines anderen Satzes von Testbeispielen/-daten ist so problemlos möglich. Die Umsetzung der Protokolle entspricht nicht an allen Stellen der Darstellung in den vorangegangenen Kapiteln. Oft sind bei den quadratischen Optimierungsalgorithmen Teilprotokolle aus Gründen der Einfachheit und der Effizienz direkter Teil des main-Programms. Dazu zählen z.B. die Funktionen, die in der sicheren Implementierung des dualen Algorithmus zum Aktualisieren der Vektoren $[\overrightarrow{x}]$ und $[\overrightarrow{u}]$ sowie des Funktionswerts $[f]$ benötigt werden. Sie sind aber im Programm klar durch Kommentare abgegrenzt.

Kompilierhinweise finden sich in der README.txt, die auf der beigefügten CD enthalten ist. Alle Ausgaben der Programme werden auch in die Datei Ausgabe.txt geschrieben. Diese wird, so nicht bereits vorhanden, automatisch erzeugt.

Die primale Aktive Mengen Strategie

Der Startwert

Die Bestimmung des Startwerts nach der in Abschnitt 5.5.3 beschriebenen Methode geschieht in der eigens dafür entworfenen Klasse `Simplex_Startwert_Catrina`. Diese ist von der Klasse `Matrix_LA` abgeleitet und enthält als wichtigste Variable das Simplex-Tableau und als Routinen die in [CdH10b] beschriebenen bzw. die in Abschnitt 5.5.3 adaptierten Methoden. Der Testdatensatz muss zu Beginn mittels `#define` ausgewählt werden.

Der Algorithmus

Der Algorithmus findet zum großen Teil im entsprechenden `main`-Teil statt. Die Erstellung des in jedem Iterationsschritt zu lösenden Gleichungssystems, die Berechnung der Schrittweite α und die Auswertung der Zielfunktion wurden jedoch in eigene Methoden ausgelagert. Zu Beginn wird, wenn der Nullpunkt nicht zulässig ist, das Phase-I Problem innerhalb von einer Instanz von `Simplex_Startwert_Catrina` gelöst.

Der Algorithmus von Goldfarb/Idnani

Auch der Algorithmus von Goldfarb/Idnani wurde großenteils im `main`-Teil implementiert. Ausgenommen davon ist die Berechnung von z und r, die Aktualisierung von J sowie die Auswahl der verletzten Nebenbedingung, die - wie in Abschnitt 5.7.4.1 beschrieben - unterschiedlich gestaltet werden kann. Andere Protokolle wie `fNeu`, `uNeu`, `xNeu` sowie die Unterteilung des Algorithmus in die Schritte 0,1, und 2 wurden innerhalb der main-Funktion implementiert, jedoch durch Kommentare gekennzeichnet. Der Testdatensatz muss zu Beginn mittels `#define` ausgewählt werden.

Berechnung von Support Vektor Maschinen

Im `main`-Teil der Implementierung des Algorithmus zur sicheren Berechnung von Support Vektor Maschinen (bei beiden Algorithmen) beginnt mit einem Abschnitt zum Einlesen der Daten. Im Anschluss daran wird die trennende Hyperebene berechnet und die Testdaten klassifiziert. Der Testdatensatz und der Kernel (linear oder quadratisch) müssen mittels `#define` vor Programmbeginn bestimmt werden. Die Parameter (für den quadratischen Kernel, falls dieser gewählt wird) werden dadurch mitdefiniert. Es ist darauf zu achten, dass sich die Datei mit den Testdaten im selben Verzeichnis befindet wie die `main`-Funktion.

C Übersicht über die Methoden der Implementierung

C.1 Die Klasse `Vektor`

```
class Vektor
{
    public:
        //Member-Variablen
        mpz_t * Vector;
        int Laenge;

        //Konstruktoren
        Vektor(int n);
        Vektor();

        //Destruktor
        virtual ~Vektor();

        //Copy-Konstruktor
        Vektor(const Vektor& Ursprungsvektor);

        //Operatoren
        Vektor& operator=(const Vektor& Ursprungsvektor);

        //Ausgabefunktionen
        virtual void print(char * name, int n);

        //Arithmetische Funktionen
        virtual void Plus_Vektor(mpz_t * Summand2, int n, mpz_t& q); //
            Addition
        virtual void minus(mpz_t * Summand2, int n, mpz_t& q); //
            Subtraktion
        virtual void Mul_Konst(mpz_t& Konstante, int n, mpz_t& q); //
            Multiplikation mit mpz-Zahl
        virtual void Mul_Konst_int(int c, int n, mpz_t& q); //
            Multiplikation mit int-Zahl
        virtual void Add_Konst(mpz_t& Konstante, int n, mpz_t& q); //
            Addition einer mpz-Zahl
        virtual void Minus_Konst(mpz_t& Konstante, int n, mpz_t& q); //
            Subtraktion einer mpz-Zahl

        //Sonstige Funktionen
        virtual void Zufall(int n, int t,mpz_t& p); //Vektor mit
            zufälligen Elementen besetzen

    protected:
    private:
};
```

C.2 Die Klasse `Matrix`

```
class Matrix
{
    public:

        mpz_t ** M;

        Matrix(int n, int m);
        void ~Matrix();
        Matrix(const Matrix& Ursprungsmatrix);
        Matrix& operator=(const Matrix& Ursprungsmatrix);

    protected:
    private:

        int Laenge;
        int Breite;
};
```

C.3 Die Klasse `Share`

```
class Share: public Vektor
{
public:
    //Member-Variablen
    int Anzahl_Spieler;
    int Schwelle;
    mpz_t Modulus;

    ///Konstruktor
    Share(int n, int t, mpz_t& q);

    ///Default-Konstruktor
    Share();

    ///Copy-Konstruktor
    Share (const Share& Ursprungsshare);

    ///Destruktor
    ~Share();

    ///Operatoren
    Share& operator=(const Share& Ursprungsshare);
    Share operator+(const Share& A);
    Share operator+(int c);
    Share operator+(mpz_t& c);
    Share& operator+=(const Share& A);
    Share& operator+=(int c);
    Share& operator+=(mpz_t& c);
    Share operator-(const Share& A);
    Share operator-();
    Share& operator-=(const Share& A);
    Share& operator-=(mpz_t& Subtrahend);
    Share operator-(int c);
    Share operator-(mpz_t& c);
```

```
Share& operator-=(int c);
Share operator*(const Share& A);
Share& operator*=(const Share& A);
Share& operator*=(mpz_t& c);
Share& operator*=(int c);
Share operator*(mpz_t& a);
Share operator*(int n);
Share& operator=(int n);//Setze ein Sharing konstant auf n
Share& operator=(mpz_t n);//Setze ein Sharing konstant auf n

///Aithmetische Operationen
void Quadrat();
void Betrag(mpz_t& q_1, int k);
void Signum(int t, mpz_t& q, mpz_t&q_1, int k);
void Mul(mpz_t *b, int t, mpz_t& q); //Multiplikation zweier
     verteilter Werte; Multiplikationsprotokoll von Peter mit beiden
     Verbesserungen; Ergebnis der Multiplikation wird in den Share-
     Vektor des Scopes geschrieben
void Mul_GRR(mpz_t * b, int t, mpz_t& q);//GRR-Multiplikation zum
     Vergleich; Das Ergebnis muß nach der Berechnung rekonstruiert
     werden
void MUL_Lory3(mpz_t *b, int t, mpz_t& q);//Multiplikation mit
     rekursiv berechneten Lagrange-Koeffizienten
void MulPub(Share& b, mpz_t& Ausgabe);//Multiplikation mit
     anschließender Ausgabe
void FPDiv(const Share& Nenner, int t, mpz_t& q, mpz_t& q_1, int k);//
     Division; (k,f)-Fixpunktzahlen werden vorausgesetzt
void DivNR(Share& Nenner, mpz_t& q_1, int k);//Division nach Newton-
     Raphson
void Inverses(mpz_t& q_1, int k);//1/*this
void Inverses_ohne_Trunc(mpz_t& q_1, int k);//Muß nach weiterer
     Multiplikation TruncPr(3k-f,2k-f) ausführen!
void Wurzel(int k, mpz_t& q, mpz_t& q_1);//Berechnung der Wurzel nach
     Goldschmidt
void Div_Konst(mpz_t& d, int t, mpz_t& q, int k, mpz_t& q_1);//
     Division durch eine Konstante
void Div_Konst_int(int d, mpz_t& q, mpz_t& q_1, int k);
void Invertiere();//Invertiere den Share mod q
void Mul_Konst(mpz_t c, mpz_t q); //Multiplikation mit Konstanter
void Mul_Konst_int(int c, mpz_t q);
void Add_Konst(mpz_t& Konstante, mpz_t& q);//Addition von Konstanter
void Add_Konst_int(int c, mpz_t& q);

///Erzeugung, Rekonstruktion und Ausgabe von Shares
void Erzeugung(mpz_t& s, int t, mpz_t& q);//Konstruktion eines
     Sharings
void Erzeugung_double(double d, int t, mpz_t& q);//Erzeugung auf
     Grundlage einer double
void Rekonstrukt(mpz_t& s, int t, mpz_t& q);//Rekonstruktion eines
     verteilten Werts
void Rekonstrukt_Ausgabe(std::string ShareName);//Rekonstruktion mit
     Ausgabe; die Skalierung der Fixpunktarithmetik wird
     herausgerechnet
void Rekonstrukt_Ausgabe(std::string ShareName, int n);
void Rekonstrukt_GRR(mpz_t& s, mpz_t& q, int t);//Rekonstruktion mit
     Hilfe der Lagrange-Interpolation
void Rekonstrukt_GRR_mit_VZ(mpz_t& s, mpz_t& q, int t);
```

```
void Rekonstrukt_Ausgabe_skaliert_hexadezimal(std::string ShareName);
    //Rekonstruktion mit Ausgabe; die Skalierung der
    Fixpunktarithmetik wird herausgerechnet
void Rekonstrukt_Ausgabe_unskaliert(std::string ShareName);//
    Rekonstruktion mit unskalierter Ausgabe - geeignet für Bit-
    Ausgaben
void Rekonstrukt_Ausgabe_unskaliert(std::string ShareName, int n);
void Rekonstrukt_Ausgabe_unskaliert(std::string ShareName, mpz_t& p);
    //Primzahl mitgeben
void Rekonstrukt_Spieler_beliebig(mpz_t& s, int * Spielerliste, int t,
     mpz_t& q);//Rekonstruktion an der Stelle 0 unter Verwendung
    beliebiger Spieler
void Rekonstrukt_Spieler_beliebig_Stelle_beliebig(mpz_t& s, int *
    Spielerliste, int t, int r, mpz_t& q);//Rekonstruktion eines
    beliebigen Werts eines Spielers
int Ausgabe_skaliert_Datei(std::string ShareName, FILE * datei);//
    Ausgabe in Datei; diese muss vorher geöffnet werden

///Erzeugung von Zahlen und Zufallszahlen
void Zufallszahl(int t, mpz_t& q);//Erzeugen einer verteilten
    Zufallszahl
void PRandInt(int t, int Laenge, mpz_t& q);// vgl. "Secure Computation
     with Fixed-Point Numbers"
void PRandBitD(Share_F2M& F2M_Bit, int t, mpz_t& q, mpz_t& q_1);//vgl.
     SecureSCM-D.9.2
void RandBit(int t, mpz_t& q);//Erzeugung eines zufälligen, verteilten
     Bits in Z_q
void gemeinsame_1(mpz_t& q);//Sharing der 1
void gemeinsame_0(mpz_t& q);//Sharing der 0

///Hilfsfunktionen für Division, Wurzel und Multiplikation
void AppRcr(int k, int t, mpz_t& q, mpz_t& q_1);//Approximation von 1/
    b durch 2.9142-2*c; c ist auf (0.5,1) normalisiertes b
void Norm(Share& v, int k, int t, mpz_t& q, mpz_t& q_1);//Normiere
    Werte auf Intervall (0.5,1)
void Norm_Find_First(Share& v, int k, int t, mpz_t& q, mpz_t& q_1);//
    Variante von Norm mit Protokoll FindFirst
void Norm_mit_Wurzel(Share& v, Share& SQ, Share& Wert, int k, int t,
    mpz_t& q, mpz_t& q_1);//Hilfsprogramm zur Wurzelziehung;
    berechnet zusätzlich noch Wurzel(2^m) bzw. Wurzel(2^{m-1})
void RecItNR(int k, mpz_t& q_1);//Hilfsprotokoll zu DivNR;
void RandNumbers(mpz_t **randNumbers, int t, mpz_t& q, mpz_t *a, mpz_t
     *b);//Hilfsfunktion zum Erzeugen von zufälligen Funktionswerten
    bei der Neuverteilung bei der Multiplikation

///Umwandlung von Sharings
void Umwandlung(int t, mpz_t& q, mpz_t& q_1);//Umwandlung von shares
    modulo q_1 in Shares modulo q mittels RISS
void Umwandlung_Bit_Zq_F2M(Share_F2M& r_F2M, Share * RISS_q, mpz_t **
    RKq, int t, mpz_t& q, mpz_t& z);//Umwandung eines Zufallsbits von
     Z_q nach F_2m
void F2MtoZq(mpz_t& q, mpz_t& q_1, Share_F2M& Bit, int t);//Umwandlung
     des Shares im Scope in einen Share A in Z/qZ

///Vergleiche
void BitLT(int t, int k, mpz_t& Zahl_1, std::vector<Share>& Bits,
    mpz_t& q);//Vergleich der Zahl im Scope mit einer bitweise
```

```
    übergebenen
void BitLT_F28(int t, int k, mpz_t& Zahl_1, std::vector<Share>& Bits,
    mpz_t& q_1);//Wie BitLT, aber elementare Operationen werden in
    F2M durchgeführt
void LTZ(int t, mpz_t& q, mpz_t& q_1, int k);//<?0
void LTZ_F28(int t, mpz_t& q, mpz_t& q_1, int k);//LTZ unter
    Verwendung von BitLT_F28
void LT(Share& Argument_1, int t, mpz_t& q, mpz_t& q_1, int k);//(a<b)
    ? 1:0
void LT(int c, int t, mpz_t& q, mpz_t& q_1, int k);
void LT(mpz_t& Argument_1, int t, mpz_t& q, mpz_t& q_1, int k);//*this
    <?Argument_1 1:0
void LT_Konst_Betrag(mpz_t& Argument_1, int t, mpz_t& q, mpz_t& q_1,
    int k);//Vergleich, ob Quadrat kleiner als "Argument"
void GTZ(int t, mpz_t& q, mpz_t& q_1, int k);//wie LTZ
void GT(Share& Argument_1, int t, mpz_t& q, mpz_t& q_1, int k);//(a>b)
    ? 1:0
void GT(mpz_t& Argument_1, int t, mpz_t& q, mpz_t& q_1, int k);//(a>b)
    ? 1:0
void GTZ_approximativ(int t, mpz_t& q, mpz_t& q_1, int k);//Überprüfe,
     ob Wert größer als 2^{-f+10} ist
void LTZ_approximativ(int t, mpz_t& q, mpz_t& q_1, int k);//Überprüfe,
     ob Wert kleiner als 2^{-f+10} ist
void GQ(int t, mpz_t& q, mpz_t& q_1, int k);//(a>=0)? 1:0
void EQZ(int t, mpz_t& q, mpz_t& q_1, int k);//(a=0)? 1:0
int EQZPub();
void EQ(Share& Argument, int t, mpz_t& q, mpz_t& q_1, int k);//
    Vergleich mit Argument
void EQ_mpz(mpz_t& Argument, int t, mpz_t&q, mpz_t& q_1, int k);
void EQ_int(int Wert, int t, mpz_t& q, mpz_t& q_1, int k);//Vergleich
    mit "Wert"

///Logische Operatoren
void OR(Share& Argument);//OR von Bit-Shares
void AND(Share& Argument);//AND von Bit-Shares

void min(const Share& a, const Share& b, mpz_t& q, mpz_t& q_1, int t,
    int k, Share& Hilfe);//In "Hilfe" wird der Index des
    minimierenden Elements zurückgegeben: Hilfe=1<=>a<b; 0 sonst

///Rundungsfunktionen
void TruncPr(int k, int m, int t, mpz_t& q, mpz_t& q_1);//Abschneiden
    überzähliger Stellen nach Multiplikation
void Trunc(int t, mpz_t& q, mpz_t& q_1, int k, int m);//
    Deterministisches Abschneiden
void Trunc_F28(int t, mpz_t& q, mpz_t& q_1, int k, int m);//Trunc
    Function mit elementaren Operationen in F_{2^8}; benötigt für
    LTZ_F_{2^8}
void Mod2m(int k, int m, mpz_t& q, mpz_t& q_1, int t);//Berechne
    modulo 2^m
void Mod2m_F28(int k, int m, mpz_t& q, mpz_t& q_1, int t);
void Mod2(int k, mpz_t& q, mpz_t& q_1, int t);//Berechne gerade oder
    ungerade

///Hilfsfunktionen für Rundung
void CarryOutL_1offen(mpz_t& Argument_1, std::vector<Share>&
    Argument_2, int t, mpz_t& q, int k);//Berechne das höchste
```

```
        gemerkte Bit bei der Addition
    void CarryOutL_cin_1offen(mpz_t& Argument_1, std::vector<Share>&
        Argument_2, int t, mpz_t& q, int k, int carry_in);//Wie
        CarryOutL_1offen nur mit zusätzlichem Input-Bit
    void CarryOutL_cin_1offen_F2M(mpz_t& Argument_1, std::vector<Share_F2M
        >& Argument_2, int t, int k, int carry_in, mpz_t& q_1);//Wie
        CarryOutL_cin_1offen, aber mit Operationen in F2M
    void CarryOutAux(std::vector<Share>& d_1, std::vector<Share>& d_2, int
         k, int t, mpz_t& q);//Hilfsprogramm
    void CarryOutAux_F2M(std::vector<Share_F2M>& d_1, std::vector<
        Share_F2M>& d_2, int k, int t);//Wie CarryOutAux nur mit
        Operationen in F28

    ///Bitzerlegung
    void BitDec(std::vector<Share_F2M>& Bits, int k, int m, int t, mpz_t&
        q, mpz_t& q_1);//Bitzerlegung eines Sharings; die Bits werden ash
         Sharings über F_{2^8} gespeichert
    void KOrCL(std::vector<Share> * Bits, int k, int t, mpz_t& q, mpz_t&
        q_1);//k-faches Oder der Sharings in Bits
    void KOrCS(std::vector<Share>& Bits, int k, int t, mpz_t& q);//
        Hilfsroutine für KOrCL

protected:
private:
};
```

C.4 Die Klasse `Vektor_LA`

```
class Vektor_LA
{
    public:
        std::vector<Share> Vector_LA;
        int d;//Dimension

        ///Konstruktor
        Vektor_LA(int Laenge, int n, int t, mpz_t& grosser_Modulus, mpz_t&
             kleiner_Modulus, int Bitlaenge, int Nachkommastellen);

        ///Standard-Konstruktor
        Vektor_LA(){
        d=0;
        mpz_init(q);
        mpz_init(q_1);
        }

        ///Copy-Konstruktor
        Vektor_LA(const Vektor_LA& other);

        ///Destruktor
        virtual ~Vektor_LA();

        ///Operatoren
        Vektor_LA& operator=(const Vektor_LA& other);
        void operator=(int c);//Vektor von konstanten Sharings
            initialisieren
        Vektor_LA& operator=(const Vektor_mpf_t& Init);//Schnittstelle zu
            Vektor_mpf_t.h
```

```
Vektor_LA operator+(const Vektor_LA& VLA);//Vektoraddition
void operator+=(const Vektor_LA& VLA); //Vektoraddititon; Ergebnis
     wird in den Scope geschrieben
Vektor_LA operator-(const Vektor_LA& VLA);//Vektorsubtraktion
void operator-=(const Vektor_LA& VLA);//Vektorsubtraktion;
     Ergebnis wird in den Scope geschrieben
Vektor_LA operator*(mpz_t& Skalar);//Multiplikation mit Skalar
Vektor_LA operator*(int c);//Multiplikation mit int-Zahl
Vektor_LA& operator*=(mpz_t& Skalar);//Multiplikation mit Skalar;
     Ergebnis wird in den Scope geschrieben
Vektor_LA& operator*=(int c);//Multiplikation mit Skalar; Ergebnis
     wird in den Scope geschrieben;
Vektor_LA operator*(Share& Faktor);//Multiplikation mit verteiltem
     Faktor
Vektor_LA& operator*=(Share& Faktor);//Multiplikation mit
     verteilten Faktor; Ergebnis wird in den Scope geschrieben
Share operator[](const Vektor_LA& Index);//Zugriff auf das in
     Index binär gespeicherte Element
Share& operator[](int i);//Zugriffsoperator

///Sonstige elementare Operationen
void Unskaliert_einlesen(const Vektor_mpf_t& rhs);//wie =const
     Vektor_mpf_t& rhs nur unskaliert
void Summe_Vektor(Share& Summe, mpz_t& Modulus);//Summiere die
     Komponenten des Vektors auf und führe die Modulo-Reduktion
     erst am Schluss durch

///Vergrößern, Verkleinern und Umsortieren von Vektoren
void Kopiere_unten(Vektor_LA& V);//den Vektor V von unten in den
     Scope schreiben;
void Oben_einpassen(Vektor_LA& V);//Analog
void Umdrehen();//Dreht die Reihenfolge des Vektors um
void Concat_Vektor(Vektor_LA * a, Vektor_LA *b);//Vektoren
     konkatenieren

///Vergleich von Vektoren
void Gleich_Null(Share& Bit);//Berechne Gleichheit mit Null anhand
     der euklidischen Norm
void Gleich_Null_approximativ(Share& Bit);//Prüfe ob, alle
     Komponenten nicht weiter als 2^-10 von Null entfernt sind

///Matrix- und Skalarmultiplikationen
void Multipliziere_Zeilenweise(Vektor_LA& Binaerer_Vektor);
void MatrixMult(const Matrix_LA& A, Vektor_LA& Ergebnis_Vektor);//
     Matrixmultiplikationmit anschließendem TruncPr; Ergebnis wird
     in Ergebnis_Vektor geschrieben
void MatrixMult_ab_Zeile(const Matrix_LA& A, Vektor_LA&
     Ergebnis_Vektor, int m, int n, int Zeilenzahl);
void SkalarMult(const Vektor_LA& b, Share& Ergebnis);//
     Skalarprodukt; Ergebnis wird in "Ergebnis" geschrieben

///Auslesen von (Teil-)zeilen und (Teil-)spalten
void getZeilenvektor(const Matrix_LA& Ma, int Zeilenindex);//Zeile
     Zeilenindex in Vector_LA kopieren
void getZeilenvektor_ab_Spalte(const Matrix_LA& Ma, int
     Zeilenindex, int Spaltenindex);//Zeilenvektor ab Position
     Spaltenindex in den Scope kopieren
```

```
void getSpaltenvektor(const Matrix_LA& Ma, int Spaltenindex);//
     Spalte Spaltenindex in Vector_LA kopieren
void getSpaltenvektor_ab_Zeile(const Matrix_LA& Ma, int
     Spaltenindex, int Zeilenindex);//Einträge ab Zeile '
     Zeilenindex' in Spalte 'Spaltenindex' in Vektor kopieren
void getSpaltenvektor_ab_Zeile(Vektor_LA& b, int Zeile);//
     Teilspalte aus Vektor auslesen
void getSpaltenvektor_bis_Zeile(Vektor_LA& b, int Zeile);
void getZeilenvektor_bis_Spalte(const Matrix_LA& Ma, int
     Zeilenindex, int Spaltenindex);//Teilzeile aus Matrix
     auslesen
void getSpaltenvektor_bis_Zeile(const Matrix_LA& Ma, int
     Spaltenindex, int Zeilenindex);//Teilspalte aus Matrix
     auslesen

///Ausgabefunktionen
void Ausgabe(std::string Name);//Ausgabe des Vektors
void Ausgabe_skaliert(std::string Name);//wie "Ausgabe";
     Skalierung mit 2^f herausgerechnet
void Ausgabe_skaliert_Vektor_mpf_t(Vektor_mpf_t& V);//Ausgabe nach
      Vektor_mpf_t
void Ausgabe_Shares(std::string Name);//Ausgabe der Shares
int Ausgabe_skaliert_Datei(std::string Name, FILE* datei1);//
     Ausgabe in Datei
int Ausgabe_Datei(std::string Name, FILE* datei1);
void Ausgabe(std::string Name, mpz_t& p);

///Abschneiden überzähliger Stellen
void TruncPr();//Komponentenweises Abschneiden des Vektors mit
     Hilfe des TruncPr-Protokolls
void TruncPr(int l, int r);//Komponentenweises Abschneiden des
     Vektors mit Hilfe des TruncPr-Protokolls ,; Länge der Zahl: k
     Bits, davon f Nachkommastellen

///Vorwärts- und Rückwärtssubstitutionen; Subfunktionen
void VWSubs(const Matrix_LA& L);//Innere Schleife zu
     Skalarprodukten zusammengefasst
void VWSubs_mit_Div(const Matrix_LA& L);//Diagonalelement !=1
void RWSubs(const Matrix_LA& A, Vektor_LA& b);//fasse die innere
     Schleife zu Skalarpodukt zusammen
void getPivot(Vektor_LA& Index);//Pivotelement bestimmen

///Geometrische Funktionen
void Norm(Share& N);//Normberechnung mit Hilfe des Goldschmidt-
     Algorithmus
void Abstand_zu_Ebene(Vektor_LA& Hesseform, Share& b, Share&
     Abstand);//Berechnet den Abstand des Scope-Vektors zu der in
     HNF kodierten Ebene; b steht auf der rechten Seite der HNF

///Funktionen zur Berechnung von und mit Householder-Vektoren
void house_neu();
void Mul_house_Pre_beta_bekannt(Vektor_LA& v, Share& beta);//Prä-
     Multiplikation mit dem Householder-Vektor v und dem bekannten
      beta; analog zur selben Funktion in Matrix_LA
void Qt_mult_b(Vektor_LA& beta, Matrix_LA& M, int d);//
     Multipliziere den Scope-Vektor mit den, aus den in Matrix M
     unterhalb der Diagonalen gespeicherten Householder-Vektoren
```

```
        abgeleiteten Householder-Matrizen
    void Qt_mult_b_neu(Vektor_LA& beta, Vektor_LA& beta_HH, Matrix_LA&
         M, int d);

    ///Manipulation von Index-Vektoren
    void binaerer_Index(Share * Index, int Obergrenze);//Wandelt den
         in "Index" gespeicherten Index in einen binären Vektor um,
         der im Scope gespeichert wird
    void erhoehe_Index();//Schiebt alle Elemente im Scope-Vektor um 1
         nach oben; ist dieser ein binärer Indikator-Vektor wird so
         der Index um 1 erhöht
    void erhoehe_Index_konditional(Share& Bedingung, mpz_t& q);//wie "
         erhöhe_Index" nur unter der Bedingung "Bedingung"

    ///Bestimmung von maximalen und minimalen Elementen
    void Max_Neu(Vektor_LA& Index, int K);//Bestimme das Maximum des
         Scope-Vektors und speichere den Index binär in "Index"; der
         Scope-Vektor wird verändert; vgl. Protocol 13 von Tofts
         Dissertation
    void Max_Bruch(Vektor_LA& B, Vektor_LA& Index); //Wie Max nur daß
         Brüche betrachtet werden
    void Min_Bruch(Vektor_LA& C, Vektor_LA& W);//basiert auf Max_Bruch
    void Min(Vektor_LA& Index, int K);//Bestimme das Minimum durch:
         Min(x)=Max(-x)

    ///Logische Funktionen
    void AND(Vektor_LA& Argument_2);//Komponentenweises AND
    void OR(Vektor_LA& Argument_2);//OR von verteilten (Bit-)Vektoren;
    void Pre_Mul_C();//PreMulC nach Protocols 4.2 in: "Improved
         Primitives for Secure Multiparty Integer Computation"
    void PreOrC();
    void KOR(Share& Ausgabe, int r, mpz_t& p);//Standard-Algorithmus

    ///Eliminieren von Nullzeilen
    void Vektor_Zusammenschieben(std::vector<Share>& Aktive_Menge_q,
         Matrix_LA& P);//Entferne Nullelemente aus Vektor; P ist
         zugehörige Permutationsmatrix
    void Vektor_Zusammenschieben_binaer(std::vector<Share>&
         Aktive_Menge_q, Matrix_LA& P);//Wie Vektor_Zusammenschieben;
         die aktiven Elemente sind bereits binär kodiert in
         Aktive_Menge_q

    ///Verdecktes Lesen und Schreiben von Elementen
    void SecReadRow(Matrix_LA& A, Vektor_LA& I,  int l);//sicheres
         Auslesen einer versteckten Zeile
    void SecWrite(Vektor_LA& V, Share& s);

    ///Bestimmung des ersten Nicht-Null-Elements
    void SelectFirst();//vgl. Secure SCM D3.2
    void FindFirst(Vektor_LA& Index);

    ///Funktionen zum Umwandeln von Zahlen in binäre Vektoren
    void BitMask(Share& x);//vgl. Secure SCM D3.2, Variante

protected:
private:
    int Spielerzahl;
```

```
        int Schwellenwert;
        int k;
        int N;
        mpz_t q;
        mpz_t q_1;
};
```

C.5 Die Klasse `Matrix_LA`

```
class Matrix_LA//Matrix-Klasse; Objekte sind Matrizen von Sharings
{
    public:
        int Zeilenzahl;
        int Spaltenzahl;

        Share ** M;

        ///Konstruktoren
        Matrix_LA(int a, int b, int n, int t, mpz_t& grosser_Modulus,
            mpz_t& kleiner_Modulus, int Bitlaenge, int Nachkommastellen);
        Matrix_LA(int a, int b, int n, int t, const mpz_t& grosser_Modulus
            , const mpz_t& kleiner_Modulus, int Bitlaenge, int
            Nachkommastellen);

        ///Destruktor
        virtual ~Matrix_LA();

        ///Copy-Konstruktor
        Matrix_LA(const Matrix_LA& other);

        ///Operatoren
        Matrix_LA& operator=(const Matrix_LA& other);
        Matrix_LA operator=(int n);//Initialisiere die Matrix mit einem
            konstanten Wert (typischerweise 0)
        Matrix_LA& operator=(const Matrix_mpf_t& Init);//Schnittstelle zu
            Matrix_mpf_t
        Matrix_LA operator+(const Matrix_LA& Matrix_2);//Addition zweier
            Matrizen
        Matrix_LA& operator+=(const Matrix_LA& Matrix_2);//Addition zweier
             Matrizen; Ergebnis wird in den Scope geschrieben
        Matrix_LA operator-(const Matrix_LA& Matrix_2);//Subtraktion
            zweier Matrizen
        Matrix_LA& operator-=(const Matrix_LA& Matrix_2);//Subtraktion
            zweier Matrizen
        Matrix_LA operator*(const Matrix_LA& A);//Matrizenmultiplikation
        Matrix_LA operator*(const Share& S);
        Matrix_LA& operator*=(const Matrix_LA& A);//Matrizenmultiplikation
            . Ergebnis wird in den Scope geschrieben
        Matrix_LA& operator*=(int c);

        ///Transponieren von Matrizen und andere Manipulationen
        Matrix_LA Transponiere();
        void Transponiere_quadratisch();
        void Loesche_links_unten();//Lösche unterhalb der Diagonale
        void Ersetze_Spalte(std::vector<Share>& Spalte, int Spaltenindex,
            int l);//Ersetze die Spalte Spaltenindex durch Spalte
```

```
void Ersetze_Zeile(std::vector<Share>& Zeile, int Zeilenindex, int
    l);

///Erstellung von Matrizen
void Erzeugung(mpz_t ** Werte);//Belege die Matrix mit den Werten
    aus "Werte"
void Erzeugung_int(int ** Werte);//Belege die Matrix mit den
    Werten aus "Werte"
void Erzeugung_double(double ** Werte);//Belege die Matrix mit den
     Werten aus "Werte"

///Erstellung besonderer Matrizen
void Einheitsmatrix(mpz_t& Q);//Erzeuge Einheitsmatrix mit Shares
    in Z/QZ
void Einheitsmatrix_Fixpunkt(mpz_t& Q);//Einheitsmatrix skaliert
    mit 2^f
void Zufallsmatrix(int k);

///Ausgabefunktionen
void Ausgabe(std::string Name);//Ausgabe der Matrix
void Ausgabe_skaliert(std::string Name);//skalierte Ausgabe der
    Matrix (in Fixpunktarithmetik)
void Ausgabe_skaliert_Matrix(mpf_t** N, int m, int n);//Ausgabe
    der Werte in N
int Ausgabe_skaliert_Datei(std::string Name, FILE* datei1);//
    skalierte Ausgabe in Datei Ausgabe.txt; Rückgabe zeigt an, ob
     Öffnen der Datei erfolgreich

///Abschneiden von Nachkommastellen nach Multiplikationen
void TruncPr(int k, int r);

///Zeilen-/Spaltenweise Multiplikationen mit Vektoren
void Multipliziere_Zeilenweise(std::vector<Share> *
    Binaerer_Vektor);
void Multipliziere_Spaltenweise(std::vector<Share>&
    Binaerer_Vektor);
void Multipliziere_Zeilenweise_ab_Zeile_X(std::vector<Share> *
    Binaerer_Vektor, int X);//wie Multipliziere_Zeilenweise; nur
    erst ab Zeile X

///Matrizenmultiplikation
Matrix_LA MatrixMultMatrix(const Matrix_LA& A);//
    Matrizenmultiplikation ohnr Trunc; die Rückgabematrix wird in
     der Methode erzeugt
Matrix_LA MatrixMultMatrix_ab_Zeile_l(Matrix_LA& A, int l);//Die
    ersten l Zeilen des Ergebnisses sind mit den Ersten l Zeilen
    von A identisch. Voraussetzung: Scope-Matrix quadratisch
void Matrix_aus_Vektor(std::vector<Share>& v_Strich,std::vector<
    Share>& w_Strich, int Laenge_v, int Laenge_w);//Bilde Produkt
     vw^t

///Kopieren aus/in größere Matrizen
void Rechts_unten_einpassen(Matrix_LA& N, int z, int s);//Matrix N
     rechts unten in Matrix einpassen
void Links_oben_einpassen(Matrix_LA& N);//Analog
void Rechts_unten_auspassen(Matrix_LA& N, int z, int s);//Rechte
    untere Ecke in kleinere Matrix N kopieren
```

```
void Rechts_einpassen(Matrix_LA& A);//A von rechts in die Scope-
     Matrix einpassen
void Kopiere_links(Matrix_LA& L);//Kopiere den linken Teil der
     Matrix L in die Scope-Matrix
void Kopiere_rechts(Matrix_LA& L);//Kopiere den rechten Teil der
     Matrix L in die Scope-Matrix
void Oben_einpassen(Matrix_LA& A);
void Unten_einpassen(Matrix_LA& A);

///Eliminieren von Nullzeilen/-spalten
void Matrix_Zusammenschieben(std::vector<Share>& Aktive_Menge_q,
     Matrix_LA& P);//Eliminiere Leerzeilen in der Scope-Matrix
void Matrix_Zusammenschieben_LT(std::vector<Share>& Aktive_Menge_q
     );//Wie Matrix_Zusammenschieben, nur daß es ausreicht, wenn
     in Aktive_Menge_q ein Index unterschritten wird
void Matrix_Zusammenschieben_binaer(std::vector<Share>&
     Aktive_Menge_q, Matrix_LA& P);//Aktive_Menge_q wird als binär
      vorausgesetzt mit 1 an den Stellen, die zu Nullzeilen
     korrespondieren
void Matrix_spaltenweise_zusammenschieben_binaer(std::vector<Share
     >& Aktive_Menge);//wie Matrix_Zusammenschieben_binaer nur
     spaltenweise
void Matrix_spaltenweise_zusammenschieben_binaer(std::vector<Share
     >& Aktive_Menge, Matrix_LA& P);

///Normberechnungen
void Eins_Norm(Share& Norm);//Berechne 1-Norm der Matrix

///Lösen von linearen Gleichungssystemen Ax=b
void Loese_GLS_LR(std::vector<Share>& b, int d);//Loese das
     Gleichungssystem mit LR-Zerlegung mit Pivotisierung; die
     Lösung wird in b geschrieben
void Loese_GLS_QR(std::vector<Share>&b, int d);//Loese das
     Gleichungssystem mit QR-Zerlegung; die Lösung wird in b
     geschrieben
void Loese_GLS_Cholesky(std::vector<Share>& b_Strich, int d);//
     Loese das GLS mit Hilfe der Cholesky-Zerlegung (geht nur bei
     s.p.d- Matrizen)
void CG(std::vector<Share>& b_Strich, std::vector<Share>& x_Strich
     , int D);//Löse das GlS Ax=b mit Hilfe des CG-Algorithmus;
     eine Prüfung auf s.p.d. der Matrix erfolgt natürlich nicht

///Hilfsroutinen zur LR-Zerlegung
void Vorwaertselimination_SMPC_mit_Pivot(Matrix_LA& P);//Die
     verwendete Permutationsmatrix wird in P gespeichert
void Erstelle_Permutationsmatrix(std::vector<Share>& Spalte, int l
     );//Sucht das betragsmäßig größte Element in Zeile k oder
     größer und vertauscht die entsprechende Zeile mit der k-ten

///Sicheres Schreiben von Zeilen/Spalten
void SecWriteSpalte(std::vector<Share>& V, std::vector<Share>& B,
     int V_d, int B_d);//schreibe V in die in B kodierte Spalte
void SecWriteZeile(std::vector<Share>& V, std::vector<Share>& B,
     int V_d, int B_d);//Schreibe V in die durch B kodierte Zeile

///Rechnen mit Householder-Vektoren und Matrizen
```

```
        void Householder_Matrix(std::vector<Share>& v, int d);//Erstelle
             Householder-Matrix mit Hilfe des Vektors v;
        void Multipliziere_Householder_Prae(std::vector<Share>& v_Strich,
             Share& beta, int d);//Multiplikation der Matrix mit dem
             Householder-Vektor v'
        void Multipliziere_Householder_Prae_beta_bekannt(std::vector<Share
             >& v_Strich, Share& beta, int d);//Prä-Householder-
             Multiplikation mit bekanntem beta
        void Multipliziere_Householder_Post_beta_bekannt(std::vector<Share
             >& v_Strich, Share& beta, int d);//Post-Householder-
             Multiplikation mit bekanntem beta, vgl. Golub
        void QR_Householder(std::vector<Share>& b, std::vector<Share>&
             beta); //QR-Zerlegung nach Householder; Householder-Vektoren
             werden links unten gespeichert; 1. Komponenten in beta; R
             rechts oben
        void QR_Householder_Rechtecksmatrix(std::vector<Share>& b, std::
             vector<Share>& beta);
        void QR_Householder_konditional(std::vector<Share>& b, std::vector
             <Share>& Index);//wie QR_Householder, es werden allerdings
             nur die in 'Index' gekennzeichneten Vektoren berücksichtigt
        void QR_Householder(Matrix_LA& Q);//QR-Zerlegung nach Householder;
              R wird im Scope gespeichert; die Householder-Matrix in Q
        void Berechne_Q_konditional(std::vector<Share>& beta, std::vector<
             Share>& Index);//Wie Berechne_Q; allerdings werden nur die HH
             -Vektoren berücksichtigt, die in 'Index' vermerkt sind
        void Berechne_Q_neu(std::vector<Share>& beta, std::vector<Share>&
             b);//Berechne Matrix Q aus Householder-Vektoren, die links
             unten gespeichert sind; in b sind die 1. Komponenten der HH-
             Vektoren gespeichert
        void Berechne_Q_neu(std::vector<Share>& beta, std::vector<Share>&
             b, int Anzahl_HH_Vektoren);//in b sind die oberen Werte der
             HH-Vektoren gespeichert; für den Fall, daß weniger als dim Q
             -1 Vektoren vorgesehen sind

        ///Cholesky-Zerlegung
        void Cholesky();//Berechne Cholesky-Zerlegung; Die untere
             Dreiecksmatrix wird überschrieben.

    protected:
        int Spielerzahl;
        int Schwellenwert;
        int k;
        int N;
        mpz_t q;
        mpz_t q_1;
    private:
};
```

C.6 Die Klasse Simplex_Startwert

```
class Simplex_Startwert : public Matrix_LA
{
    public:
        //Konstruktor
        Simplex_Startwert(int m, int n, int
             Anzahl_Ungleichheitsbedingungen, int Anzahl_Spieler, int
             Schwelle, mpz_t& qmod, mpz_t& q_1mod, int kl, int ffl);
```

```
        //Destruktor
        virtual ~Simplex_Startwert();

        //Copy-Konstruktor
        Simplex_Startwert(const Simplex_Startwert& other);

        //Operatoren
        Simplex_Startwert& operator=(const Simplex_Startwert& other);

        //Simplex-Algorithmus
        void Simplex_Algorithmus(Matrix_LA& A_alt, Vektor_LA&b, Vektor_LA&
             c, Vektor_LA& x0, Vektor_LA& X, int
            Anzahl_Gleichheitsbedingungen, int
            Anzahl_Ungleichheitsbedingungen, Vektor_LA& B, Vektor_LA& N);

        //Hilfsfunktionen
        void UpdTab(Vektor_LA& C, Vektor_LA& R, Vektor_LA& V, Vektor_LA& W
            , Share& p);//Update des Tableaus
        int GetPivColumn(Vektor_LA& V);
        int GetPivRow(Vektor_LA& C, Vektor_LA& W);//D Eingabe-, W
            Rückgabevektor. Eingabevektor nicht als Referenz, da im
            Algorithmus manipuliert
        void UpdVar(Vektor_LA& B, Vektor_LA& N, Vektor_LA& V, Vektor_LA& W
            );//Aktualisieren von Basis- und Nichtbasisvariablen
        void GetSolution(Vektor_LA& B, Vektor_LA& X);
        void UpdCost(Vektor_LA& B, Matrix_LA& A, Vektor_LA b, int d);//
            Aktualisiere die Kostenfunktion
        void TransformCost(Vektor_LA& W, Matrix_LA& A, Vektor_LA& b_Signum
            );//Stelle die aktualisierte Kostenfunktion in Abhängigkeit
            von der Menge N dar
        void TransformCost(Matrix_LA& A, Vektor_LA& b_Signum);//Stelle die
             aktualisierte Kostenfunktion in Abhängigkeit von der Menge N
             dar; Vektor C_B wird nicht mit dem Signum von b
            multipliziert

    protected:
    private:
        //Member-Variablen
        int Tableau_Zeilen;
        int Tableau_Spalten;
        int Spielerzahl;
        int Schwellenwert;
        mpz_t q;
        mpz_t q_1;
        int k;
        int ff;
};
```

C.7 Die Klasse `Share_F2M`

```
class Share_F2M
{
    public:
        int Anzahl_Spieler;
        int Schwellenwert;
```

```
        GF256 * Vector;

        Share_F2M(int n, int t);
        virtual ~Share_F2M();
        Share_F2M(const Share_F2M& other);

        Share_F2M& operator=(const Share_F2M& other);
        Share_F2M& operator*=(int Faktor);
        Share_F2M& operator+=(int Faktor);
        Share_F2M operator*(const Share_F2M& Faktor);
        Share_F2M& operator*=(const Share_F2M& Faktor);

        void Erzeugung(GF256 Wert);//Share-Erzeugung mittels dividierter
             Differenzen
        void Erzeugung_GRR(GF256 Wert);//Share-Erzeugung mittels
             Polyomauswertung
        void Rekonstrukt(GF256& s, int t);//mit dividierten Differenzen;
        void Rekonstrukt_Ausgabe(std::string N, int t);
        void Rekonstrukt_GRR(GF256& Wert, int t);//Rekonstruktion mittels
             Lagrange-Interpolation
        void Rekonstrukt_Spieler_beliebig(GF256 * s, int * Spielerliste,
             int t);//Rekonstruktion mit beliebigen Spielern - abgeleitet
             von Share-Version
        void Zufall();//Zufallszahlen auf Positionen 1,...,t des Vektors

        void Add(Share_F2M Summand_2);//Zweiten Share addieren
        void Add_Konst_int(int Konstante);//Konstante addieren
        void Mul(const Share_F2M& Faktor_2);//Multiplikation zweier Shares
             analog Lory_2
        void Mul_Konst(GF256 Konstante);
        void Mul_Konst_int(int c);

        void RandNumbers(std::vector<Share_F2M> * randNumbers, int t,
             const Share_F2M& a, const Share_F2M& b);//Hilfsprogramm für
             Multiplikation
        void LagIntKoeff(GF256 *lambda, int n, char *name);//Hilfprogramm
             für Multiplikation GRR
        void Mul_GRR(const Share_F2M& b, int t);

        void PreOr_Bits_nicht_cR(std::vector<Share_F2M>& Faktoren, int
             Anzahl_Faktoren);//PreOr für Biteingabe
        void FindFirst(std::vector<Share_F2M>& VektorF2M, std::vector<
             Share_F2M>& Index, int l);//Findet erstes 1-Element eines
             Vektors und speichert es in Index
        void BitAdd_F2M(std::vector<Share_F2M>& a, std::vector<Share_F2M>&
             b, int k);//Bitweise Addition von bitweisen c und r;
             Ergebnis in c
        void BitAdd_F2M_offen(std::vector<Share_F2M> *c, int * r, int k);
             //Bitweise Addition von bitweisen c und r; r offen; Ergebnis
             landet in a
    protected:
    private:
};
```

C.8 Die Klasse SVM

```
class SVM
{
    public:
        Matrix_LA Daten;

        int Hoehe_Daten;
        int Breite_Daten;

        int Spielerzahl;
        int Schwellenwert;

        mpz_t q;
        mpz_t q_1;

        int k;
        int N;

        Matrix_LA K;//Kernel-Matrix
        Matrix_LA G;//Matrix des Optimierungsproblems
        Vektor_LA Y;//Klassenindikatoren
        Vektor_LA Alpha;//Lösungsvektor für die Lagrange-Multiplikatoren
        Vektor_LA SV;//Support-Vektoren binär kodiert
        Share b;//Konstanter Term der ges. Hyperfläche
        Share Lambda;//Toleranzparameter für SM-SVM
        Share Koeff;//Faktor bei quadratischem Kernel
        Share r;//Konstanter Term bei quadratischem Kernel

        ///Konstruktor
        SVM(int m, int n, int Anzahl_Spieler, int PolyGrad, mpz_t& Modulus
            , mpz_t& kleiner_Modulus, int kk, int ff);

        ///Destruktor
        virtual ~SVM();

        ///Copy-Konstruktor
        SVM(const SVM& other);

        ///Operatoren
        SVM& operator=(const Matrix_mpf_t& rhs);//lädt in Member-Variable
            "Daten"
        SVM& operator=(const SVM& other);

        ///Methoden zur Kernel-Berechnung
        void Linearer_Kernel();
        void Quadratischer_Kernel(Share& Koeff, Share& Konst);
        void Quadratischer_Kernel(double Koeff, double Konst);//ruft QK
            auf

        ///Berechnung der Matrix des QPP
        void Berechne_G();//Berechne die Matrix des Optimierungsproblems \
            sum_i,j y_iy_jK_ij

        ///Berechnung der Translation der Hyperebene
        void Berechne_b(); //Berechnung des konstanten Terms in der
            Entscheidungsregel
```

```
        ///Berechnung der Support-Vektoren
        void Berechne_SV();//Berechne SVen

        ///Entscheidungsregeln
        void Entscheidungsregel_linear(Share& Ergebnis, Vektor_LA& x);
        void Entscheidungsregel_Test(std::string Name);//Teste die (
            lineare) SVM mit den Trainingsdaten
        void Entscheidungsregel_quadratisch(Share& Ergebnis, Vektor_LA& x)
            ;
        void SVM_Test_linear(Matrix_LA& Daten_Test, Vektor_LA& Y_Test,std
            ::string Name);//Klassifiziere die Testdaten und gib den
            korrekt klassifizierten Prozentsatz an Daten aus
        void SVM_Test_quadratisch(Matrix_LA& Daten_Test, Vektor_LA& Y_Test
            ,std::string Name);//w.o. nur quadratischer Kernel

        ///Speicherung de relevanten Daten auf Festplatte
        void Save_SVM(std::string Dateiname);//Speichern der SVM in
            Dateien
    protected:
    private:
};
```

C.9 Die Klasse Vektor_mpf_t

```
class Vektor_mpf_t
{
    public:
        int Dimension;
        mpf_t * V; //!< Member variable "V"

        Vektor_mpf_t(int d);
        Vektor_mpf_t(int d, double*w);
        Vektor_mpf_t(int d, mpf_t*w);

        virtual ~Vektor_mpf_t();
        Vektor_mpf_t(const Vektor_mpf_t& other);
        Vektor_mpf_t& operator=(const Vektor_mpf_t& other);

        Vektor_mpf_t operator+(const Vektor_mpf_t& W);
        Vektor_mpf_t& operator+=(const Vektor_mpf_t& W);
        Vektor_mpf_t operator-(const Vektor_mpf_t& W);
        Vektor_mpf_t& operator-=(const Vektor_mpf_t& W);
        Vektor_mpf_t& operator*=(mpf_t& Faktor);
        void operator=(int n);

        void SkalarProdukt(Vektor_mpf_t& v2, mpf_t& Ergebnis);
        void MatrixMultVektor(Matrix_mpf_t& M, Vektor_mpf_t& Ergebnis);

        void Zufallsvektor();

        double Norm();

        void SetV(mpf_t * val) { V = val; }
        void getSpaltenvektor(int Spalte, Matrix_mpf_t& A); //
            Spaltenvektor auslesen
```

```
        void getSpaltenvektor_ab_Zeile(int Spalte, Matrix_mpf_t& A, int
             Zeile);//Spaltenvektor ab Zeile "Zeile" auslesen

        void VWSubs(Matrix_mpf_t& L);
        void RWSubs(Matrix_mpf_t& L);

        void Ausgabe(std::string Name);
        void Ausgabe_wissenschaftlich(std::string Name);//in
             exponentieller Notation
        void Ausgabe_Datei(std::string Name, FILE* datei);
        void Ausgabe_wissenschaftlich_Datei(std::string Name, FILE* datei)
             ;//Ausgabe in Datei

        int Max();

        void Householder_Vektor();
    protected:
    private:
};
```

C.10 Die Klasse `Matrix_mpf_t`

```
class Matrix_mpf_t
{
    public:
        mpf_t ** M;
        int Hoehe, Breite;
        Matrix_mpf_t(int m, int n);
        Matrix_mpf_t(int m, int n, double** MM);

        virtual ~Matrix_mpf_t();
        Matrix_mpf_t(const Matrix_mpf_t& other);

        Matrix_mpf_t& operator=(const Matrix_mpf_t& other);
        Matrix_mpf_t& operator+=(const Matrix_mpf_t& A);
        Matrix_mpf_t& operator-=(const Matrix_mpf_t& A);
        Matrix_mpf_t& operator=(int n);//n in ale Einträge der Matrix
             schreiben
        Matrix_mpf_t operator*(const Matrix_mpf_t& B);//
             Matrizenmultiplikation
        Matrix_mpf_t& operator*=(double d);//Multiplikation mit Skalar

        void Einheitsmatrix();
        void Zufallsmatrix();
        void InverseMatrix();

        void SpalteEinfuegen(int Spalte, mpf_t* v, int Dimension);
        void Rechts_unten_auspassen(Matrix_mpf_t& N, int z, int s);
        void Rechts_unten_einpassen(Matrix_mpf_t& N, int z, int s);
        Matrix_mpf_t Transponiere();
        void Loesche_links_unten();
        Matrix_mpf_t Matrix_aus_Vektor(mpf_t * H, mpf_t * HG);
        void Kopiere_rechts(Matrix_mpf_t& R);//Rechten Teil der Matrix R
             in die Scope-Matrix kopieren
```

```
        void Norm_Frobenius(mpf_t& Norm);//Berechne die Frobenius-Norm der
             Matrix
        double Kondition();//Berechne die Kondition anhand der Frobenius-
            Norm

        void VWElimination(mpf_t * b);
        void GaussAlgorithmus(mpf_t * b);
        void Pivot(mpf_t * b, int k);

        void Ausgabe(std::string Name);
        void Ausgabe_Datei(std::string Name, std::string DateiName); //
            Ausgabe in DateiName

        void Multipliziere_Householder_Prae(mpf_t * v, int d);
        void QR_Householder(Matrix_mpf_t& Q);
    protected:
    private:
};
```

C.11 Die Klasse GF256

```
class GF256 {
        private:
                static const int modulus = 256;

        public:
        unsigned char value;

                GF256();
                inline ~GF256(){};
                GF256(unsigned char value);

                void operator = (const GF256 &gf);
                void operator = (const int value);
                void operator = (const unsigned char value);

                GF256 operator +(const GF256 &gf);
                GF256 operator -(const GF256 &gf);
                GF256 operator ^(const GF256 &gf);
                GF256 operator *(const GF256 &gf);
                GF256 operator /(const GF256 &gf);

                void operator +=(const GF256 &gf);
                void operator -=(const GF256 &gf);
                void operator ^=(const GF256 &gf);
                void operator *=(const GF256 &gf);
                void operator /=(const GF256 &gf);


                GF256 inverse();
                unsigned char getValue();
};
```

D Die Implementierungen der Optimierungsroutinen

D.1 Der primale Algorithmus

Die `main`-Funktion

```
int main()
{
int i,j,n=5, t=2, k=128, Iterationszaehler=1;
mpf_set_default_prec(300);

mpz_t s, Ende;
clock_t start_programm,stop_programm;

mpz_init(Ende);
mpz_init_set_ui(s,1);

mpz_t q;
mpz_t q_1;
mpz_t basis;
mpz_t modulo;
mpz_t modulo_q1;

mpz_init(q);
mpz_init(q_1);
mpz_init_set_ui(basis,2);
mpz_init(modulo);
mpz_init(modulo_q1);

mpz_pow_ui(q,basis,Modullaenge);
mpz_pow_ui(q_1,basis,63);

do{
mpz_nextprime(q,q);
mpz_nextprime(q_1,q_1);
mpz_mod_ui(modulo,q, 4);
mpz_mod_ui(modulo_q1,q_1, 4);
}
while(mpz_cmp_ui(modulo,3)!=0 || mpz_cmp_ui(modulo_q1,3)!=0); //Erzeuge
     geeignete Primzahl für Berechnung des modulus (muß kongruent 3 mod 4
     sein)

haupt::Berechne_Koeffizienten(n,t,k,f,q,q_1);

mpz_init_set(grosser_Modulus,q);
mpz_init_set(kleiner_Modulus,q_1);
```

```
int Zaehler_voller_Schritt=0, Zaehler_halber_Schritt=0;

mpz_t unendlich;
mpz_init_set_ui(unendlich,1);
mpz_mul_2exp(unendlich,unendlich,k-2);
mpz_sub_ui(unendlich,unendlich,1);//unendlich=2^{k-1}-1

double ** Werte, **Q_Werte;

#ifdef Problem_A
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,2,n,t,q,q_1,k,f);


Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=2.0;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=1.0;
Werte[0][1]=-2.0;
Werte[1][0]=-1.0;
Werte[1][1]=-2.0;
Werte[2][0]=-1.0;
Werte[2][1]=2.0;
Werte[3][0]=1.0;
Werte[3][1]=0.0;
Werte[4][0]=0.0;
Werte[4][1]=1.0;

b[0].Erzeugung_double(-2.0,t,q);
b[1].Erzeugung_double(-6.0,t,q);
b[2].Erzeugung_double(-2.0,t,q);
b[3].Erzeugung_double(0.0,t,q);
b[4].Erzeugung_double(0.0,t,q);

c.Vector_LA.at(0).Erzeugung_double(-2.0,t,q);
c.Vector_LA.at(1).Erzeugung_double(-5.0,t,q);

Vektor_mpf_t x_korrekt(2);
mpf_set_d(x_korrekt.V[0],1.4);
mpf_set_d(x_korrekt.V[1],1.7);

mpf_t f_korrekt;
```

```
mpf_init_set_d(f_korrekt,-6.45);
#endif

#ifdef Problem_21
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);//ist hier =0
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,2,n,t,q,q_1,k,f);


Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=0.02;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=10.0;
Werte[0][1]=-1.0;
Werte[1][0]=-1.0;
Werte[1][1]=0.0;
Werte[2][0]=1.0;
Werte[2][1]=0.0;
Werte[3][0]=0.0;
Werte[3][1]=-1.0;
Werte[4][0]=0.0;
Werte[4][1]=1.0;

b.Vector_LA.at(0).Erzeugung_double(10.0,t,q);
b.Vector_LA.at(1).Erzeugung_double(-50.0,t,q);
b.Vector_LA.at(2).Erzeugung_double(2.0,t,q);
b[3].Erzeugung_double(-50.0,t,q);
b[4].Erzeugung_double(-50.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],2.0);
mpf_set_d(x_korrekt.V[1],0.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,0.04);
#endif

#ifdef Problem_35
Vektor_LA x(3,n,t,q,q_1,k,f);
Vektor_LA d(3,n,t,q,q_1,k,f);
Vektor_LA c(3,n,t,q,q_1,k,f);
Vektor_LA b(4,n,t,q,q_1,k,f);
Matrix_LA A(4,3,n,t,q,q_1,k,f);
```

```
Matrix_LA G(3,3,n,t,q,q_1,k,f);

Werte=new double *[4];
for(i=0; i<4; i++){
Werte[i]=new double[3];
}


Q_Werte=new double*[3];
for(i=0; i<5; i++){
     Q_Werte[i]=new double[3];
     }

Q_Werte[0][0]=4.0;
Q_Werte[0][1]=2.0;
Q_Werte[0][2]=2.0;

Q_Werte[1][0]=2.0;
Q_Werte[1][1]=4.0;
Q_Werte[1][2]=0.0;

Q_Werte[2][0]=2.0;
Q_Werte[2][1]=0.0;
Q_Werte[2][2]=2.0;


Werte[0][0]=-1.0;
Werte[0][1]=-1.0;
Werte[0][2]=-2.0;

Werte[1][0]=1.0;
Werte[1][1]=0.0;
Werte[1][2]=0.0;

Werte[2][0]=0.0;
Werte[2][1]=1.0;
Werte[2][2]=0.0;

Werte[3][0]=0.0;
Werte[3][1]=0.0;
Werte[3][2]=1.0;


c[0].Erzeugung_double(-8.0,t,q);
c[1].Erzeugung_double(-6.0,t,q);
c[2].Erzeugung_double(-4.0,t,q);

b[0].Erzeugung_double(-3.0,t,q);
b[1].Erzeugung_double(0.0,t,q);
b[2].Erzeugung_double(0.0,t,q);
b[3].Erzeugung_double(0.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],4.0);
mpf_div_ui(x_korrekt.V[0],x_korrekt.V[0],3);
mpf_set_d(x_korrekt.V[1],7.0);
mpf_div_ui(x_korrekt.V[1],x_korrekt.V[1],9);
```

```
mpf_set_d(x_korrekt.V[2],4.0);
mpf_div_ui(x_korrekt.V[2],x_korrekt.V[2],9);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-8.0);
mpf_div_ui(f_korrekt,f_korrekt,9);
mpf_sub_ui(f_korrekt,f_korrekt,8);//f=-8 8/9
#endif

#ifdef Problem_76
Vektor_LA x(4,n,t,q,q_1,k,f);
Vektor_LA d(4,n,t,q,q_1,k,f);
Vektor_LA c(4,n,t,q,q_1,k,f);
Vektor_LA b(7,n,t,q,q_1,k,f);
Matrix_LA A(7,4,n,t,q,q_1,k,f);
Matrix_LA G(4,4,n,t,q,q_1,k,f);

Werte=new double *[7];
for(i=0; i<7; i++){
Werte[i]=new double[4];
}


Q_Werte=new double*[4];
for(i=0; i<5; i++){
    Q_Werte[i]=new double[4];
    }

Q_Werte[0][0]=2.0;
Q_Werte[0][1]=0.0;
Q_Werte[0][2]=-1.0;
Q_Werte[0][3]=0.0;

Q_Werte[1][0]=0.0;
Q_Werte[1][1]=1.0;
Q_Werte[1][2]=0.0;
Q_Werte[1][3]=0.0;

Q_Werte[2][0]=-1.0;
Q_Werte[2][1]=0.0;
Q_Werte[2][2]=2.0;
Q_Werte[2][3]=1.0;

Q_Werte[3][0]=0.0;
Q_Werte[3][1]=0.0;
Q_Werte[3][2]=1.0;
Q_Werte[3][3]=1.0;


Werte[0][0]=-1.0;
Werte[0][1]=-2.0;
Werte[0][2]=-1.0;
Werte[0][3]=-1.0;

Werte[1][0]=-3.0;
Werte[1][1]=-1.0;
Werte[1][2]=-2.0;
```

```
Werte[1][3]=1.0;

Werte[2][0]=0.0;
Werte[2][1]=1.0;
Werte[2][2]=4.0;
Werte[2][3]=0.0;

Werte[3][0]=1.0;
Werte[3][1]=0.0;
Werte[3][2]=0.0;
Werte[3][3]=0.0;

Werte[4][0]=0.0;
Werte[4][1]=1.0;
Werte[4][2]=0.0;
Werte[4][3]=0.0;

Werte[5][0]=0.0;
Werte[5][1]=0.0;
Werte[5][2]=1.0;
Werte[5][3]=0.0;

Werte[6][0]=0.0;
Werte[6][1]=0.0;
Werte[6][2]=0.0;
Werte[6][3]=1.0;

c[0].Erzeugung_double(-1.0,t,q);
c[1].Erzeugung_double(-3.0,t,q);
c[2].Erzeugung_double(1.0,t,q);
c[3].Erzeugung_double(-1.0,t,q);

b[0].Erzeugung_double(-5.0,t,q);
b[1].Erzeugung_double(-4.0,t,q);
b[2].Erzeugung_double(1.5,t,q);
b[3].Erzeugung_double(0.0,t,q);
b[4].Erzeugung_double(0.0,t,q);
b[5].Erzeugung_double(0.0,t,q);
b[6].Erzeugung_double(0.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],3.0);
mpf_div_ui(x_korrekt.V[0],x_korrekt.V[0],11);//0,272727272727... =3/11
mpf_set_d(x_korrekt.V[1],23.0);
mpf_div_ui(x_korrekt.V[1],x_korrekt.V[1],11);//2,09090909090909... =23/11
mpf_set_d(x_korrekt.V[2],0.26);
mpf_div_ui(x_korrekt.V[2],x_korrekt.V[2],10000000000);
mpf_set_d(x_korrekt.V[3],6.0);
mpf_div_ui(x_korrekt.V[3],x_korrekt.V[3],11);//0,545454545454... =6/11

mpf_t f_korrekt;
mpf_init(f_korrekt);

mpf_t h;
mpf_init(h);

mpf_mul(f_korrekt,x_korrekt.V[0],x_korrekt.V[0]);
```

```
mpf_mul(h,x_korrekt.V[1],x_korrekt.V[1]);
mpf_div_ui(h,h,2);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[2],x_korrekt.V[2]);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[3],x_korrekt.V[3]);
mpf_div_ui(h,h,2);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[0],x_korrekt.V[2]);
mpf_sub(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[2],x_korrekt.V[3]);
mpf_add(f_korrekt,f_korrekt,h);
mpf_sub(f_korrekt,f_korrekt,x_korrekt.V[0]);
mpf_mul_ui(h,x_korrekt.V[1],3);
mpf_sub(f_korrekt,f_korrekt,h);
mpf_add(f_korrekt,f_korrekt,x_korrekt.V[2]);
mpf_sub(f_korrekt,f_korrekt,x_korrekt.V[3]);
#endif

#ifdef Problem_224
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);//ist hier =0
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(4,n,t,q,q_1,k,f);
Matrix_LA A(4,2,n,t,q,q_1,k,f);

Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=4.0;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=1.0;
Werte[0][1]=3.0;
Werte[1][0]=-1.0;
Werte[1][1]=-3.0;
Werte[2][0]=1.0;
Werte[2][1]=1.0;
Werte[3][0]=-1.0;
Werte[3][1]=-1.0;

b.Vector_LA.at(0).Erzeugung_double(0.0,t,q);
b.Vector_LA.at(1).Erzeugung_double(-18.0,t,q);
b.Vector_LA.at(2).Erzeugung_double(0.0,t,q);
b[3].Erzeugung_double(-8.0,t,q);

c[0].Erzeugung_double(-48.0,t,q);
c[1].Erzeugung_double(-40.0,t,q);
```

```
Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],4.0);
mpf_set_d(x_korrekt.V[1],4.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-304.0);
#endif

#ifdef Problem_268
Vektor_LA x(5,n,t,q,q_1,k,f);
Vektor_LA d(5,n,t,q,q_1,k,f);
Vektor_LA c(5,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,5,n,t,q,q_1,k,f);
Matrix_LA G(5,5,n,t,q,q_1,k,f);

Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[5];
}


Q_Werte=new double*[5];
for(i=0; i<5; i++){
    Q_Werte[i]=new double[5];
    }

Q_Werte[0][0]=20394.0;
Q_Werte[0][1]=-24908.0;
Q_Werte[0][2]=-2026.0;
Q_Werte[0][3]=3896.0;
Q_Werte[0][4]=658.0;

Q_Werte[1][0]=-24908.0;
Q_Werte[1][1]=41818.0;
Q_Werte[1][2]=-3466.0;
Q_Werte[1][3]=-9828.0;
Q_Werte[1][4]=-372.0;

Q_Werte[2][0]=-2026.0;
Q_Werte[2][1]=-3466.0;
Q_Werte[2][2]=3510.0;
Q_Werte[2][3]=2178.0;
Q_Werte[2][4]=-348.0;

Q_Werte[3][0]=3896.0;
Q_Werte[3][1]=-9828.0;
Q_Werte[3][2]=2178.0;
Q_Werte[3][3]=3030.0;
Q_Werte[3][4]=-44.0;

Q_Werte[4][0]=658.0;
Q_Werte[4][1]=-372.0;
Q_Werte[4][2]=-348.0;
Q_Werte[4][3]=-44.0;
Q_Werte[4][4]=54.0;
```

```
Werte[0][0]=-1.0;
Werte[0][1]=-1.0;
Werte[0][2]=-1.0;
Werte[0][3]=-1.0;
Werte[0][4]=-1.0;

Werte[1][0]=10.0;
Werte[1][1]=10.0;
Werte[1][2]=-3.0;
Werte[1][3]=5.0;
Werte[1][4]=4.0;

Werte[2][0]=-8.0;
Werte[2][1]=1.0;
Werte[2][2]=-2.0;
Werte[2][3]=-5.0;
Werte[2][4]=-.0;

Werte[3][0]=8.0;
Werte[3][1]=-1.0;
Werte[3][2]=2.0;
Werte[3][3]=5.0;
Werte[3][4]=-3.0;

Werte[4][0]=-4.0;
Werte[4][1]=-2.0;
Werte[4][2]=3.0;
Werte[4][3]=-5.0;
Werte[4][4]=1.0;

b[0].Erzeugung_double(-5.0,t,q);
b[1].Erzeugung_double(20.0,t,q);
b[2].Erzeugung_double(-40.0,t,q);
b[3].Erzeugung_double(11.0,t,q);
b[4].Erzeugung_double(-30.0,t,q);

c[0].Erzeugung_double(18340.0,t,q);
c[1].Erzeugung_double(-34198.0,t,q);
c[2].Erzeugung_double(4542.0,t,q);
c[3].Erzeugung_double(8672.0,t,q);
c[4].Erzeugung_double(86.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],1.0);
mpf_set_d(x_korrekt.V[1],2.0);
mpf_set_d(x_korrekt.V[2],-1.0);
mpf_set_d(x_korrekt.V[3],3.0);
mpf_set_d(x_korrekt.V[4],-4.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-14463.0);
#endif

Q.Erzeugung_double(Q_Werte);
A.Erzeugung_double(Werte);
```

```
Simplex_Startwert S(A.Zeilenzahl,A.Spaltenzahl,A.Zeilenzahl,n,t,q,q_1,k,f)
    ;

Vektor_LA C(S.Spaltenzahl-1,n,t,q,q_1,k,f);
for(i=C.d-1; i>=A.Spaltenzahl;i--){
    C[i].Erzeugung_double(1.0,t,q);
    }

Vektor_LA x0(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA X(A.Spaltenzahl+A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA B(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA N(S.Spaltenzahl-1,n,t,q,q_1,k,f);

clock_t start_simplex, stop_simplex;
clock_t start_Phase2, stop_Phase_2;

start_programm=clock();
start_simplex=clock();
int Anzahl_Simplex_Iterationen=S.Simplex_Algorithmus(A,b,C,x0,X,0,A.
    Zeilenzahl,B,N);
stop_simplex=clock();

A*=-1;
b*=-1;
x0.Kopiere_Vektor_oben(X);

///Ende Phase 1

Matrix_LA G(Q.Zeilenzahl+A.Spaltenzahl,Q.Spaltenzahl+A.Spaltenzahl,n,t,q,
    q_1,k,f);
Vektor_LA l(Q.Zeilenzahl+A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA lambda(b.d,n,t,q,q_1,k,f);
Vektor_LA lambda_min=lambda;
Vektor_LA lambda_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Share Bit(n,t,q);
Share Hilfe(n,t,q);
Share lambda_q (n,t,q);//Speichere das aktuelle lambda
Vektor_LA Hilfe_Vektor(A.Spaltenzahl,n,t,q,q_1,k,f);
Share alpha(n,t,q);
Vektor_LA Index(A.Zeilenzahl,n,t,q,q_1,k,f);//wird zur Bestimmung des
    Minimums von alpha benötigt
Vektor_LA Index_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Index_AM(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA ax(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_Zq(b.d,n,t,q,q_1,k,f);
Vektor_LA Kopie_AM(b.d,n,t,q,q_1,k,f);
Vektor_LA AM_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Matrix_LA Kopie_A=A;
Matrix_LA A_Kompakt(A.Spaltenzahl,A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Kopie_c=c;
Vektor_LA Kopie_x=x;
Vektor_LA Ergebnis=x;
Share Hilfe_2=Hilfe;
Matrix_LA P(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_t(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);//Die Transponierte
    von P
```

```
Share alpha_kleiner_eins(n,t,q);

///Beginn Phase 2
start_Phase2=clock();
x=x0;//x ist der in Phase 1 bestimmte Startwert

haupt::Bestimme_Aktive_Menge(A,x,b,AM_Zq,b.d,k,n,t,q,q_1);

for(i=0; i<A.Zeilenzahl; i++){//Bestimme die initialen Werte ax(i)
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(x,ax[i].Vector,Hilfe_Vektor.d);
    }

for(Iterationszaehler=1; Iterationszaehler<51; Iterationszaehler++){

Kopie_c=c;
Kopie_x=x;

Kopie_x.MatrixMult(Q,Kopie_x,Q.Spaltenzahl,c.d);
Kopie_x.TruncPr();
Kopie_c+=Kopie_x;//Kopie_c=Q*x+c

Kopie_A.Multipliziere_Zeilenweise(&AM_Zq.Vector_LA);
Kopie_A.Matrix_Zusammenschieben_binaer(AM_Zq.Vector_LA,P);
A_Kompakt.Oben_einpassen(Kopie_A);

P_t=P;
P_t.Transponiere_quadratisch();

AM_Zq.MatrixMult(P,Kopie_AM,P.Spaltenzahl,Kopie_AM.d);
AM_kurz.Kopiere_Vektor_oben(Kopie_AM);

G=0;
l=0;
Erstelle_Gleichungssystem(G,Q,A_Kompakt);
Erstelle_Vektor(l,Kopie_c,c.d,l.d);

#ifdef LR
G.Loese_GLS_QR(l.Vector_LA,l.d);
#endif

#ifdef QR
G.Loese_GLS_LR(l.Vector_LA,l.d);
#endif

j=0;
for(i=c.d; i<l.d; i++){
    lambda.Vector_LA.at(j)=l.Vector_LA.at(i);
    j++;
    }
P.Transponiere_quadratisch();
lambda.MatrixMult(P,lambda,P.Spaltenzahl,lambda.d);//Setze lambda an die
    richtigen Stellen zurück; die ungültigen Stellen werden automatisch
    mit Null besetzt

for(i=0; i<lambda_kurz.d; i++){
```

```
    lambda_kurz[i]=l[i+c.d]; //Kompaktifizierte Form des Vektors lambda
        erstellen
    lambda_kurz[i]=lambda_kurz[i]*AM_kurz[i]+(-AM_kurz[i]+1)*unendlich; //
        Ersetze nicht-aktive Komponenten mit unendlich; diese sind wg.
        Komplementärbedingung = 0
    }

for(i=0; i<x.d; i++){
    d.Vector_LA.at(i)=l.Vector_LA.at(i);
    }

d.Gleich_Null_approximativ(Bit);

///Schritt 3
Index_kurz=0;
lambda_kurz.Min(Index_kurz,k);

lambda_q=0;
lambda_q=lambda_kurz[Index_kurz];

///Verkürzten Vektor lambda wieder auf die richtige Länge expandieren
lambda_min.Oben_einpassen(Index_kurz);
lambda_min.MatrixMult(P_t,lambda_min,P_t.Spaltenzahl,lambda_min.d);

lambda_q.GTZ_approximativ(t,q,q_1,k);
lambda_q.MulPub(Bit,Ende);

if(mpz_cmp_si(Ende,1)==0){
    stop_programm=clock();
    stop_Phase_2=clock();
    Programmende(start_programm,stop_programm,x,x_korrekt,ff,f_korrekt,
        Iterationszaehler,t,q);
    return 0;
    }
///Schritt 2
//d auf Null setzen, falls der (appr.) Test dies ergibt; korrigiere so für
    Vektoren, die nur auf Grund von Rundungsfehlern!=0 sind
Hilfe=Bit;
Hilfe*=-1;
Hilfe+=1;
for(i=0; i<x.d; i++){
    d[i]=d[i]*Hilfe;
    }

Berechne_alpha(A,d,x,b,alpha,Index,AM_Zq,alpha_kleiner_eins,A.Zeilenzahl,n
    ,t,k,q,q_1);

Hilfe=Bit;
Hilfe*=-1;
Hilfe.Add_Konst_int(1,q); //Hilfe=1-(d=?0)
d*=alpha;
d*=Hilfe; //korrigiere d
d.TruncPr();
x+=d; //x_{k+1}=x_k+alpha_k*d_k

Hilfe=Bit;
```

```
Index*=alpha_kleiner_eins; //Index_AM=Index_AM*(alpha_k <?1)
Hilfe*=-1;
Hilfe.Add_Konst_int(1,q); //Hilfe=1-(d=?0)
Index_AM*=Hilfe; //Index=Index_AM*(alpha_k <?=1)*(1-(d=?0))
AM_Zq+=Index; //W_{k+1}=W_k+I*(alpha <?0)*(1-(d=?0))

lambda_min*=-1;
lambda_min*=Bit;
AM_Zq+=lambda_min; //W_{k+1}=W_k+I*(alpha <?0)*(1-(d=?0))-d*I_lambda_min

for(i=0; i<A.Zeilenzahl; i++){//Bestimme die neuen Werte ax(i)
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(x,ax[i].Vector,Hilfe_Vektor.d);
    }

Kopie_A=A;
Kopie_x=x;
Kopie_c=c;
}

return 0;
}
```

Berechnung der Schrittweite α

```
void Berechne_alpha(const Matrix_LA& A, const Vektor_LA& d, const
     Vektor_LA&x, Vektor_LA& b, Share& alpha, Vektor_LA& Index, Vektor_LA&
      AM_Zq, Share& alpha_kleiner_eins, int Dimension, int n, int t, int k
     , mpz_t& q, mpz_t& q_1){

int i;
Vektor_LA ax(Dimension, n, t, q, q_1, k, f);
Vektor_LA ad(Dimension, n, t, q, q_1, k, f);
Vektor_LA ad_Vergleich_Null(Dimension, n, t, q, q_1, k, f);
Vektor_LA Hilfe_Vektor(A.Spaltenzahl, n, t, q, q_1, k, f);
Vektor_LA eins_minus_AM_Zq=AM_Zq;
Vektor_LA Kriterium(Dimension, n, t, q, q_1, k, f);
Vektor_LA Kriterium_Komplement(Dimension, n, t, q, q_1, k, f);
Vektor_LA alpha_ad(Dimension, n, t, q, q_1, k, f);

mpz_t unendlich;
mpz_init_set_ui(unendlich,1);
mpz_mul_2exp(unendlich,unendlich,k-1);
mpz_sub_ui(unendlich,unendlich,1);

mpz_t minus_eins;
mpz_init_set(minus_eins,zwei_hoch_f);
mpz_mul_si(minus_eins,minus_eins,-1);


for(i=0; i<Dimension; i++){
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(x,ax[i].Vector,A.Spaltenzahl);
    ax[i].TruncPr(k,f,t,q,q_1);
    }

for(i=0; i<A.Zeilenzahl; i++){
```

```
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(d,ad[i].Vector,Hilfe_Vektor.d);

    ad[i].TruncPr(k,f,t,q,q_1);
    ad_Vergleich_Null[i]=ad[i];//Überprüfe, ob ad[i]<0 ist; die
        betreffenden Komponenten werden bei der Berechnung von alpha
        nicht betrachtet
    ad_Vergleich_Null[i].LTZ_approximativ(t,q,q_1,k);

    eins_minus_AM_Zq[i].Mul_Konst_int(-1,q);
    eins_minus_AM_Zq[i].Add_Konst_int(1,q);//eins_minus_AM_Zq[i]=1-AM_Zq[i
        ]

    Kriterium[i]=eins_minus_AM_Zq[i];
    Kriterium[i].AND(ad_Vergleich_Null[i]);
    Kriterium_Komplement[i]=Kriterium[i];
    Kriterium_Komplement[i]*=-1;
    Kriterium_Komplement[i]+=1;

    alpha_ad[i]=b[i]-ax[i];
    alpha_ad[i]=Kriterium[i]*alpha_ad[i]-Kriterium_Komplement[i]*unendlich
        ;
    ad[i]=Kriterium[i]*ad[i]+Kriterium_Komplement[i]*minus_eins;//nicht
        anwendbare Nenner auf Eins setzen
    }

alpha_ad.Min_Bruch(ad,Index);

Share Zaehler(n,t,q);
Share Nenner(n,t,q);

Zaehler=alpha_ad[Index];
Nenner=ad[Index];

alpha=Zaehler;
alpha.FPDiv(Nenner,t,q,q_1,k);//alpha=min((alpha_ad[i]-ax[i])/ad[i])

Zaehler=alpha;
Zaehler.LT(zwei_hoch_f,t,q,q_1,k);//Überprüfe, ob das Minimum größer 1 ist

alpha_kleiner_eins=Zaehler;
alpha=Zaehler*alpha+(-Zaehler+1)*zwei_hoch_f;

mpz_clear(minus_eins);
mpz_clear(unendlich);
}
```

Berechnung der Zielfunktion

```
void Zielfunktion(Matrix_LA& Q, Vektor_LA& x, Vektor_LA& c, mpf_t& Wert,
    int n, int t, int k, mpz_t& q, mpz_t& q_1){

Share Ergebnis(n,t,q);
Share Ergebnis2(n,t,q);
mpz_t z;
mpz_init(z);
```

```
Vektor_LA Kopie_x=x;
Kopie_x.MatrixMult(Q,Kopie_x,Q.Spaltenzahl,x.d);
Kopie_x.SkalarMult(x,Ergebnis.Vector,Kopie_x.d);
Ergebnis.TruncPr(2*k,2*f,t,q,q_1);
Ergebnis.Div_Konst_int(2,q,q_1,k);

x.SkalarMult(c,Ergebnis2.Vector,c.d);
Ergebnis2.TruncPr(k,f,t,q,q_1);

Ergebnis+=Ergebnis2;
Ergebnis.Rekonstrukt(z,t,q);

mpf_set_z(Wert,z);
mpf_div_2exp(Wert,Wert,f);

mpz_clear(z);
}
```

Aufstellen der Matrix zum Lösen des iterativen Gleichheitsproblems

```
void Erstelle_Gleichungssystem(Matrix_LA& M, Matrix_LA& Q, Matrix_LA& A){
//Erstelle das Gleichungssystem, dessen Lösung Lösung des
     Optimierungsproblems unter den Gleichheitsbedingungen ist

int i,j,l;

for(i=0; i<Q.Zeilenzahl; i++){//Linke obere Ecke mit Q befüllen
    for(j=0; j<Q.Spaltenzahl; j++){
        M.M[i][j]=Q.M[i][j];
        }
    l=0;
    for(j=Q.Spaltenzahl;j<Q.Spaltenzahl+A.Zeilenzahl; j++){//Rechte obere
         Ecke mit -A^t befüllen
        M.M[i][j]=-A.M[l][i];
        l++;
    }
}

l=0;
for(i=Q.Zeilenzahl; i<Q.Zeilenzahl+A.Zeilenzahl; i++){
    for(j=0; j<A.Spaltenzahl; j++){
        M.M[i][j]=A.M[l][j];
        }
    l++;
    }
}
```

Erstellen des zugehörigen Vektors

```
void Erstelle_Vektor(Vektor_LA& V, Vektor_LA& b, int m, int n){//m=dim b,
     n=dim *this; n>=m; benötigt für Optimierung
//Wird benötigt im Iterationsschritt der generischen aktiven Mengen
     Strategie
if(n<m || n<0 || m<0){
    printf("\nWarnung: Erstelle_Vektor");
    printf("\nDimensionen falsch. Erstellen des Vektors nicht möglich.
         Abbruch");
```

```
    exit(1);
    }

int i;

for(i=0; i<m; i++){
    V.Vector_LA.at(i)=b[i];
    V.Vector_LA.at(i)*=-1;//*this(i)=-b_i
    }
}
```

Berechnung der Aktiven Menge

```
void Bestimme_Aktive_Menge(Matrix_LA& A, Vektor_LA& x, Vektor_LA& b,
     Vektor_LA& AM_Zq, int d, int k, int n, int t, mpz_t& q, mpz_t& q_1){
     //Bestimmung der aktiven Menge

int i;
mpz_t z;
mpz_init(z);

Vektor_LA Hilfsvektor=x;
Hilfsvektor.MatrixMult(A,AM_Zq,A.Zeilenzahl,A.Spaltenzahl);
AM_Zq.TruncPr();
AM_Zq-=b;

#ifdef debug
AM_Zq.Ausgabe_skaliert("AM_Zq");
AM_Zq.Ausgabe("2^f*AM_Zq");
getchar();
#endif

mpz_set_ui(z,8);
for(i=0; i<d; i++){
AM_Zq[i].Quadrat();
AM_Zq[i].TruncPr(k,f,t,q,q_1);
AM_Zq[i].LT(z,t,q,q_1,k);
}

mpz_clear(z);
}
```

D.2 Die Implementierung des dualen Algorithmus

Die `main`-Funktion

```
int main()
{
int i,j,n=5, t=2, k=128, Iterationszaehler=1;
mpf_set_default_prec(300);

mpz_t q;
mpz_t q_1;
mpz_t basis;
mpz_t modulo;
mpz_t modulo_q1;
```

```
mpz_init(q);
mpz_init(q_1);
mpz_init_set_ui(basis,2);
mpz_init(modulo);
mpz_init(modulo_q1);

mpz_pow_ui(q,basis,Modullaenge);
mpz_pow_ui(q_1,basis,Modullaenge_kurz);

do{
mpz_nextprime(q,q);
mpz_nextprime(q_1,q_1);
mpz_mod_ui(modulo,q, 4);
mpz_mod_ui(modulo_q1,q_1, 4);
}
while(mpz_cmp_ui(modulo,3)!=0 || mpz_cmp_ui(modulo_q1,3)!=0); //Erzeuge
     geeignete Primzahl für Berechnung des modulus (muß kongruent 3 mod 4
     sein)

haupt::Berechne_Koeffizienten(n,t,k,f,q,q_1);

mpz_init_set(grosser_Modulus,q);
mpz_init_set(kleiner_Modulus,q_1);

mpz_t unendlich;
mpz_init(unendlich);

clock_t start, stop;

mpz_t seed_mpz;
mpz_init(seed_mpz);
time_t seed;
double seed_d;

seed=clock();
seed_d=(double) seed;
mpz_set_d(seed_mpz,seed_d);

gmp_randstate_t state;
gmp_randinit_default(state);
gmp_randseed(state,seed_mpz);

mpz_set_ui(unendlich,1);
mpz_mul_2exp(unendlich,unendlich,k-1);
mpz_sub_ui(unendlich,unendlich,1);

double ** Werte, ** Q_Werte;

#ifdef Problem_21
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);//ist hier =0
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,2,n,t,q,q_1,k,f);
```

```
Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=0.02;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=10.0;
Werte[0][1]=-1.0;
Werte[1][0]=-1.0;
Werte[1][1]=0.0;
Werte[2][0]=1.0;
Werte[2][1]=0.0;
Werte[3][0]=0.0;
Werte[3][1]=-1.0;
Werte[4][0]=0.0;
Werte[4][1]=1.0;

b.Vector_LA.at(0).Erzeugung_double(10.0,t,q);
b.Vector_LA.at(1).Erzeugung_double(-50.0,t,q);
b.Vector_LA.at(2).Erzeugung_double(2.0,t,q);
b[3].Erzeugung_double(-50.0,t,q);
b[4].Erzeugung_double(-50.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],2.0);
mpf_set_d(x_korrekt.V[1],0.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,0.04);
#endif

#ifdef Problem_35
Vektor_LA x(3,n,t,q,q_1,k,f);
Vektor_LA d(3,n,t,q,q_1,k,f);
Vektor_LA c(3,n,t,q,q_1,k,f);
Vektor_LA b(4,n,t,q,q_1,k,f);
Matrix_LA A(4,3,n,t,q,q_1,k,f);
Matrix_LA G(3,3,n,t,q,q_1,k,f);

Werte=new double *[4];
for(i=0; i<4; i++){
Werte[i]=new double[3];
}


Q_Werte=new double*[3];
for(i=0; i<5; i++){
    Q_Werte[i]=new double[3];
```

```
    }

Q_Werte[0][0]=4.0;
Q_Werte[0][1]=2.0;
Q_Werte[0][2]=2.0;

Q_Werte[1][0]=2.0;
Q_Werte[1][1]=4.0;
Q_Werte[1][2]=0.0;

Q_Werte[2][0]=2.0;
Q_Werte[2][1]=0.0;
Q_Werte[2][2]=2.0;


Werte[0][0]=-1.0;
Werte[0][1]=-1.0;
Werte[0][2]=-2.0;

Werte[1][0]=1.0;
Werte[1][1]=0.0;
Werte[1][2]=0.0;

Werte[2][0]=0.0;
Werte[2][1]=1.0;
Werte[2][2]=0.0;

Werte[3][0]=0.0;
Werte[3][1]=0.0;
Werte[3][2]=1.0;


c[0].Erzeugung_double(-8.0,t,q);
c[1].Erzeugung_double(-6.0,t,q);
c[2].Erzeugung_double(-4.0,t,q);

b[0].Erzeugung_double(-3.0,t,q);
b[1].Erzeugung_double(0.0,t,q);
b[2].Erzeugung_double(0.0,t,q);
b[3].Erzeugung_double(0.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],4.0);
mpf_div_ui(x_korrekt.V[0],x_korrekt.V[0],3);
mpf_set_d(x_korrekt.V[1],7.0);
mpf_div_ui(x_korrekt.V[1],x_korrekt.V[1],9);
mpf_set_d(x_korrekt.V[2],4.0);
mpf_div_ui(x_korrekt.V[2],x_korrekt.V[2],9);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-8.0);
mpf_div_ui(f_korrekt,f_korrekt,9);
mpf_sub_ui(f_korrekt,f_korrekt,8);//f=-8 8/9
#endif

#ifdef Problem_76
Vektor_LA x(4,n,t,q,q_1,k,f);
```

```
Vektor_LA d(4,n,t,q,q_1,k,f);
Vektor_LA c(4,n,t,q,q_1,k,f);
Vektor_LA b(7,n,t,q,q_1,k,f);
Matrix_LA A(7,4,n,t,q,q_1,k,f);
Matrix_LA G(4,4,n,t,q,q_1,k,f);

Werte=new double *[7];
for(i=0; i<7; i++){
Werte[i]=new double[4];
}


Q_Werte=new double*[4];
for(i=0; i<5; i++){
    Q_Werte[i]=new double[4];
    }

Q_Werte[0][0]=2.0;
Q_Werte[0][1]=0.0;
Q_Werte[0][2]=-1.0;
Q_Werte[0][3]=0.0;

Q_Werte[1][0]=0.0;
Q_Werte[1][1]=1.0;
Q_Werte[1][2]=0.0;
Q_Werte[1][3]=0.0;

Q_Werte[2][0]=-1.0;
Q_Werte[2][1]=0.0;
Q_Werte[2][2]=2.0;
Q_Werte[2][3]=1.0;

Q_Werte[3][0]=0.0;
Q_Werte[3][1]=0.0;
Q_Werte[3][2]=1.0;
Q_Werte[3][3]=1.0;


Werte[0][0]=-1.0;
Werte[0][1]=-2.0;
Werte[0][2]=-1.0;
Werte[0][3]=-1.0;

Werte[1][0]=-3.0;
Werte[1][1]=-1.0;
Werte[1][2]=-2.0;
Werte[1][3]=1.0;

Werte[2][0]=0.0;
Werte[2][1]=1.0;
Werte[2][2]=4.0;
Werte[2][3]=0.0;

Werte[3][0]=1.0;
Werte[3][1]=0.0;
Werte[3][2]=0.0;
Werte[3][3]=0.0;
```

```
Werte[4][0]=0.0;
Werte[4][1]=1.0;
Werte[4][2]=0.0;
Werte[4][3]=0.0;

Werte[5][0]=0.0;
Werte[5][1]=0.0;
Werte[5][2]=1.0;
Werte[5][3]=0.0;

Werte[6][0]=0.0;
Werte[6][1]=0.0;
Werte[6][2]=0.0;
Werte[6][3]=1.0;

c[0].Erzeugung_double(-1.0,t,q);
c[1].Erzeugung_double(-3.0,t,q);
c[2].Erzeugung_double(1.0,t,q);
c[3].Erzeugung_double(-1.0,t,q);

b[0].Erzeugung_double(-5.0,t,q);
b[1].Erzeugung_double(-4.0,t,q);
b[2].Erzeugung_double(1.5,t,q);
b[3].Erzeugung_double(0.0,t,q);
b[4].Erzeugung_double(0.0,t,q);
b[5].Erzeugung_double(0.0,t,q);
b[6].Erzeugung_double(0.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],3.0);
mpf_div_ui(x_korrekt.V[0],x_korrekt.V[0],11);//0,272727272727... =3/11
mpf_set_d(x_korrekt.V[1],23.0);
mpf_div_ui(x_korrekt.V[1],x_korrekt.V[1],11);//2,09090909090909... =23/11
mpf_set_d(x_korrekt.V[2],0.26);
mpf_div_ui(x_korrekt.V[2],x_korrekt.V[2],10000000000);
mpf_set_d(x_korrekt.V[3],6.0);
mpf_div_ui(x_korrekt.V[3],x_korrekt.V[3],11);//0,545454545454... =6/11

mpf_t f_korrekt;
mpf_init(f_korrekt);

mpf_t h;
mpf_init(h);

mpf_mul(f_korrekt,x_korrekt.V[0],x_korrekt.V[0]);
mpf_mul(h,x_korrekt.V[1],x_korrekt.V[1]);
mpf_div_ui(h,h,2);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[2],x_korrekt.V[2]);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[3],x_korrekt.V[3]);
mpf_div_ui(h,h,2);
mpf_add(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[0],x_korrekt.V[2]);
mpf_sub(f_korrekt,f_korrekt,h);
mpf_mul(h,x_korrekt.V[2],x_korrekt.V[3]);
```

```
mpf_add(f_korrekt,f_korrekt,h);
mpf_sub(f_korrekt,f_korrekt,x_korrekt.V[0]);
mpf_mul_ui(h,x_korrekt.V[1],3);
mpf_sub(f_korrekt,f_korrekt,h);
mpf_add(f_korrekt,f_korrekt,x_korrekt.V[2]);
mpf_sub(f_korrekt,f_korrekt,x_korrekt.V[3]);
#endif

#ifdef Problem_224
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);//ist hier =0
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(4,n,t,q,q_1,k,f);
Matrix_LA A(4,2,n,t,q,q_1,k,f);

Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=4.0;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=1.0;
Werte[0][1]=3.0;
Werte[1][0]=-1.0;
Werte[1][1]=-3.0;
Werte[2][0]=1.0;
Werte[2][1]=1.0;
Werte[3][0]=-1.0;
Werte[3][1]=-1.0;

b.Vector_LA.at(0).Erzeugung_double(0.0,t,q);
b.Vector_LA.at(1).Erzeugung_double(-18.0,t,q);
b.Vector_LA.at(2).Erzeugung_double(0.0,t,q);
b[3].Erzeugung_double(-8.0,t,q);

c[0].Erzeugung_double(-48.0,t,q);
c[1].Erzeugung_double(-40.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],4.0);
mpf_set_d(x_korrekt.V[1],4.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-304.0);
#endif

#ifdef Prioblem_268
Vektor_LA x(5,n,t,q,q_1,k,f);
```

```
Vektor_LA d(5,n,t,q,q_1,k,f);
Vektor_LA c(5,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,5,n,t,q,q_1,k,f);
Matrix_LA G(5,5,n,t,q,q_1,k,f);

Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[5];
}


Q_Werte=new double*[5];
for(i=0; i<5; i++){
    Q_Werte[i]=new double[5];
    }

Q_Werte[0][0]=20394.0;
Q_Werte[0][1]=-24908.0;
Q_Werte[0][2]=-2026.0;
Q_Werte[0][3]=3896.0;
Q_Werte[0][4]=658.0;

Q_Werte[1][0]=-24908.0;
Q_Werte[1][1]=41818.0;
Q_Werte[1][2]=-3466.0;
Q_Werte[1][3]=-9828.0;
Q_Werte[1][4]=-372.0;

Q_Werte[2][0]=-2026.0;
Q_Werte[2][1]=-3466.0;
Q_Werte[2][2]=3510.0;
Q_Werte[2][3]=2178.0;
Q_Werte[2][4]=-348.0;

Q_Werte[3][0]=3896.0;
Q_Werte[3][1]=-9828.0;
Q_Werte[3][2]=2178.0;
Q_Werte[3][3]=3030.0;
Q_Werte[3][4]=-44.0;

Q_Werte[4][0]=658.0;
Q_Werte[4][1]=-372.0;
Q_Werte[4][2]=-348.0;
Q_Werte[4][3]=-44.0;
Q_Werte[4][4]=54.0;


Werte[0][0]=-1.0;
Werte[0][1]=-1.0;
Werte[0][2]=-1.0;
Werte[0][3]=-1.0;
Werte[0][4]=-1.0;

Werte[1][0]=10.0;
Werte[1][1]=10.0;
Werte[1][2]=-3.0;
```

```
Werte[1][3]=5.0;
Werte[1][4]=4.0;

Werte[2][0]=-8.0;
Werte[2][1]=1.0;
Werte[2][2]=-2.0;
Werte[2][3]=-5.0;
Werte[2][4]=-.0;

Werte[3][0]=8.0;
Werte[3][1]=-1.0;
Werte[3][2]=2.0;
Werte[3][3]=5.0;
Werte[3][4]=-3.0;

Werte[4][0]=-4.0;
Werte[4][1]=-2.0;
Werte[4][2]=3.0;
Werte[4][3]=-5.0;
Werte[4][4]=1.0;


b[0].Erzeugung_double(-5.0,t,q);
b[1].Erzeugung_double(20.0,t,q);
b[2].Erzeugung_double(-40.0,t,q);
b[3].Erzeugung_double(11.0,t,q);
b[4].Erzeugung_double(-30.0,t,q);

c[0].Erzeugung_double(18340.0,t,q);
c[1].Erzeugung_double(-34198.0,t,q);
c[2].Erzeugung_double(4542.0,t,q);
c[3].Erzeugung_double(8672.0,t,q);
c[4].Erzeugung_double(86.0,t,q);

Vektor_mpf_t x_korrekt(x.d);
mpf_set_d(x_korrekt.V[0],1.0);
mpf_set_d(x_korrekt.V[1],2.0);
mpf_set_d(x_korrekt.V[2],-1.0);
mpf_set_d(x_korrekt.V[3],3.0);
mpf_set_d(x_korrekt.V[4],-4.0);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-14463.0);
#endif

#ifdef Problem_A
Vektor_LA x(2,n,t,q,q_1,k,f);
Vektor_LA d(2,n,t,q,q_1,k,f);
Vektor_LA c(2,n,t,q,q_1,k,f);
Matrix_LA G(2,2,n,t,q,q_1,k,f);
Vektor_LA b(5,n,t,q,q_1,k,f);
Matrix_LA A(5,2,n,t,q,q_1,k,f);


Werte=new double *[5];
for(i=0; i<5; i++){
Werte[i]=new double[2];
```

```
}

Q_Werte=new double*[2];
Q_Werte[0]=new double[2];
Q_Werte[1]=new double[2];

Q_Werte[0][0]=2.0;
Q_Werte[0][1]=0.0;
Q_Werte[1][0]=0.0;
Q_Werte[1][1]=2.0;

Werte[0][0]=1.0;
Werte[0][1]=-2.0;
Werte[1][0]=-1.0;
Werte[1][1]=-2.0;
Werte[2][0]=-1.0;
Werte[2][1]=2.0;
Werte[3][0]=1.0;
Werte[3][1]=0.0;
Werte[4][0]=0.0;
Werte[4][1]=1.0;

b[0].Erzeugung_double(-2.0,t,q);
b[1].Erzeugung_double(-6.0,t,q);
b[2].Erzeugung_double(-2.0,t,q);
b[3].Erzeugung_double(0.0,t,q);
b[4].Erzeugung_double(0.0,t,q);

c.Vector_LA.at(0).Erzeugung_double(-2.0,t,q);
c.Vector_LA.at(1).Erzeugung_double(-5.0,t,q);

Vektor_mpf_t x_korrekt(2);
mpf_set_d(x_korrekt.V[0],1.4);
mpf_set_d(x_korrekt.V[1],1.7);

mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-6.45);
#endif


G.Erzeugung_double(Q_Werte);
A.Erzeugung_double(Werte);

printf("\nDualer Algorithmus");
printf("\nParameter des Problems:");
G.Ausgabe_skaliert("G");
c.Ausgabe_skaliert("c");
A.Ausgabe_skaliert("A");
b.Ausgabe_skaliert("b");

mpz_t eins;
mpz_init_set_ui(eins,1);

Vektor_LA AM_Zq(b.d,n,t,q,q_1,k,f);
Vektor_LA Index_AM(b.d,n,t,q,q_1,k,f);
Vektor_LA NN(b.d,n,t,q,q_1,k,f);
Vektor_LA AM_kurz(G.Zeilenzahl,n,t,q,q_1,k,f);
```

```
Vektor_LA AM_kurz_alt(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_kurz_Arbeit(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_kurz_Arbeit_C(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_Arbeit(b.d,n,t,q,q_1,k,f);///Aktive Menge bei der ausgewählte
    , verletzte NB provisorisch hinzugefügt wird
Vektor_LA AM_Arbeit_C(b.d,n,t,q,q_1,k,f);///Komplement der temporären
    aktiven Menge

Share ff(n,t,q);

Vektor_LA s(b.d,n,t,q,q_1,k,f);
Vektor_LA S=s;///s wird noch bei der Berechnung von t_2 gebraucht
Share s_p(n,t,q);

Vektor_LA u(A.Zeilenzahl,n,t,q,q_1,k,f);//Vektor der Lagrange-
    Multiplikatoren
Vektor_LA u_plus(A.Zeilenzahl,n,t,q,q_1,k,f);//Vektor der Änderungen der
    Lagrange-Multiplikatoren

Matrix_LA Kopie_G=G;

Vektor_LA z(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA tz(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_Arbeit(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_GTZ(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_GTZ_C(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_gross(A.Zeilenzahl,n,t,q,q_1,k,f);//wird für die Berechnung
    der Lagrange-Multiplikatoren und ihre Zuordnung zu den einzelnen NBen
     benötigt.

Vektor_LA t_1_Vektor(A.Zeilenzahl,n,t,q,q_1,k,f);//Zur Bestimmung von t_1
    benötigt
Vektor_LA t_1_Index(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA t_1_Index_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA u_plus_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);

Matrix_LA B(A.Spaltenzahl,A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA n_plus_minus(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA n_p(A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA y(A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA Hilfsvektor(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Hilfsvektor2(A.Spaltenzahl,n,t,q,q_1,k,f);

Share V(n,t,q);
Share Hilfe(n,t,q);
Share qq(n,t,q);///Anzahl der aktiven Gleichungen
Share t_1(n,t,q);
Share t_1_2(n,t,q);
Share Norm_z(n,t,q);
Share t_2(n,t,q);
Share zn(n,t,q);
Share T(n,t,q);
Share T_Hilfe(n,t,q);
```

```
Share T_Vergleich_t_2(n,t,q);
Share T_Vergleich_t_1(n,t,q);
Share eins_minus_T_Vergleich_t1(n,t,q);
Share eins_minus_T_Vergleich_t2(n,t,q);
Share t_2_Vergleich_unendlich(n,t,q);//Wird in 1. Iteration noch nicht
    gebraucht
Share eins_minus_t_2_Vergleich_unendlich(n,t,q);

Share q_Vergleich_Null(n,t,q);
Share r_kleiner_Null(n,t,q);
Share Schalter_t_1(n,t,q);

Vektor_LA nn(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA Additionshilfe(A.Zeilenzahl,n,t,q,q_1,k,f);

Vektor_LA beta(x.d-1,n,t,q,q_1,k,f);

Matrix_LA P(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P2(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_t(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_alt(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);

Matrix_LA N(A.Spaltenzahl, A.Zeilenzahl,n,t,q,q_1,k,f);///Matrizen, die
    für die Berechnung von z und r benötigt werden
Matrix_LA J(G.Zeilenzahl,G.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA R(G.Zeilenzahl,G.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA RR(1,1,n,t,q,q_1,k,f);

Vektor_LA Nummerierung(A.Zeilenzahl,n,t,q,q_1,k,f);
mpz_t Zahl_mpz;
mpz_init(Zahl_mpz);
for(i=0; i<A.Zeilenzahl; i++){
    mpz_set_ui(Zahl_mpz,i+1);//Nummerierung startet bei 1; so wird bei
        nicht-besetzten Einträgen in Nummern_AM erkannt, dass keine
        Änderung vorzunehmen ist
    Nummerierung[i].Erzeugung(Zahl_mpz,t,q);//Für jede NB eine Nummer
        erzeugen
    }
Share Nummer(n,t,q);
Vektor_LA Nummern_AM(J.Spaltenzahl,n,t,q,q_1,k,f);

start=clock();

/**Schritt 0: Bestimme als Startwert das Minimum ohne Nebenbedingungen x=G
    ^{-1}c */

x=c;
Kopie_G.Loese_GLS_Cholesky(x.Vector_LA,x.d);
x*=-1;

c.SkalarMult(x,ff);
ff.TruncPr(k,f,t,q,q_1);
ff.Div_Konst_int(2,q,q_1,k); ///ff=1/2*c^tx

/*H=G^{-1} wird nicht bestimmt; ist in Kopie_Q implizit gespeichert.
    Aktive Menge ist leer*/
```

```
/** Schritt 1: Aktive Menge bestimmen*/

///Bestimme verletzte Nebenbedingungen
x.MatrixMult(A,s);
s.TruncPr();
s=s-b;///s=Ax-b

S=s;
for(i=0; i<s.d; i++)
    {
    S[i].LTZ_approximativ(t,q,q_1,k);
    V=V+S[i];//V zählt Anzahl der verletzten NBen
    }

i=V.EQZPub();
if(i==1){///Ist keine Nebenbedingung verletzt, Ende.
    stop=clock();
    Programmende(start,stop,x,x_korrekt,u,ff,f_korrekt,Iterationszaehler,t
        ,q);
    return 0;
    }

Iterationszaehler++;

#ifdef G_Norm
Max_G_Norm(A,Kopie_G,b,x,nn,S,s,NN,n,t,k,q,q_1,Iterationszaehler);//S: Die
    Stellen an denen eine auswählbare NB sitzt
#endif

#ifdef Euklid
Max_Eukl_Norm(A,b,x,nn,S,n,t,k,q,q_1);
#endif

for(i=0; i<A.Zeilenzahl; i++){
    for(j=0; j<A.Spaltenzahl; j++){
        n_p[j]=A.M[i][j]*nn[i]+n_p[j];
        }
    }

///*Berechne Schrittrichtung (Schritt 2a)*/

y=n_p;

y.VWSubs_mit_Div(Kopie_G);//In Tests war noch die alte Version enthalten
z.RWSubs(Kopie_G,y);

///*Berechne Schrittweite; im 1. Schritt ist t_1 immer gleich unendlich!*/
z.Gleich_Null_approximativ(Norm_z);
z.SkalarMult(n_p,zn);
zn.TruncPr(k,f,t,q,q_1);

s_p=s[nn];
s_p.FPDiv(zn,t,q,q_1,k);
s_p*=-1;
t_2=Norm_z*unendlich;
Hilfe=-Norm_z;
Hilfe+=1;
```

```
t_2+=Hilfe*s_p;
T=t_2;

///*Schritt 2c*/
T_Hilfe=T-unendlich;

T_Vergleich_t_2=T;
T_Vergleich_t_1=T;
eins_minus_T_Vergleich_t1=T_Vergleich_t_1;
eins_minus_T_Vergleich_t2=T_Vergleich_t_2;

i=T_Hilfe.EQZPub();
if(i==1){
    printf("\nNebenbedingungen widerspruechlich. Abbruch");
    stop=clock();
    Programmende(start,stop,x,x_korrekt,u,ff,f_korrekt,Iterationszaehler,t
        ,q);
    exit(1);
    }

t_2_Vergleich_unendlich=t_2;//Wird in 1. Iteration noch nicht gebraucht
eins_minus_t_2_Vergleich_unendlich=-t_2_Vergleich_unendlich+1;

tz=z;
tz*=T;
tz.TruncPr(); //tz=T*z
x=x+tz;

///Bestimme verletzte Nebenbedingungen
x.MatrixMult(A,s);
s.TruncPr();
s=s-b;//s=Ax-b

///Bestimme verletzte NB gleich nach Neuberechnung von x

S=s;//s wird noch bei der Berechnung von t_2 gebraucht
V=0;
for(i=0; i<s.d; i++)
    {
    S[i].LTZ_approximativ(t,q,q_1,k);
    V=V+S[i];//V zählt Anzahl der verletzten NBen
    }

i=V.EQZPub();

zn=zn*T*T;
zn.Div_Konst_int(2,q,q_1,2*k);//Division vor TruncPr ist teurer, aber
    genauer
zn.TruncPr(2*k,2*f,t,q,q_1);


ff+=zn;//f=f+tz^tn_p(0.5t+u); (u=0) in der ersten Iteration!!!

u_plus=nn*T;//Lagrange-Multiplikator der aktivierten Gleichung wird auf T
    gesetzt (u_plus ist am Anfang Null)
```

```
u=u_plus;

/** Aktive Menge aktualisieren*/
Additionshilfe=nn;
Additionshilfe+=AM_Zq;
AM_Zq=nn;

if(i==1){//Ist keine Nebenbedingung verletzt, Ende.
    stop=clock();
    Programmende(start,stop,x,x_korrekt,u,ff,f_korrekt,Iterationszaehler,t
        ,q);
    return 0;
    }

qq=1;

//Aktive Menge kompaktifizieren
/** N^{\ast},H aktualisieren!!!*/
P.Ersetze_Zeile(AM_Zq.Vector_LA,0,AM_Zq.d);//Das so definierte P schiebt
    als Permutationsmatrix die 1. aktivierte NB an die Stelle 1
AM_kurz[0].Erzeugung(eins,t,q);//Nach dem 1. Schritt is genau eine NB
    aktiv; im verkürzten Vektor AM_kurz steht diese an Stelle 0

int check;
check=Berechne_J(J,RR,A,B,Kopie_G,nn,n,t,k,q,q_1,Iterationszaehler);//nn
    statt AM_Zq, da im ersten Schritt nur eine NB aktiv ist

AM_Arbeit=AM_Zq;///Aktive Menge bei der ausgewählte, verletzte NB
    provisorisch hinzugefügt wird
AM_Arbeit_C=AM_Arbeit;///Komplement der temporären aktiven Menge

for(Iterationszaehler=3; Iterationszaehler<50; Iterationszaehler++){

///Bestimme verletzte Nebenbedingungen - Verschoben ans Ende; nach der
    Neuberechnung

#ifdef G_Norm
Max_G_Norm(A,Kopie_G,b,x,nn,S,s,NN,n,t,k,q,q_1,Iterationszaehler);
#endif

#ifdef Euklid
Max_Eukl_Norm(A,b,x,nn,S,n,t,k,q,q_1);
#endif

#ifdef FindFirst
S.FindFirst(nn);
#endif

AM_Arbeit=AM_Zq+nn;
for(i=0; i<AM_Arbeit.d; i++){
    AM_Arbeit_C[i]=AM_Arbeit[i];
    AM_Arbeit_C[i]*=-1;
    AM_Arbeit_C[i]+=1;
    }//Bestimmt Komplement der temporären aktiven Menge

Index_AM=AM_Arbeit;
```

```
Index_AM.Vektor_Zusammenschieben_binaer(AM_Arbeit.Vector_LA,P2);//
     Aktualisiere die Matrix P, mit deren Hilfe die aktiven Ungleichungen
     kompaktifziert werden

AM_kurz_Arbeit.Oben_einpassen(Index_AM);

for(i=0; i<AM_kurz_Arbeit.d; i++){
     AM_kurz_Arbeit_C[i]=AM_kurz_Arbeit[i];
     AM_kurz_Arbeit_C[i]*=-1;
     AM_kurz_Arbeit_C[i]+=1;
     }

n_p=0;
for(i=0; i<A.Zeilenzahl; i++){
     for(j=0; j<A.Spaltenzahl; j++){
          n_p[j]=A.M[i][j]*nn[i]+n_p[j];
          }
     }
Nummer=Nummerierung[nn];//Nummer der NB auslesen

/** Berechne Schrittrichtung (Schritt 2a)*/

int v=haupt::min(Iterationszaehler,G.Zeilenzahl);
Matrix_LA R_klein(v-1,v-1,n,t,q,q_1,k,f);//Größe hängt von Iterationszahl
     ab; muß deshalb in jeder Runde neu erzeugt werden
R_klein.Links_oben_einpassen(R);
if(Iterationszaehler==3){
     check=Berechne_z_r(J,RR,n_p,AM_kurz,z,r,n,t,q,q_1,k,Iterationszaehler)
          ;
     }
else
     check=Berechne_z_r(J,R_klein,n_p,AM_kurz,z,r,n,t,q,q_1,k,
          Iterationszaehler);
/** Berechne Schrittweite; im 1. Schritt ist t_1 immer gleich unendlich!*/

z.Gleich_Null_approximativ(Norm_z);

z.SkalarMult(n_p,zn);
zn.TruncPr(k,f,t,q,q_1);
Hilfe=zn;

/** Berechne t_1*/
q_Vergleich_Null=qq;
q_Vergleich_Null.EQZ(t,q,q_1,k);
r_kleiner_Null=r[0];
r_kleiner_Null.GTZ_approximativ(t,q,q_1,k);
r_GTZ[0]=r_kleiner_Null;
for(i=1; i<r.d; i++){
     r_GTZ[i]=r[i];
     r_GTZ[i].GTZ_approximativ(t,q,q_1,k);
     r_kleiner_Null=r_kleiner_Null+r_GTZ[i]-r_kleiner_Null*r_GTZ[i];//
          r_kleiner_Null ist 1, wenn mindestens eine Komponente größer als
          0 ist
     }

Schalter_t_1=r_kleiner_Null*(-q_Vergleich_Null+1);
```

```
r_gross=0;
r_gross.Oben_einpassen(r);

P_t=P;
P_t.Transponiere_quadratisch();

r_gross.MatrixMult(P_t,r_gross);//Ordne die r's ihren Indizes zu

t_1_Vektor=u_plus;
t_1_Vektor.MatrixMult(P,t_1_Vektor);//permutiere die Einträge von u^+ an
     die oberen Stellen

u_plus_kurz.Oben_einpassen(t_1_Vektor);

for(i=0; i<r.d; i++){
    r_GTZ_C[i]=r_GTZ[i];
    r_GTZ_C[i]*=-1;
    r_GTZ_C[i]+=1;
    }

for(i=0; i<A.Spaltenzahl; i++){// TODO (LocalAdmin#1#): Kriteriumsvektor
     verwenden!
    u_plus_kurz[i]=u_plus_kurz[i]*AM_kurz_Arbeit[i]+AM_kurz_Arbeit_C[i]*
         unendlich;
    u_plus_kurz[i]=r_GTZ[i]*u_plus_kurz[i]+r_GTZ_C[i]*unendlich;
    r_Arbeit[i]=r[i]*AM_kurz_Arbeit[i]+AM_kurz_Arbeit_C[i]*zwei_hoch_f;//
         r_gross wird später noch für die Aktualisierung von u benötigt
    r_Arbeit[i]=r_Arbeit[i]*r_GTZ[i]+r_GTZ_C[i]*zwei_hoch_f;
    }

u_plus_kurz.Min_Bruch(r_Arbeit,t_1_Index_kurz);
t_1_2=u_plus_kurz[t_1_Index_kurz];

t_1_2.FPDiv(r_Arbeit[t_1_Index_kurz],t,q,q_1,k);
t_1=t_1_2;

t_1_Index=0;
t_1_Index.Oben_einpassen(t_1_Index_kurz);
t_1_Index.MatrixMult(P_t,t_1_Index);//Index von t_1 an die richtige Stelle
     permutieren

/** Berechne t_2*/
s_p=s[nn];
s_p.FPDiv(zn,t,q,q_1,k);
s_p*=-1;
t_2=Norm_z*unendlich;
Hilfe=-Norm_z;
Hilfe+=1;

t_2+=Hilfe*s_p;

T.min(t_1,t_2,q,q_1,t,k,Hilfe);

/**Schritt 2c*/
T_Hilfe=T;
T_Hilfe-=unendlich;
i=T_Hilfe.EQZPub();
```

```
if(i==1){
    stop=clock();
    Programmende(start,stop,x,x_korrekt,u,ff,f_korrekt,Iterationszaehler,t
        ,q);
    printf("\nNebenbedingungen widerspruechlich. Abbruch");
    exit(1);
    }

t_2_Vergleich_unendlich=t_2;
t_2_Vergleich_unendlich.EQ_mpz(unendlich,t,q,q_1,k);//t_2=?unendlich
eins_minus_t_2_Vergleich_unendlich=-t_2_Vergleich_unendlich;
eins_minus_t_2_Vergleich_unendlich+=1;

T_Vergleich_t_1=Hilfe;//Hilfe==1 <=> T=t_1, sonst 0
T_Vergleich_t_2=-Hilfe+1;

eins_minus_T_Vergleich_t1=T_Vergleich_t_1;
eins_minus_T_Vergleich_t2=T_Vergleich_t_2;
eins_minus_T_Vergleich_t1*=-1;
eins_minus_T_Vergleich_t1+=1;
eins_minus_T_Vergleich_t2*=-1;
eins_minus_T_Vergleich_t2+=1;

///xNeu
tz=z;
tz*=T;
tz.TruncPr(); //tz=T*z

x+=tz;
x*=eins_minus_t_2_Vergleich_unendlich;
Hilfsvektor=x*t_2_Vergleich_unendlich;
x+=Hilfsvektor;

///fNeu
zn*=T;//wird benötigt für die Aktualisierung von f
zn.TruncPr(k,f,t,q,q_1);
Hilfe=u_plus[nn];//Hilfe=u^+_{q+1}
Hilfe*=2;
Hilfe+=T;
Hilfe.Div_Konst_int(2,q,q_1,k);
zn*=Hilfe;
zn.TruncPr(k,f,t,q,q_1);

ff=eins_minus_t_2_Vergleich_unendlich*(ff+zn)+t_2_Vergleich_unendlich*ff;;
    //f=f+tz^tn_p(0.5t+u); (u=0) in der ersten Iteration!!!

///uNeu
u_plus=u_plus+nn*T;

r_gross*=T;
r_gross.TruncPr(k,f);

u_plus-=r_gross;//u^+=u^+t(-r 1)^t

u=u_plus*T_Vergleich_t_2+u*eins_minus_T_Vergleich_t2;//If T=t_2 ->u=u_plus
```

```
///Aktualisierung der aktiven Menge
Additionshilfe=nn+AM_Zq;

AM_Zq=AM_Zq*eins_minus_T_Vergleich_t2+Additionshilfe*T_Vergleich_t_2;//
     neuen Index zur aktiven Menge hinzufügen

Additionshilfe=AM_Zq-t_1_Index;
AM_Zq=AM_Zq*eins_minus_T_Vergleich_t1+Additionshilfe*T_Vergleich_t_1;

qq=(qq+1)*T_Vergleich_t_2+(qq-1)*eins_minus_T_Vergleich_t2; //
     Aktualisierung des Zählers der aktiven Menge

///Überprüfen der Zulässigkeit des neuen Punkts

x.MatrixMult(A,s);
s.TruncPr();
s=s-b;//s=Ax-b

S=s;//s wird noch bei der Berechnung von t_2 gebraucht
V=0;
for(i=0; i<s.d; i++)
     {
     S[i].LTZ_approximativ(t,q,q_1,k);
     V=V+S[i];//V zählt Anzahl der verletzten NBen
     }

i=V.EQZPub();
if(i==1){//Ist keine Nebenbedingung verletzt, Ende.
     stop=clock();
     Programmende(start,stop,x,x_korrekt,u,ff,f_korrekt,Iterationszaehler,t
          ,q);
     break;
     }

/** N^{\ast},H aktualisieren!!!*/
int check;

Index_AM=AM_Zq;
P_alt=P;
Index_AM.Vektor_Zusammenschieben_binaer(AM_Zq.Vector_LA,P);
AM_kurz_alt=AM_kurz;
AM_kurz.Oben_einpassen(Index_AM);

Matrix_LA R_gross(v,v,n,t,q,q_1,k,f);
if(A.Spaltenzahl<=3){//Bei Größe 3 müsste bei Aktualisierung von Matrix
     der Größe 2 UND 3 berechnet werden. Da ist Neuberechnung billiger
     check=Berechne_J(J,R_gross,A,B,Kopie_G,AM_Zq,n,t,k,q,q_1,
          Iterationszaehler);
     }
else{
     if(Iterationszaehler==3)
          J.Ausgabe_skaliert("J");
     Hilfe=qq-1;//qq ist um eins größer als die Spalte, die es indiziert
     Hilfsvektor.BitMask(Hilfe);//HV ist Bitvektorkodierung der Größe der
          aktiven Menge
```

```
    t_1_Index.MatrixMult(P_alt,Additionshilfe);//Additionshilfe=P*
        t_1_Index; multipliziere t_1_Index an eine vordere Position;
        verwende P_alt, da neues P auf t_1_I nicht anspricht

    Hilfsvektor2.Oben_einpassen(Additionshilfe);//AH ist einer der ersten
        Einträge
    Hilfsvektor=Hilfsvektor*T_Vergleich_t_2;//HV ist Bitvektorkodierung
        von qq, wenn neue NB hinzugefügt wird (dann steht die Spalte - qq
         - an der sie in J eingetragen wir, fest)
    Hilfe=-T_Vergleich_t_2+1;
    Hilfsvektor+=(Hilfsvektor2*Hilfe);//If T=t_2 -> HV<-HV else HV<-HV2

    n_plus_minus=(n_p*T_Vergleich_t_2);//T!=t_2 ->n+- = 0 und somit wird
        in B die 0-Spalte substituiert

    AM_kurz_Arbeit.Oben_einpassen(Additionshilfe);
    AM_kurz_Arbeit=AM_kurz_alt-AM_kurz_Arbeit;//Aus alter, verkürzter AM
        gelöschten Index entfernen
    AM_kurz_Arbeit=AM_kurz*T_Vergleich_t_2+AM_kurz_Arbeit*T_Vergleich_t_1;
        //neue verkürzte AM übergeben, wenn NB hinzugefügt wurde; sonst
        manipulierte alte, verkürzte AM
    Nummer=Nummer*T_Vergleich_t_2+t_1_Index[Nummerierung]*T_Vergleich_t_1;
        //Index der zu löschenden Spalte übergeben, wenn T=t_1
    T_Vergleich_t_1.Rekonstrukt_Ausgabe_unskaliert("T=?t_1");
    check=Aktualisiere_J(J,R_gross,A,Kopie_G,B,P,n_plus_minus,Nummer,
        Nummern_AM,Hilfsvektor,AM_kurz_Arbeit,n,t,k,q,q_1,
        Iterationszaehler);
    }
R.Links_oben_einpassen(R_gross);
}


return 0;
}
```

Die Berechnung von z und r

```
int Berechne_z_r(Matrix_LA& J, Matrix_LA& R, Vektor_LA& n_p, Vektor_LA&
    Aktive_Menge_kurz, Vektor_LA& z, Vektor_LA& r, int n, int t, mpz_t& q
    , mpz_t& q_1, int k, int Iteration){

Matrix_LA J_1=J;
Matrix_LA J_2=J;
Matrix_LA J_t=J;
Vektor_LA d(J.Spaltenzahl, n, t, q, q_1, k, f);

J_t.Transponiere_quadratisch();
n_p.MatrixMult(J_t,d,J_t.Zeilenzahl,n_p.d);
d.TruncPr();//D=J^t*n_p

Vektor_LA d_1=d;
d_1.Multipliziere_Zeilenweise(&Aktive_Menge_kurz);

J_1.Multipliziere_Spaltenweise(Aktive_Menge_kurz.Vector_LA);
J_2=J-J_1;

/// Explizite Berechnung von H nicht nötig
```

```
d.MatrixMult(J_2,z,J_2.Zeilenzahl,d.d);
z.TruncPr(k,f);

Vektor_LA D_1(R.Zeilenzahl,n,t,q,q_1,k,f);
D_1.Kopiere_Vektor_oben(d_1);
if(Iteration>3){
    r=0;
    Vektor_LA r_klein(D_1.d,n,t,q,q_1,k,f);
    r_klein.Kopiere_Vektor_oben(r);

    r_klein.Rueckwaertssubstitution_SMPC(R,D_1);
    r.Kopiere_Vektor_oben(r_klein);
    }
else{//Für die erste Berechnung von r wird nur eine Division benötigt
    d_1[0].FPDiv(R.M[0][0],t,q,q_1,k);
    r[0]=d_1[0];
    }

return 0;
}
```

Die Aktualisierung von J

```
int Aktualisiere_J(Matrix_LA& J, Matrix_LA& R, Matrix_LA& A, Matrix_LA& G,
      Matrix_LA& B, Matrix_LA& P, Vektor_LA& n_plus_minus, Share&
     Nummer_NB, Vektor_LA& Nummern_AM, Vektor_LA& Indikatorvektor,
     Vektor_LA& AM_kurz, int n, int t, int k, mpz_t& q, mpz_t& q_1, int
     Iteration){//Neuberechnung der Matrizen
//in G ist eine Cholesky-Zerlegung gespeichert
//Ab dim(G)=4 ist es billiger B zu aktualisieren
int i,j,r;

if(Iteration<A.Spaltenzahl){
    r=Iteration;
    }
else{
    r=A.Spaltenzahl;
    }
Matrix_LA B_Hilfe(A.Spaltenzahl,r,n,t,q,q_1,k,f);

Vektor_LA Hilfsvektor=n_plus_minus;//Berechne B=L^{-1}N
Vektor_LA x(G.Zeilenzahl,n,t,q,q_1,k,f);

Hilfsvektor.VWSubs_mit_Div(G);

///Schaltervariablen
Share Hilfe(n,t,q);
Share Hilfe2(n,t,q);
Share Index_Gleich(n,t,q);

Share Minus_Hilfe(n,t,q);
Share Schalter(n,t,q);
Share Nummer_Gleich_Null(n,t,q);
mpz_t eins;
mpz_init_set_ui(eins,1);
Schalter.Erzeugung(eins,t,q);
```

```
Vektor_LA HV(B.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA HV2(B.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Nullvektor(B.Spaltenzahl,n,t,q,q_1,k,f);
int Groesse_Spalten=ceil(log((double)B.Spaltenzahl)/log(2.0));//kleinste 2
    er-Potenz größer als B.Spaltenzahl

for(i=0; i<r; i++){//i<B.Spaltenzahl ??
    Hilfe=Nummer_NB;
    Nummer_Gleich_Null=Nummern_AM[i];
    Index_Gleich=Nummern_AM[i];
    Nummer_Gleich_Null.EQZ(t,q,q_1,k);//Überprüfe, ob aktuelle Nummer des
        Index gleich 0; in diesem Fall wird die erste solche Spalte mit
        der neuen Spalte besetzt; die anderen bleiben unverändert
    Index_Gleich.EQ(Nummer_NB,t,q,q_1,k);//Übergebener Index gleich
        aktuellem Index der AM?
    Hilfe.LQ(Nummern_AM[i],t,q,q_1,k);//Überprüfe, ob Index der neuen NB
        <= einem vorhandenen ist ("=" kommt bei Entfernen der NB zum
        Tragen)
    Hilfe2=Schalter*Nummer_Gleich_Null;
    Hilfe.OR(Hilfe2);
    Minus_Hilfe=-Hilfe+1;

    ///Spalte und Nummerierung ersetzen
    HV.getSpaltenvektor(B,i);//i-ten Spaltenvekor auslesen
    HV2=Hilfsvektor*Hilfe+HV*Minus_Hilfe;
    B.Ersetze_Spalte(HV2.Vector_LA,i,HV2.d);//Neuen Vektor in Spalte
        schreiben, WENN Index kleiner ist, als der des vorherigen
        Eintrags
    Hilfe2=Nummern_AM[i];//Merke den aktuellen Index; er wird für die
        Aktualisierung von Nummer_NB benötigt.
    Nummern_AM[i]=Nummer_NB*Hilfe+Nummern_AM[i]*(-Hilfe+1);//Nummerierung
        der Vektoren einsetzen

    ///Übergebenen Vektor mit Index erstellen
    Nummer_NB=Hilfe2*Hilfe*(-Index_Gleich+1)+Nummer_NB*(-Hilfe+1)+
        Index_Gleich*(A.Zeilenzahl+1);//Merke die Nummer des gemerkten (d
        .h. in die nächste Iteration hinübergetragenen Vektors) Vektors;
        wenn eine spalte gelöscht wird, wird Nummer_NB auf A.Zeilenzahl+1
         gesetzt: auf diese Weise werden die folgenden Spalten nicht
        verändert und der Inhalt von Hilfsvektor spielt keine Rolle
    Hilfsvektor=HV*Hilfe+Hilfsvektor*Minus_Hilfe;//Merke den nicht
        geschriebenen Vektor für die nächste Iteration; er kann dort
        eingesetzt werden
    Hilfe2=-Index_Gleich+1;
    Hilfsvektor=Hilfsvektor*Hilfe2+Nullvektor*Index_Gleich;//bei Streichen
         einer Spalte den an die nächste Iteration übergebenen Vektor auf
         0 setzen
    Schalter=Schalter*(-Nummer_Gleich_Null+1);//Schalter umlegen, wenn
        Nummer gleich 0
    }

Matrix_LA PP(B.Spaltenzahl,B.Spaltenzahl,n,t,q,q_1,k,f);
B.Matrix_spaltenweise_zusammenschieben_binaer(AM_kurz.Vector_LA,PP);//
    falls Spalte gelöscht wurde, muß Matrix kompaktifiziert werden
Nummern_AM.MatrixMult(PP,Nummern_AM);
```

```
Matrix_LA Merke_B=B;//B merken
if(r<A.Spaltenzahl)
    B_Hilfe.Kopiere_links(B);//QR_householder_neu_Rechtecksmatrix benötigt
        kleinere Matrix

Vektor_LA beta(B.Zeilenzahl-1,n,t,q,q_1,k,f);
Vektor_LA b(B.Zeilenzahl-1,n,t,q,q_1,k,f);

if(r!=A.Spaltenzahl){
    B_Hilfe.QR_Householder_neu_Rechtecksmatrix(b.Vector_LA,beta.Vector_LA)
        ;
    B.Kopiere_links(B_Hilfe);
    }
else
    B.QR_Householder_neu(b.Vector_LA,beta.Vector_LA);

R=0;
R.Links_oben_einpassen(B);

Matrix_LA Q(B.Zeilenzahl,B.Zeilenzahl,n,t,q,q_1,k,f);
for(i=1; i<B.Zeilenzahl; i++){//Kopiere die Householder Vektoren aus B in
     die Matrix Q, da die Funktion Berechne_Q eine quadratische Matrix im
     Scope voraussetzt
    for(j=0; j<i; j++){
        if(j<r){
            Q.M[i][j]=B.M[i][j];
            }
        }
    }

if(r!=A.Spaltenzahl)
    Q.Berechne_Q_neu(b.Vector_LA,beta.Vector_LA,Iteration);
else
    Q.Berechne_Q_neu(b.Vector_LA,beta.Vector_LA);

for(i=0; i<G.Spaltenzahl; i++){
    Hilfsvektor.getSpaltenvektor(Q,i);
    x.RWSubs(G,Hilfsvektor);
    J.Ersetze_Spalte(x.Vector_LA,i,x.d);
    }

B=Merke_B;
return 0;//Methode korrekt durchlaufen
}
```

Auswahl der verletzten Nebenbedingung mit Hilfe der *G*-Norm-Strategie

```
void Max_G_Norm(Matrix_LA& A, Matrix_LA& G, Vektor_LA& b, Vektor_LA& x,
     Vektor_LA& I, Vektor_LA& S, Vektor_LA& s, Vektor_LA& N, int n, int t,
      int k, mpz_t& q, mpz_t& q_1, int Iterationszaehler){

int i;

mpz_t minus_unendlich;
mpz_init_set_ui(minus_unendlich,1);
```

```
mpz_mul_2exp(minus_unendlich,minus_unendlich,k-1);
mpz_mul_si(minus_unendlich,minus_unendlich,-1);
mpz_add_ui(minus_unendlich,minus_unendlich,1);//minus_unendlich=-2^{k-1}+1

Vektor_LA Z(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA Kopie_N=N;
Vektor_LA Kopie_b=b;
Vektor_LA HV(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA ZV(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Lsg(A.Spaltenzahl,n,t,q,q_1,k,f);

Share Hilfe_Z(n,t,q);
for(i=0; i<A.Zeilenzahl; i++){

    Z[i]=s[i];
    Z[i].Quadrat();
    Z[i].TruncPr(k,f,t,q,q_1);

    if(Iterationszaehler<=2){
        ZV.getZeilenvektor(A,i);
        HV=ZV;
        HV.Vorwaertssubstitution(G);
        Lsg.Rueckwaertssubstitution_SMPC(G,HV);
        ZV.SkalarMult(Lsg,N[i].Vector,A.Spaltenzahl); //Der jeweilige
            Nenner ist c^tG^{-1}c (G-Norm-Strategie; vgl. Korn)
        N[i].TruncPr(k,f,t,q,q_1);
        Kopie_N[i]=N[i];
        }

    Hilfe_Z=-S[i]+1;
    Z[i]=S[i]*Z[i]+Hilfe_Z*minus_unendlich;//Ersetze bereits aktive NBen
        durch -unendlich im Zähler.
    Kopie_N[i]=S[i]*N[i]-Hilfe_Z*zwei_hoch_f;//Ersetze Nenner durch 1,
        falls die NB bereits aktiv ist.
    }

Z.Max_Bruch(Kopie_N,I);

mpz_clear(minus_unendlich);
}
```

Auswahl der verletzen Nebenbedingungen mit Hilfe der Euklidischen Norm-Stratgie

```
void Max_Eukl_Norm(Matrix_LA& A, Vektor_LA& b, Vektor_LA& x, Vektor_LA& I,
      Vektor_LA& AM, int n, int t, int k, mpz_t& q, mpz_t& q_1){

int i;

mpz_t minus_unendlich;
mpz_init_set_ui(minus_unendlich,1);
mpz_mul_2exp(minus_unendlich,minus_unendlich,k-1);
mpz_mul_si(minus_unendlich,minus_unendlich,-1);
mpz_add_ui(minus_unendlich,minus_unendlich,1);//minus_unendlich=-2^{k-1}+1

Vektor_LA Z(A.Zeilenzahl,n,t,q,q_1,k,f);
```

```
Vektor_LA N(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA ZV(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Kopie_b=b;

Share Hilfe_Z(n,t,q);

for(i=0; i<A.Zeilenzahl; i++){

    ZV.getZeilenvektor(A,i);
    ZV.SkalarMult(x,Z[i].Vector,A.Spaltenzahl);
    Kopie_b[i].Mul_Konst(zwei_hoch_f,q);
    Z[i]-=Kopie_b[i];
    Z[i].Quadrat();
    Z[i].TruncPr(3*k,3*f,t,q,q_1);

    ZV.SkalarMult(ZV,N[i].Vector,A.Spaltenzahl); //Der jeweilige Nenner
        ist c^tG^{-1}c (G-Norm-Strategie; vgl. Korn)
    N[i].TruncPr(k,f,t,q,q_1);

    Hilfe_Z=-AM[i]+1;
    Z[i]=AM[i]*Z[i]+Hilfe_Z*minus_unendlich;//Ersetze bereits aktive NBen
        durch unendlich im Zähler.
    N[i]=AM[i]*N[i]+(Hilfe_Z*zwei_hoch_f)*zwei_hoch_f;//Ersetze Nenner
        durch 1, falls die NB bereits aktiv ist.
    }

Z.Max_Bruch(N,I);

mpz_clear(minus_unendlich);
}
```

Auswahl der verletzten Nebenbedingung mit der Residuumsstrategie

```
void Max_Residuum(Matrix_LA& A, Vektor_LA& b, Vektor_LA& x, Vektor_LA& I,
     int n, int t, int k, mpz_t& q, mpz_t& q_1){

int i,m=A.Spaltenzahl;
Vektor_LA nn(I.d,n,t,q,q_1,k,f);
Vektor_LA HV(A.Spaltenzahl,n,t,q,q_1,k,f);

for(i=0; i<A.Zeilenzahl; i++){//
    HV.getZeilenvektor(A,i);
    HV.SkalarMult(x,nn[i].Vector,m);
    nn[i].TruncPr(k,f,t,q,q_1);
    nn[i]-=b[i];
    }

nn.Min(I,A.Zeilenzahl);
}
```

E Die SVM-Routinen

E.1 Verwendung des primalen Algorihmus

```
int main()
{
int i,j,n=5, t=2, KK=128, Iterationszaehler=1;
mpf_set_default_prec(300);

mpz_t s, Ende;

clock_t start, stop;

mpz_init(Ende);
mpz_init_set_ui(s,1);

mpz_t q;
mpz_t q_1;
mpz_t basis;
mpz_t modulo;
mpz_t modulo_q1;

mpz_init(q);
mpz_init(q_1);
mpz_init_set_ui(basis,2);
mpz_init(modulo);
mpz_init(modulo_q1);

mpz_pow_ui(q,basis,Modullaenge);
mpz_pow_ui(q_1,basis,63);

do{
mpz_nextprime(q,q);
mpz_nextprime(q_1,q_1);
mpz_mod_ui(modulo,q, 4);
mpz_mod_ui(modulo_q1,q_1, 4);
}
while(mpz_cmp_ui(modulo,3)!=0 || mpz_cmp_ui(modulo_q1,3)!=0); //Erzeuge
     geeignete Primzahl für Berechnung des modulus (muß kongruent 3 mod 4
     sein)

haupt::Berechne_Koeffizienten(n,t,KK,f,q,q_1);

mpz_init_set(grosser_Modulus,q);
mpz_init_set(kleiner_Modulus,q_1);

int Zaehler_voller_Schritt=0, Zaehler_halber_Schritt=0;

mpz_t unendlich;
mpz_init_set_ui(unendlich,1);
mpz_mul_2exp(unendlich,unendlich,KK-2);
```

```
mpz_sub_ui(unendlich,unendlich,1);//unendlich=2^{k-1}-1

#ifdef leu
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0);
mpf_set_d(x_korrekt.V[1],0.0007037378410163503);
mpf_set_d(x_korrekt.V[2],2.738219115626705e-005);
mpf_set_d(x_korrekt.V[3],0.0002661991887693762);
mpf_set_d(x_korrekt.V[4],0.0);
mpf_set_d(x_korrekt.V[5],1.11746574570356e-005);
mpf_set_d(x_korrekt.V[6],0.0006611988340406431);
mpf_set_d(x_korrekt.V[7],0.0004940540806878521);
mpf_set_d(x_korrekt.V[8],0.0002199832793215027);
mpf_set_d(x_korrekt.V[9],0.0006233535553629146);
mpf_set_d(x_korrekt.V[10],0.000368738130907603);
mpf_set_d(x_korrekt.V[11],0.001084491724119105);
mpf_set_d(x_korrekt.V[12],0.0);
mpf_set_d(x_korrekt.V[13],0.0002018521429914197);
mpf_set_d(x_korrekt.V[14],0.0);
mpf_set_d(x_korrekt.V[15],0.0);
mpf_set_d(x_korrekt.V[16],0.0002746316505334807);
mpf_set_d(x_korrekt.V[17],0.0004988694607769221);
mpf_set_d(x_korrekt.V[18],0.0006114602227588423);
mpf_set_d(x_korrekt.V[19],0.0);
mpf_set_d(x_korrekt.V[20],0.0);
mpf_set_d(x_korrekt.V[21],0.0006207347058335339);
mpf_set_d(x_korrekt.V[22],0.0001310930520448684);
mpf_set_d(x_korrekt.V[23],0.0001219745433488594);
mpf_set_d(x_korrekt.V[24],0.0008178010580922872);
mpf_set_d(x_korrekt.V[25],0.0001634813610327911);
mpf_set_d(x_korrekt.V[26],4.925094408131503e-005);
mpf_set_d(x_korrekt.V[27],0.001033088568627202);
mpf_set_d(x_korrekt.V[28],0.001804400380611476);
mpf_set_d(x_korrekt.V[29],0.0002397833987186662);
mpf_set_d(x_korrekt.V[30],0.001059097430855059);
mpf_set_d(x_korrekt.V[31],0.0008735355459969439);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.0004599750396072778);
mpf_set_d(x_korrekt.V[34],0.001558329620175675);
mpf_set_d(x_korrekt.V[35],0.000212701065025905);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.0006613410810805574);

double b_korrekt=-0.447146;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.007903);
#endif

#ifdef leu_quadratisch
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0);
mpf_set_d(x_korrekt.V[1],0.03518230670361319);
mpf_set_d(x_korrekt.V[2],0.001389228563445081);
```

```
mpf_set_d(x_korrekt.V[3],0.01331188759841301);
mpf_set_d(x_korrekt.V[4],0.0);
mpf_set_d(x_korrekt.V[5],0.0005611960077244911);
mpf_set_d(x_korrekt.V[6],0.03055735881225561);
mpf_set_d(x_korrekt.V[7],0.0246353982334123);
mpf_set_d(x_korrekt.V[8],0.01100555465278763);
mpf_set_d(x_korrekt.V[9],0.03113865144088284);
mpf_set_d(x_korrekt.V[10],0.01839972654842207);
mpf_set_d(x_korrekt.V[11],0.05410257024708683);
mpf_set_d(x_korrekt.V[12],0.0);
mpf_set_d(x_korrekt.V[13],0.01005445587802409);
mpf_set_d(x_korrekt.V[14],0.0);
mpf_set_d(x_korrekt.V[15],0.0);
mpf_set_d(x_korrekt.V[16],0.01369005973462452);
mpf_set_d(x_korrekt.V[17],0.02489942843005731);
mpf_set_d(x_korrekt.V[18],0.03054298628732526);
mpf_set_d(x_korrekt.V[19],0.0);
mpf_set_d(x_korrekt.V[20],0.0);
mpf_set_d(x_korrekt.V[21],0.03097404579157284);
mpf_set_d(x_korrekt.V[22],0.006550508128536769);
mpf_set_d(x_korrekt.V[23],0.006117018281969215);
mpf_set_d(x_korrekt.V[24],0.04084738479673983);
mpf_set_d(x_korrekt.V[25],0.008173218014019578);
mpf_set_d(x_korrekt.V[26],0.002475143338421961);
mpf_set_d(x_korrekt.V[27],0.05160742013246438);//ab hier negative Klasse
mpf_set_d(x_korrekt.V[28],0.09008823268637441);
mpf_set_d(x_korrekt.V[29],0.01198810187077248);
mpf_set_d(x_korrekt.V[30],0.05292635056350507);
mpf_set_d(x_korrekt.V[31],0.04361851397595798);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.02300648072171758);
mpf_set_d(x_korrekt.V[34],0.0777358287744629);
mpf_set_d(x_korrekt.V[35],0.0106170968387591);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.03302010192532048);

double b_korrekt=-0.447014;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.394643);
#endif

#ifdef colon
Matrix_mpf_t Daten(30,2001);
Vektor_mpf_t x_korrekt(30);

mpf_set_d(x_korrekt.V[0],0.001687083657503246);
mpf_set_d(x_korrekt.V[1],0.002631082717828218);
mpf_set_d(x_korrekt.V[2],0.003449820954301625);
mpf_set_d(x_korrekt.V[3],0.003938151970240279);
mpf_set_d(x_korrekt.V[4],0.00158961365550937);
mpf_set_d(x_korrekt.V[5],0.001848751653980758);
mpf_set_d(x_korrekt.V[6],0.00087409795064976);
mpf_set_d(x_korrekt.V[7],0.001666007178610175);
mpf_set_d(x_korrekt.V[8],0.0005080496255244833);
mpf_set_d(x_korrekt.V[9],0.0005502448654384846);
mpf_set_d(x_korrekt.V[10],0.0);
mpf_set_d(x_korrekt.V[11],0.0007292664654302471);
```

```
mpf_set_d(x_korrekt.V[12],0.001933347899523181);
mpf_set_d(x_korrekt.V[13],0.001392388234183991);
mpf_set_d(x_korrekt.V[14],0.003122836019868257);
mpf_set_d(x_korrekt.V[15],0.002494088274429402);
mpf_set_d(x_korrekt.V[16],0.0);
mpf_set_d(x_korrekt.V[17],6.181247618752138e-005);
mpf_set_d(x_korrekt.V[18],0.0);
mpf_set_d(x_korrekt.V[19],0.0009536801935352902);
mpf_set_d(x_korrekt.V[20],0.001061966751975262);
mpf_set_d(x_korrekt.V[21],0.002597094170556967);
mpf_set_d(x_korrekt.V[22],0.0007305667795913224);
mpf_set_d(x_korrekt.V[23],0.0008712603550831134);
mpf_set_d(x_korrekt.V[24],0.0009871026479213463);
mpf_set_d(x_korrekt.V[25],0.0);
mpf_set_d(x_korrekt.V[26],0.001777719540111608);
mpf_set_d(x_korrekt.V[27],0.0);
mpf_set_d(x_korrekt.V[28],0.0);
mpf_set_d(x_korrekt.V[29],0.0);

double b_korrekt=-0.347524;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.018729);
#endif

#ifdef colon_quadratisch
Matrix_mpf_t Daten(30,2001);
Vektor_mpf_t x_korrekt(30);

mpf_set_d(x_korrekt.V[0],0.02602578140371289);
mpf_set_d(x_korrekt.V[1],0.03423608811161808);
mpf_set_d(x_korrekt.V[2],0.01849746694578573);
mpf_set_d(x_korrekt.V[3],0.02738524324068293);
mpf_set_d(x_korrekt.V[4],0.0290238482814);
mpf_set_d(x_korrekt.V[5],0.0242287066152327);
mpf_set_d(x_korrekt.V[6],0.02002858952012789);
mpf_set_d(x_korrekt.V[7],0.0238998460510916);
mpf_set_d(x_korrekt.V[8],0.00798345019560899);
mpf_set_d(x_korrekt.V[9],0.02310218342516291);
mpf_set_d(x_korrekt.V[10],0.009961338411178215);
mpf_set_d(x_korrekt.V[11],0.03075926195260311);
mpf_set_d(x_korrekt.V[12],0.03706012108962523);
mpf_set_d(x_korrekt.V[13],0.04632127586926887);
mpf_set_d(x_korrekt.V[14],0.03693711292769399);
mpf_set_d(x_korrekt.V[15],0.02609467495460025);
mpf_set_d(x_korrekt.V[16],0.01954083400944914);
mpf_set_d(x_korrekt.V[17],0.02285221576938233);
mpf_set_d(x_korrekt.V[18],0.009177950471338174);
mpf_set_d(x_korrekt.V[19],0.001969389826406242);
mpf_set_d(x_korrekt.V[20],0.01883456456279536);
mpf_set_d(x_korrekt.V[21],0.02861088503343977);
mpf_set_d(x_korrekt.V[22],0.009231423508808642);
mpf_set_d(x_korrekt.V[23],0.01884535752364217);
mpf_set_d(x_korrekt.V[24],0.0168273674157438);
mpf_set_d(x_korrekt.V[25],0.0);
mpf_set_d(x_korrekt.V[26],0.02682592001824902);
mpf_set_d(x_korrekt.V[27],0.0007687646563423519);
mpf_set_d(x_korrekt.V[28],0.005159981539461174);
```

```
mpf_set_d(x_korrekt.V[29],0.0);

double b_korrekt=0.302713;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.300096);
#endif

#ifdef duke
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0006206705338290365);
mpf_set_d(x_korrekt.V[1],0.0002246535024378646);
mpf_set_d(x_korrekt.V[2],0.001420154104024451);
mpf_set_d(x_korrekt.V[3],0.002957040640321748);
mpf_set_d(x_korrekt.V[4],0.00127081356060853);
mpf_set_d(x_korrekt.V[5],0.001031511164316221);
mpf_set_d(x_korrekt.V[6],0.001551752860746905);
mpf_set_d(x_korrekt.V[7],0.001133432136188666);
mpf_set_d(x_korrekt.V[8],0.000312588666952854);
mpf_set_d(x_korrekt.V[9],0.0001540423336580267);
mpf_set_d(x_korrekt.V[10],0.001805291005741564);
mpf_set_d(x_korrekt.V[11],0.002320442504392584);
mpf_set_d(x_korrekt.V[12],0.0002039140817325677);
mpf_set_d(x_korrekt.V[13],0.001783944275377242);
mpf_set_d(x_korrekt.V[14],0.001863960976299457);
mpf_set_d(x_korrekt.V[15],0.0009324754147303066);
mpf_set_d(x_korrekt.V[16],0.001786676647944464);
mpf_set_d(x_korrekt.V[17],0.0);
mpf_set_d(x_korrekt.V[18],0.002066132104009223);
mpf_set_d(x_korrekt.V[19],0.001169491571013092);
mpf_set_d(x_korrekt.V[20],0.002008455595720181);
mpf_set_d(x_korrekt.V[21],0.002067723329792848);
mpf_set_d(x_korrekt.V[22],0.000584437223329038);
mpf_set_d(x_korrekt.V[23],0.0001592061814764154);
mpf_set_d(x_korrekt.V[24],0.0);
mpf_set_d(x_korrekt.V[25],0.003795219951353398);
mpf_set_d(x_korrekt.V[26],0.0003481311850569071);
mpf_set_d(x_korrekt.V[27],0.0);
mpf_set_d(x_korrekt.V[28],0.0002398798010494223);
mpf_set_d(x_korrekt.V[29],0.0);
mpf_set_d(x_korrekt.V[30],0.0);
mpf_set_d(x_korrekt.V[31],0.001645282355538716);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.0005274650387891897);
mpf_set_d(x_korrekt.V[34],0.0007213775519165544);
mpf_set_d(x_korrekt.V[35],0.0004072288091995537);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.0009910281590234267);

double b_korrekt=-0.201494;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.019052);
#endif

#ifdef duke_quadratisch
Matrix_mpf_t Daten(38,7130);
```

```
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],3.243478855960109e-007);
mpf_set_d(x_korrekt.V[1],5.90635869328691e-008);
mpf_set_d(x_korrekt.V[2],5.738501638530179e-007);
mpf_set_d(x_korrekt.V[3],1.05506397625681e-006);
mpf_set_d(x_korrekt.V[4],3.813466398786363e-007);
mpf_set_d(x_korrekt.V[5],4.645873940718977e-007);
mpf_set_d(x_korrekt.V[6],6.776502546319503e-007);
mpf_set_d(x_korrekt.V[7],4.888469857427162e-007);
mpf_set_d(x_korrekt.V[8],1.240092349189592e-007);
mpf_set_d(x_korrekt.V[9],3.380128467148818e-008);
mpf_set_d(x_korrekt.V[10],5.02680046624246e-007);
mpf_set_d(x_korrekt.V[11],7.516224057395228e-007);
mpf_set_d(x_korrekt.V[12],1.675884245492257e-007);
mpf_set_d(x_korrekt.V[13],6.213655768224026e-007);
mpf_set_d(x_korrekt.V[14],7.750674083028019e-007);
mpf_set_d(x_korrekt.V[15],1.700501901870856e-007);
mpf_set_d(x_korrekt.V[16],5.757680086212623e-007);
mpf_set_d(x_korrekt.V[17],7.027092293756611e-008);
mpf_set_d(x_korrekt.V[18],6.783540356553512e-007);
mpf_set_d(x_korrekt.V[19],4.564978156399474e-007);
mpf_set_d(x_korrekt.V[20],6.330127558576494e-007);
mpf_set_d(x_korrekt.V[21],6.868112494222045e-007);
mpf_set_d(x_korrekt.V[22],3.413542147883727e-007);
mpf_set_d(x_korrekt.V[23],1.770721128012483e-007);
mpf_set_d(x_korrekt.V[24],2.22787309190812e-007);
mpf_set_d(x_korrekt.V[25],1.175104062500625e-006);
mpf_set_d(x_korrekt.V[26],8.526231255068886e-008);
mpf_set_d(x_korrekt.V[27],8.526331893755719e-008);
mpf_set_d(x_korrekt.V[28],4.068538943923275e-008);
mpf_set_d(x_korrekt.V[29],3.025944775041957e-008);
mpf_set_d(x_korrekt.V[30],0.0);
mpf_set_d(x_korrekt.V[31],5.128982654019395e-007);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],1.0666046072128e-007);
mpf_set_d(x_korrekt.V[34],2.611872522149039e-007);
mpf_set_d(x_korrekt.V[35],1.607784444513643e-007);
mpf_set_d(x_korrekt.V[36],7.543509619131898e-008);
mpf_set_d(x_korrekt.V[37],2.460179116686543e-007);

double b_korrekt=-0.290858;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.000007);
#endif

int k, Anzahl_Gleichungsnebenbedingungen=1;

ifstream data;

#ifdef leu
data.open("leu.txt.scale");
#endif

#ifdef leu_quadratisch
data.open("leu.txt.scale");
#endif
```

```
#ifdef colon
data.open("cc-train.tr.scale");
#endif

#ifdef colon_quadratisch
data.open("cc-train.tr.scale");
#endif

#ifdef duke
data.open("duke.tr.scale");
#endif

#ifdef duke_quadratisch
data.open("duke.tr.scale");
#endif

if(!data)
    {
    cout<<"Datei konnte nicht geoeffnet werden. Abbruch"<<endl;
    exit(1);
    }
string Daten_String;
string Hilfsstring;
string Hilfsstring2;


char C;
while(data.get(C)){
    Daten_String+=C;//Input in Daten_String schhreiben
    }

k=0;
int counter=0;
size_t found,found_d;
double hilfe=0.0;
int Zahl;
while(!Daten_String.empty()){
    found=Daten_String.find(' ');
    Hilfsstring=Daten_String.substr(0,(int)found);
    Daten_String.erase(0,found+(size_t)1);

    found_d=Hilfsstring.find(":");
    if(found_d==string::npos){
        hilfe=atof(Hilfsstring.c_str());
        Zahl=0;
        }
    else{
        Hilfsstring2=Hilfsstring.substr(0,(int)found_d);

        Hilfsstring.erase(0,(int)found_d+1);
        Zahl=atoi(Hilfsstring2.c_str());

        hilfe=atof(Hilfsstring.c_str());
        }

    mpf_set_d(Daten.M[counter][Zahl],hilfe);
```

```
    #ifdef leu
    if(Zahl==7129){
    #endif

    #ifdef leu_quadratisch
    if(Zahl==7129){
    #endif

    #ifdef colon
    if(Zahl==2000){
    #endif

    #ifdef colon_quadratisch
    if(Zahl==2000){
    #endif

    #ifdef duke
    if(Zahl==7129){
    #endif

    #ifdef duke_quadratisch
    if(Zahl==7129){
    #endif

        found=Daten_String.find('\n');
        Daten_String.erase(0,found+(size_t)1);
        counter++;
        k++;
        }
    }

data.close();
k=KK;

Vektor_mpf_t Daten_Y(Daten.Hoehe);
Daten_Y.getSpaltenvektor(0,Daten);//Klassenindikatoren auslesen
SVM Daten_verteilt(Daten.Hoehe,Daten.Breite-1,n,t,q,q_1,k,f);

Matrix_mpf_t Daten_rechts(Daten.Hoehe,Daten.Breite-1);
Daten_rechts.Kopiere_rechts(Daten);
Daten_verteilt.Daten=Daten_rechts;
Daten_verteilt.Y.Unskaliert_einlesen(Daten_Y);
Daten_verteilt.Y.Ausgabe("Y_s");
Daten_verteilt.Lambda.Erzeugung_double(0.5,t,q);//Parameter lambda, der
    die Toleranz der SVM angibt

//Berechnung der Kernel-Matrix
#ifdef leu
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef leu_quadratisch
Daten_verteilt.Quadratischer_Kernel(0.0001,100.0);
#endif

#ifdef colon
```

```
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef colon_quadratisch
Daten_verteilt.Quadratischer_Kernel(0.01,1.0);
#endif

#ifdef duke
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef duke_quadratisch
Daten_verteilt.Quadratischer_Kernel(1.0,1.0);
#endif

Vektor_LA c(Daten_verteilt.Hoehe_Daten,n,t,q,q_1,k,f);
for(i=0; i<c.d; i++){
    c[i].Erzeugung_double(-1.0,t,q);
    }

Daten_verteilt.Berechne_G();
Matrix_LA Q=Daten_verteilt.G;

Vektor_LA b(2*Daten_verteilt.K.Zeilenzahl+1,n,t,q,q_1,k,f);
for(i=1; i<Daten_verteilt.K.Zeilenzahl+1; i++){
    b[i]=-Daten_verteilt.Lambda;
    }

Matrix_LA A(b.d,Daten_verteilt.Y.d,n,t,q,q_1,k,f);
for(i=1; i<Daten_verteilt.K.Zeilenzahl+1; i++){
    A.M[i][i-1].Erzeugung_double(-1.0,t,q);
    }

for(i=Daten_verteilt.K.Zeilenzahl+1; i<A.Zeilenzahl; i++){
    A.M[i][i-Daten_verteilt.K.Zeilenzahl-1].Erzeugung_double(1.0,t,q);
    }

for(i=0; i<A.Spaltenzahl; i++){
    A.M[0][i]=Daten_verteilt.Y[i]*zwei_hoch_f;//Gleichheitsbedingung
    }

Vektor_LA x(A.Spaltenzahl,n,t,q,q_1,k,f);
x[0]=Daten_verteilt.Lambda;
x[0].Div_Konst_int(2,q,q_1,k);
x[x.d-1]=x[0];//Eine positive und eine negative Klasse (die erste und die
    letzte) werden mit einem positiven wert belegt, der nicht dau führt,
    dass die Ungleicungen aktiv werden

Vektor_LA S(b.d,n,t,q,q_1,k,f);

Matrix_LA G(Q.Zeilenzahl+A.Spaltenzahl,Q.Spaltenzahl+A.Spaltenzahl,n,t,q,
    q_1,k,f);
Vektor_LA l(Q.Zeilenzahl+A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA lambda(b.d,n,t,q,q_1,k,f);
Vektor_LA lambda_min=lambda;
Vektor_LA lambda_kurz(A.Spaltenzahl-Anzahl_Gleichungsnebenbedingungen,n,t,
    q,q_1,k,f);
```

```
Share Bit(n,t,q);
Share Hilfe(n,t,q);
Share Hilfe2(n,t,q);
Share lambda_q (n,t,q);//Speichere das aktuelle lambda
Vektor_LA Hilfe_Vektor(A.Spaltenzahl,n,t,q,q_1,k,f);
Share alpha(n,t,q);
Vektor_LA d(x.d,n,t,q,q_1,k,f);
Vektor_LA Index(A.Zeilenzahl,n,t,q,q_1,k,f);//wird zur Bestimmung des
    Minimums von alpha benötigt
Vektor_LA Index_kurz(A.Spaltenzahl-Anzahl_Gleichungsnebenbedingungen,n,t,q
    ,q_1,k,f);
Vektor_LA Index_kurz_mit_GHNB(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Index_AM(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA ax(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_Zq(b.d,n,t,q,q_1,k,f);
Vektor_LA Kopie_AM(b.d,n,t,q,q_1,k,f);
Vektor_LA AM_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Matrix_LA Kopie_A=A;
Matrix_LA Merke_A=Kopie_A;//Speichere die Matrix A mit den temporären VZ
    der GHNB
Vektor_LA Kopie_b=b;//Dito für die entsprechenden Zeilen von b
Matrix_LA A_Kompakt(A.Spaltenzahl,A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Kopie_c=c;
Vektor_LA Kopie_x=x;
Vektor_LA Ergebnis=x;
Share Hilfe_2=Hilfe;
Matrix_LA P(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_t(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);//Die Transponierte
     von P
Share alpha_kleiner_eins(n,t,q);

mpz_t eins;
mpz_init_set_ui(eins,1);

mpz_t epsilon;
mpz_init_set_ui(epsilon,1);
mpz_mul_2exp(epsilon,epsilon,f-30);

time_t start_Phase2,stop_Phase_2;
///Beginn Phase 2
Share Eins(n,t,q);
Eins.Erzeugung(eins,t,q);


AM_Zq[0]=Eins;
for(i=Daten_verteilt.K.Zeilenzahl+2; i<A.Zeilenzahl-1; i++){
    AM_Zq[i]=Eins; //Die Variablen 2,...,m-1 sind 0, also die
        entsprechenden Ungleichungen aktiv
    }

start_Phase2=clock();

for(i=0; i<A.Zeilenzahl; i++){//Bestimme die initialen Werte ax(i)
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(x,ax[i]);
    }
```

```
for(Iterationszaehler=1; Iterationszaehler<501; Iterationszaehler++){

Kopie_c=c;
Kopie_x=x;

Kopie_x.MatrixMult(Q,Kopie_x);
Kopie_x.TruncPr();
Kopie_c+=Kopie_x;//Kopie_c=Q*x+c

Kopie_A.Multipliziere_Zeilenweise(&AM_Zq.Vector_LA);
Kopie_A.Matrix_Zusammenschieben_binaer(AM_Zq.Vector_LA,P);

A_Kompakt.Oben_einpassen(Kopie_A);

P_t=P;
P_t.Transponiere_quadratisch();

AM_Zq.MatrixMult(P,Kopie_AM);

AM_kurz.Oben_einpassen(Kopie_AM);

G=0;
l=0;
Erstelle_Gleichungssystem(G,Q,A_Kompakt,x,A_Kompakt.Zeilenzahl,A_Kompakt.
    Spaltenzahl,n,t,q,q_1,k,f);
Erstelle_Vektor(l,Kopie_c,c.d,l.d);

G.Loese_GLS_QR(l.Vector_LA,l.d);
//G.Loese_GLS_LR(l.Vector_LA,l.d);

j=0;
for(i=c.d; i<l.d; i++){
    lambda.Vector_LA.at(j)=l.Vector_LA.at(i);
    j++;
    }

lambda.MatrixMult(P_t,lambda);//Setze lambda an die richtigen Stellen
    zurück; die ungültigen Stellen werden automatisch mit Null besetzt

for(i=0; i<lambda_kurz.d; i++){
    lambda_kurz[i]=l[i+c.d+Anzahl_Gleichungsnebenbedingungen];//
        Kompaktifizierte Form des Vektors lambda erstellen
    lambda_kurz[i]=lambda_kurz[i]*AM_kurz[i+
        Anzahl_Gleichungsnebenbedingungen]+(-AM_kurz[i+
        Anzahl_Gleichungsnebenbedingungen]+1)*unendlich;//Ersetze nicht-
        aktive Komponenten mit unendlich; diese sind wg.
        Komplementärbedingung = 0
    }

for(i=0; i<x.d; i++){
    d.Vector_LA.at(i)=l.Vector_LA.at(i);
    }

d.Gleich_Null_approximativ(Bit);

///Schritt 3
Index_kurz=0;
```

```
lambda_kurz.Min(Index_kurz,k);
Index_kurz_mit_GHNB.Kopiere_unten(Index_kurz);

lambda_q=0;
lambda_q=lambda_kurz[Index_kurz];

///Verkürzten Vektor lambda wieder auf die richtige Länge expandieren
lambda_min.Oben_einpassen(Index_kurz_mit_GHNB);
lambda_min.MatrixMult(P_t,lambda_min);

lambda_q.GTZ_approximativ(t,q,q_1,k);
lambda_q.MulPub(Bit,Ende);

if(mpz_cmp_si(Ende,1)==0){
    stop_Phase_2=clock();
    FILE* datei1;
    datei1=fopen("Ausgabe.txt","a");

    fprintf(datei1,"\n\n\n
        ----------------------------------------------------------");
    time_t Zeit;
    struct tm *ts;//Variable zur Zeitmessung
    time(&Zeit);
    ts=localtime(&Zeit);
    fprintf(datei1,"\n\n\nZeit: %s\n\n",asctime(ts));

    cout<<"\n\nOptimaler Punkt x erreicht"<<endl;
    Q.Ausgabe_skaliert("Q");
    c.Ausgabe_skaliert("c");
    A.Ausgabe_skaliert("A");
    b.Ausgabe_skaliert("b");

    Q.Ausgabe_skaliert_Datei("Q",datei1);
    c.Ausgabe_skaliert_Datei("c",datei1);
    A.Ausgabe_skaliert_Datei("A",datei1);
    b.Ausgabe_skaliert_Datei("b",datei1);
    fprintf(datei1,"\nOptimaler Punkt x erreicht");

    fprintf(datei1,"\nLaufzeit des Programms: %f s",((double)stop_Phase_2
        -(double)start_Phase2)/CLOCKS_PER_SEC);
    printf("\nZeit fuer Phase 2: %f s",((double)stop_Phase_2-(double)
        start_Phase2)/CLOCKS_PER_SEC);
    printf("\n\nAnzahl der Iterationen: %d", Iterationszaehler);
    printf("\nAnzahl volle Schritte: %d",Zaehler_voller_Schritt);
    printf("\nAnzahl halbe Schritte: %d",Zaehler_halber_Schritt);
    fprintf(datei1,"\n\nAnzahl der Iterationen: %d", Iterationszaehler);
    fprintf(datei1,"\nAnzahl volle Schritte: %d",Zaehler_voller_Schritt);
    fprintf(datei1,"\nAnzahl halbe Schritte: %d\n\n",
        Zaehler_halber_Schritt);
    mpf_t Z;
    mpf_init(Z);
    Zielfunktion(Q,x,c,Z,n,t,k,q,q_1);
    gmp_printf("\nZielfunktion: %.10Ff",Z);
    gmp_fprintf(datei1,"\nZielfunktion: %.10Ff",Z);
    mpf_sub(Z,Z,f_korrekt);
    gmp_printf("\nDifferenz zu exaktem Wert der Zielfunktion: %.10Fe",Z);
```

```
    gmp_fprintf(datei1,"\nDifferenz␣zu␣exaktem␣Wert␣der␣Zielfunktion:␣%.10
        Fe",Z);
    mpf_div(Z,Z,f_korrekt);
    gmp_printf("\nRelativer␣Fehler␣der␣Zielfunktion:␣%.10Fe",Z);
    gmp_fprintf(datei1,"\nRelativer␣Fehler␣der␣Zielfunktion:␣%.10Fe",Z);
    Vektor_mpf_t x_Werte(x.d);
    x.Ausgabe_skaliert_Vektor_mpf_t(x_Werte);
    x_Werte-=x_korrekt;
    x_Werte.Ausgabe_wissenschaftlich("x_Diff");
    x_Werte.Ausgabe_wissenschaftlich_Datei("x_Diff",datei1);
    double Norm_Differenz=x_Werte.Norm();
    printf("\n\nNorm␣der␣Differenz:␣%e",Norm_Differenz);
    fprintf(datei1,"\n\nNorm␣der␣Differenz:␣%e", Norm_Differenz);
    Norm_Differenz=Norm_Differenz/x_korrekt.Norm();
    printf("\nRelativer␣Fehler␣der␣Norm:␣%e",Norm_Differenz);
    fprintf(datei1,"\nRelativer␣Fehler␣der␣Norm:␣%e", Norm_Differenz);
    for(i=0; i<x.d; i++){
        x[i].Rekonstrukt(s,t,q);
        mpf_set_z(Z,s);
        mpf_div_2exp(Z,Z,f);
        gmp_fprintf(datei1,"\nx(%d)=%.10Ff",i,Z);
        }
    x.Ausgabe_skaliert("x");

    AM_Zq.Ausgabe("AM");

    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_b();

    mpf_set_z(Z,s);
    mpf_div_2exp(Z,Z,f);
    mpf_t bf;
    mpf_init_set_d(bf,b_korrekt);
    mpf_add(Z,Z,bf);
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Daten_verteilt.b.Rekonstrukt_Ausgabe("b");
    Daten_verteilt.b.Ausgabe_skaliert_Datei("b",datei1);
    Daten_verteilt.b.Rekonstrukt_GRR_mit_VZ(s,q,t);
    printf("\n\nkorrektes␣b:␣%f",b_korrekt);
    fprintf(datei1,"\n\nkorrektes␣b:␣%f",b_korrekt);
    gmp_printf("\nAbsoluter␣Fehler␣in␣b:␣%Ff",Z);
    gmp_fprintf(datei1,"\nAbsoluter␣Fehler␣in␣b:␣%Ff",Z);
    mpf_div(Z,Z,bf);
    gmp_printf("\nRelativer␣Fehler␣in␣b:␣%Ff",Z);
    gmp_fprintf(datei1,"\nRelativer␣Fehler␣in␣b:␣%Ff",Z);

    fclose(datei1);
    break;
    }
///Schritt 2
//d auf Null setzen, falls der (appr.) Test dies ergibt; korrigiere so für
     Vektoren, die nur auf Grund von Rundungsfehlern!=0 sind
Hilfe=Bit;
Hilfe*=-1;
```

```
Hilfe+=1;
for(i=0; i<x.d; i++){
    d[i]=d[i]*Hilfe;
    }

Berechne_alpha(A,d,x,b,alpha,Index,AM_Zq,alpha_kleiner_eins,A.Zeilenzahl,n
    ,t,k,q,q_1);

Hilfe=Bit;
Hilfe*=-1;
Hilfe.Add_Konst_int(1,q);//Hilfe=1-(d=?0)
d*=alpha;
d*=Hilfe;//korrigiere d
d.TruncPr();
x+=d;//x_{k+1}=x_k+alpha_k*d_k

Hilfe=Bit;

Index*=alpha_kleiner_eins;//Index_AM=Index_AM*(alpha_k<?1)
Hilfe*=-1;
Hilfe.Add_Konst_int(1,q);//Hilfe=1-(d=?0)
Index_AM*=Hilfe;//Index=Index_AM*(alpha_k<?=1)*(1-(d=?0))
AM_Zq+=Index;//W_{k+1}=W_k+I*(alpha<?0)*(1-(d=?0))

lambda_min*=-1;
lambda_min*=Bit;
AM_Zq+=lambda_min;//W_{k+1}=W_k+I*(alpha<?0)*(1-(d=?0))-d*I_lambda_min

for(i=0; i<A.Zeilenzahl; i++){//Bestimme die neuen Werte ax(i)
    Hilfe_Vektor.getZeilenvektor(A,i);
    Hilfe_Vektor.SkalarMult(x,ax[i]);
    }

Kopie_A=A;
Kopie_x=x;
Kopie_c=c;
}

Daten_verteilt.Alpha=x;//Lösungsvektor in den Scope der SVM schreieben
Daten_verteilt.Berechne_SV();
Daten_verteilt.Berechne_b();
Daten_verteilt.SV.Ausgabe("SV");
Daten_verteilt.Entscheidungsregel_Test("Ausgabe.txt");

ifstream data_test;

#ifdef leu
Daten_verteilt.Save_SVM("leu");
Matrix_mpf_t Daten_Test(34,7130);
data_test.open("leu.t.scale");
#endif

#ifdef leu_quadratisch
Daten_verteilt.Save_SVM("leu");
Matrix_mpf_t Daten_Test(34,7130);
data_test.open("leu.t.scale");
#endif
```

```
#ifdef colon
Daten_verteilt.Save_SVM("colon");
Matrix_mpf_t Daten_Test(32,2001);
data_test.open("cc-test.scale");
#endif

#ifdef colon_quadratisch
Daten_verteilt.Save_SVM("colon");
Matrix_mpf_t Daten_Test(32,2001);
data_test.open("cc-test.scale");
#endif

#ifdef duke
Daten_verteilt.Save_SVM("duke");
Matrix_mpf_t Daten_Test(4,7130);
data_test.open("duke.val.scale");
#endif

#ifdef duke_quadratisch
Daten_verteilt.Save_SVM("duke");
Matrix_mpf_t Daten_Test(4,7130);
data_test.open("duke.val.scale");
#endif

if(!data_test)
    {
    cout<<"Datei konnte nicht geoeffnet werden. Abbruch"<<endl;
    exit(1);
    }
string Daten_String_Test;
string Hilfsstring_Test;
string Hilfsstring2_Test;


while(data_test.get(C)){
    Daten_String_Test+=C;//Input in Daten_String schreiben
    }

k=0;
 counter=0;
hilfe=0.0;

while(!Daten_String_Test.empty()){
    found=Daten_String_Test.find(' ');
    Hilfsstring_Test=Daten_String_Test.substr(0,(int)found);
    Daten_String_Test.erase(0,found+(size_t)1);

    found_d=Hilfsstring_Test.find(":");
    if(found_d==string::npos){
        hilfe=atof(Hilfsstring_Test.c_str());
        Zahl=0;
        }
    else{
        Hilfsstring2_Test=Hilfsstring_Test.substr(0,(int)found_d);

        Hilfsstring_Test.erase(0,(int)found_d+1);
```

```
        Zahl=atoi(Hilfsstring2_Test.c_str());

        hilfe=atof(Hilfsstring_Test.c_str());
        }

    mpf_set_d(Daten_Test.M[counter][Zahl],hilfe);

    #ifdef leu
    if(Zahl==7129){
    #endif

    #ifdef leu_quadratisch
    if(Zahl==7129){
    #endif

    #ifdef colon
    if(Zahl==2000){
    #endif

    #ifdef colon_quadratisch
    if(Zahl==2000){
    #endif

    #ifdef duke
    if(Zahl==7129){
    #endif

    #ifdef duke_quadratisch
    if(Zahl==7129){
    #endif

        found=Daten_String_Test.find('\n');
        Daten_String_Test.erase(0,found+(size_t)1);
        counter++;
        k++;
        }
    }

data_test.close();

Vektor_mpf_t Daten_Y_Test(Daten_Test.Hoehe);
Daten_Y_Test.getSpaltenvektor(0,Daten_Test);//Klassenindikatoren auslesen
SVM Daten_verteilt_Test(Daten_Test.Hoehe,Daten_Test.Breite-1,n,t,q,q_1,k,f
    );
Matrix_mpf_t Daten_rechts_Test(Daten_Test.Hoehe,Daten_Test.Breite-1);
Daten_rechts_Test.Kopiere_rechts(Daten_Test);
Daten_verteilt_Test.Daten=Daten_rechts_Test;

Daten_verteilt_Test.Y.Unskaliert_einlesen(Daten_Y_Test);
Daten_verteilt_Test.Lambda=Daten_verteilt.Lambda;//Parameter lambda, der
    die Toleranz der SVM angibt

start=clock();

#ifdef leu
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");
```

```
#elif colon
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif duke
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");
#endif

#ifdef leu_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif colon_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif duke_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");
#endif
stop=clock();

FILE * datei1;
printf("\n\nZeit fuer Verifikation: %f s",((double)stop-(double)start)/
    CLOCKS_PER_SEC);
datei1=fopen("Ausgabe.txt","a");
fprintf(datei1,"\nZeit fuer Verifikation: %f s",((double)stop-(double)
    start)/CLOCKS_PER_SEC);
getchar();

mpz_clear(s);
mpz_clear(unendlich);
return 0;
```

E.2 Verwendung des dualen Algorithmus

```
int main()
{
int i,j,n=5, t=2, k, KK=128, Iterationszaehler=1;
mpf_set_default_prec(300);

mpz_t q;
mpz_t q_1;
mpz_t basis;
mpz_t modulo;
mpz_t modulo_q1;

mpz_init(q);
mpz_init(q_1);
mpz_init_set_ui(basis,2);
mpz_init(modulo);
mpz_init(modulo_q1);

mpz_pow_ui(q,basis,Modullaenge);
```

```
mpz_pow_ui(q_1,basis,Modullaenge_kurz);

do{
mpz_nextprime(q,q);
mpz_nextprime(q_1,q_1);
mpz_mod_ui(modulo,q, 4);
mpz_mod_ui(modulo_q1,q_1, 4);
}
while(mpz_cmp_ui(modulo,3)!=0 || mpz_cmp_ui(modulo_q1,3)!=0); //Erzeuge
    geeignete Primzahl für Berechnung des modulus (muß kongruent 3 mod 4
    sein)

haupt::Berechne_Koeffizienten(n,t,KK,f,q,q_1);

mpz_init_set(grosser_Modulus,q);
mpz_init_set(kleiner_Modulus,q_1);

mpz_t unendlich;
mpz_init(unendlich);

clock_t start, stop, Zeit;


mpz_t seed_mpz;
mpz_init(seed_mpz);
time_t seed;
double seed_d;

seed=clock();
seed_d=(double) seed;
mpz_set_d(seed_mpz,seed_d);

gmp_randstate_t state;
gmp_randinit_default(state);
gmp_randseed(state,seed_mpz);

mpz_set_ui(unendlich,1);
mpz_mul_2exp(unendlich,unendlich,KK-1);
mpz_sub_ui(unendlich,unendlich,1);

#ifdef leu
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0);
mpf_set_d(x_korrekt.V[1],0.00070373784101635 03);
mpf_set_d(x_korrekt.V[2],2.738219115626705e-005);
mpf_set_d(x_korrekt.V[3],0.0002661991887693762);
mpf_set_d(x_korrekt.V[4],0.0);
mpf_set_d(x_korrekt.V[5],1.11746574570356e-005);
mpf_set_d(x_korrekt.V[6],0.0006119883404064 31);
mpf_set_d(x_korrekt.V[7],0.0004940540806878521);
mpf_set_d(x_korrekt.V[8],0.0002199832793215027);
mpf_set_d(x_korrekt.V[9],0.0006233535553629146);
mpf_set_d(x_korrekt.V[10],0.000368738130907603);
mpf_set_d(x_korrekt.V[11],0.0010844917241191 05);
mpf_set_d(x_korrekt.V[12],0.0);
```

```
mpf_set_d(x_korrekt.V[13],0.0002018521429914197);
mpf_set_d(x_korrekt.V[14],0.0);
mpf_set_d(x_korrekt.V[15],0.0);
mpf_set_d(x_korrekt.V[16],0.0002746316505334807);
mpf_set_d(x_korrekt.V[17],0.0004988694607769221);
mpf_set_d(x_korrekt.V[18],0.0006114602227588423);
mpf_set_d(x_korrekt.V[19],0.0);
mpf_set_d(x_korrekt.V[20],0.0);
mpf_set_d(x_korrekt.V[21],0.0006207347058335339);
mpf_set_d(x_korrekt.V[22],0.0001310930520448684);
mpf_set_d(x_korrekt.V[23],0.0001219745433488594);
mpf_set_d(x_korrekt.V[24],0.0008178010580922872);
mpf_set_d(x_korrekt.V[25],0.0001634813610327911);
mpf_set_d(x_korrekt.V[26],4.925094408131503e-005);
mpf_set_d(x_korrekt.V[27],0.001033088568627202);
mpf_set_d(x_korrekt.V[28],0.001804400380611476);
mpf_set_d(x_korrekt.V[29],0.000239783398718662);
mpf_set_d(x_korrekt.V[30],0.001059097430855059);
mpf_set_d(x_korrekt.V[31],0.0008735355459969439);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.0004599750396072778);
mpf_set_d(x_korrekt.V[34],0.001558329620175675);
mpf_set_d(x_korrekt.V[35],0.000212701065025905);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.0006613410810805574);

double b_korrekt=-0.447146;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.007903);
#endif

#ifdef leu_quadratisch
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0);
mpf_set_d(x_korrekt.V[1],0.03518230670361319);
mpf_set_d(x_korrekt.V[2],0.001389228563445081);
mpf_set_d(x_korrekt.V[3],0.01331188759841301);
mpf_set_d(x_korrekt.V[4],0.0);
mpf_set_d(x_korrekt.V[5],0.0005611960077244911);
mpf_set_d(x_korrekt.V[6],0.03055735881225561);
mpf_set_d(x_korrekt.V[7],0.0246353982334123);
mpf_set_d(x_korrekt.V[8],0.01100555465278763);
mpf_set_d(x_korrekt.V[9],0.03113865144088284);
mpf_set_d(x_korrekt.V[10],0.01839972654842207);
mpf_set_d(x_korrekt.V[11],0.05410257024708683);
mpf_set_d(x_korrekt.V[12],0.0);
mpf_set_d(x_korrekt.V[13],0.01005445587802409);
mpf_set_d(x_korrekt.V[14],0.0);
mpf_set_d(x_korrekt.V[15],0.0);
mpf_set_d(x_korrekt.V[16],0.01369005973462452);
mpf_set_d(x_korrekt.V[17],0.02489942843005731);
mpf_set_d(x_korrekt.V[18],0.03054298628732526);
mpf_set_d(x_korrekt.V[19],0.0);
mpf_set_d(x_korrekt.V[20],0.0);
mpf_set_d(x_korrekt.V[21],0.03097404579157284);
```

```
mpf_set_d(x_korrekt.V[22],0.006550508128536769);
mpf_set_d(x_korrekt.V[23],0.006117018281969215);
mpf_set_d(x_korrekt.V[24],0.04084738479673983);
mpf_set_d(x_korrekt.V[25],0.008173218014019578);
mpf_set_d(x_korrekt.V[26],0.002475143338421961);
mpf_set_d(x_korrekt.V[27],0.05160742013246438);//ab hier negative Klasse
mpf_set_d(x_korrekt.V[28],0.09008823268637441);
mpf_set_d(x_korrekt.V[29],0.01198810187077248);
mpf_set_d(x_korrekt.V[30],0.05292635056350507);
mpf_set_d(x_korrekt.V[31],0.04361851397595798);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.02300648072171758);
mpf_set_d(x_korrekt.V[34],0.0777358287744629);
mpf_set_d(x_korrekt.V[35],0.0106170968387591);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.03302010192532048);

double b_korrekt=-0.447014;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.394643);
#endif

#ifdef colon
Matrix_mpf_t Daten(30,2001);
Vektor_mpf_t x_korrekt(30);

mpf_set_d(x_korrekt.V[0],0.002631082717828218);
mpf_set_d(x_korrekt.V[1],0.001687083657503246);
mpf_set_d(x_korrekt.V[2],0.003449820954301625);
mpf_set_d(x_korrekt.V[3],0.003938151970240279);
mpf_set_d(x_korrekt.V[4],0.00158961365550937);
mpf_set_d(x_korrekt.V[5],0.001848751653980758);
mpf_set_d(x_korrekt.V[6],0.00087409795064976);
mpf_set_d(x_korrekt.V[7],0.001666007178610175);
mpf_set_d(x_korrekt.V[8],0.0005080496255244833);
mpf_set_d(x_korrekt.V[9],0.0005502448654384846);
mpf_set_d(x_korrekt.V[10],0.0);
mpf_set_d(x_korrekt.V[11],0.0007292664654302471);
mpf_set_d(x_korrekt.V[12],0.001933347899523181);
mpf_set_d(x_korrekt.V[13],0.001392388234183991);
mpf_set_d(x_korrekt.V[14],0.003122836019868257);
mpf_set_d(x_korrekt.V[15],0.002494088274429402);
mpf_set_d(x_korrekt.V[16],0.0);
mpf_set_d(x_korrekt.V[17],6.181247618752138e-005);
mpf_set_d(x_korrekt.V[18],0.0);
mpf_set_d(x_korrekt.V[19],0.0009536801935352902);
mpf_set_d(x_korrekt.V[20],0.001061966751975262);
mpf_set_d(x_korrekt.V[21],0.002597094170556967);
mpf_set_d(x_korrekt.V[22],0.0007305667795913224);
mpf_set_d(x_korrekt.V[23],0.0008712603550831134);
mpf_set_d(x_korrekt.V[24],0.0009871026479213463);
mpf_set_d(x_korrekt.V[25],0.0);
mpf_set_d(x_korrekt.V[26],0.001777719540111608);
mpf_set_d(x_korrekt.V[27],0.0);
mpf_set_d(x_korrekt.V[28],0.0);
mpf_set_d(x_korrekt.V[29],0.0);
```

```
double b_korrekt=-0.347524;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.018729);
#endif

#ifdef colon_quadratisch
Matrix_mpf_t Daten(30,2001);
Vektor_mpf_t x_korrekt(30);

mpf_set_d(x_korrekt.V[0],0.02602578140371289);
mpf_set_d(x_korrekt.V[1],0.03423608811161808);
mpf_set_d(x_korrekt.V[2],0.01849746694578573);
mpf_set_d(x_korrekt.V[3],0.02738524324068293);
mpf_set_d(x_korrekt.V[4],0.0290238482814);
mpf_set_d(x_korrekt.V[5],0.0242287066152327);
mpf_set_d(x_korrekt.V[6],0.02002858952012789);
mpf_set_d(x_korrekt.V[7],0.0238998460510916);
mpf_set_d(x_korrekt.V[8],0.00798345019560899);
mpf_set_d(x_korrekt.V[9],0.02310218342516291);
mpf_set_d(x_korrekt.V[10],0.009961338411178215);
mpf_set_d(x_korrekt.V[11],0.03075926195260311);
mpf_set_d(x_korrekt.V[12],0.03706012108962523);
mpf_set_d(x_korrekt.V[13],0.04632127586926887);
mpf_set_d(x_korrekt.V[14],0.03693711292769399);
mpf_set_d(x_korrekt.V[15],0.02609467495460025);
mpf_set_d(x_korrekt.V[16],0.01954083400944914);
mpf_set_d(x_korrekt.V[17],0.02285221576938233);
mpf_set_d(x_korrekt.V[18],0.009177950471338174);
mpf_set_d(x_korrekt.V[19],0.001969389826406242);
mpf_set_d(x_korrekt.V[20],0.01883456456279536);
mpf_set_d(x_korrekt.V[21],0.02861088503343977);
mpf_set_d(x_korrekt.V[22],0.009231423508808642);
mpf_set_d(x_korrekt.V[23],0.01884535752364217);
mpf_set_d(x_korrekt.V[24],0.0168273674157438);
mpf_set_d(x_korrekt.V[25],0.0);
mpf_set_d(x_korrekt.V[26],0.02682592001824902);
mpf_set_d(x_korrekt.V[27],0.0007687646563423519);
mpf_set_d(x_korrekt.V[28],0.005159981539461174);
mpf_set_d(x_korrekt.V[29],0.0);

double b_korrekt=0.302713;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.300096);
#endif

#ifdef duke
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],0.0006206705338290365);
mpf_set_d(x_korrekt.V[1],0.0002246535024378646);
mpf_set_d(x_korrekt.V[2],0.001420154104024451);
mpf_set_d(x_korrekt.V[3],0.002957040640321748);
mpf_set_d(x_korrekt.V[4],0.00127081356060853);
mpf_set_d(x_korrekt.V[5],0.001031511164316221);
mpf_set_d(x_korrekt.V[6],0.001551752860746905);
mpf_set_d(x_korrekt.V[7],0.001133432136188666);
```

```
mpf_set_d(x_korrekt.V[8],0.000312588666952854);
mpf_set_d(x_korrekt.V[9],0.0001540423336580267);
mpf_set_d(x_korrekt.V[10],0.001805291005741564);
mpf_set_d(x_korrekt.V[11],0.002320442504392584);
mpf_set_d(x_korrekt.V[12],0.0002039140817325677);
mpf_set_d(x_korrekt.V[13],0.001783944275377242);
mpf_set_d(x_korrekt.V[14],0.001863960976299457);
mpf_set_d(x_korrekt.V[15],0.0009324754147303066);
mpf_set_d(x_korrekt.V[16],0.001786676647944464);
mpf_set_d(x_korrekt.V[17],0.0);
mpf_set_d(x_korrekt.V[18],0.002066132104009223);
mpf_set_d(x_korrekt.V[19],0.001169491571013092);
mpf_set_d(x_korrekt.V[20],0.002008455595720181);
mpf_set_d(x_korrekt.V[21],0.002067723329792848);
mpf_set_d(x_korrekt.V[22],0.000584437223329038);
mpf_set_d(x_korrekt.V[23],0.0001592061814764154);
mpf_set_d(x_korrekt.V[24],0.0);
mpf_set_d(x_korrekt.V[25],0.003795219951353398);
mpf_set_d(x_korrekt.V[26],0.0003481311850569071);
mpf_set_d(x_korrekt.V[27],0.0);
mpf_set_d(x_korrekt.V[28],0.0002398798010494223);
mpf_set_d(x_korrekt.V[29],0.0);
mpf_set_d(x_korrekt.V[30],0.0);
mpf_set_d(x_korrekt.V[31],0.001645282355538716);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],0.0005274650387891897);
mpf_set_d(x_korrekt.V[34],0.0007213775519165544);
mpf_set_d(x_korrekt.V[35],0.0004072288091995537);
mpf_set_d(x_korrekt.V[36],0.0);
mpf_set_d(x_korrekt.V[37],0.0009910281590234267);

double b_korrekt=-0.201494;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.019052);
#endif

#ifdef duke_quadratisch
Matrix_mpf_t Daten(38,7130);
Vektor_mpf_t x_korrekt(38);

mpf_set_d(x_korrekt.V[0],3.243478855960109e-007);
mpf_set_d(x_korrekt.V[1],5.90635869328691e-008);
mpf_set_d(x_korrekt.V[2],5.738501638530179e-007);
mpf_set_d(x_korrekt.V[3],1.05506397625681e-006);
mpf_set_d(x_korrekt.V[4],3.813466398786363e-007);
mpf_set_d(x_korrekt.V[5],4.645873940718977e-007);
mpf_set_d(x_korrekt.V[6],6.776502546319503e-007);
mpf_set_d(x_korrekt.V[7],4.888469857427162e-007);
mpf_set_d(x_korrekt.V[8],1.240092349189592e-007);
mpf_set_d(x_korrekt.V[9],3.380128467148818e-008);
mpf_set_d(x_korrekt.V[10],5.02680046624246e-007);
mpf_set_d(x_korrekt.V[11],7.516224057395228e-007);
mpf_set_d(x_korrekt.V[12],1.675884245492257e-007);
mpf_set_d(x_korrekt.V[13],6.213655768224026e-007);
mpf_set_d(x_korrekt.V[14],7.750674083028019e-007);
mpf_set_d(x_korrekt.V[15],1.700501901870856e-007);
mpf_set_d(x_korrekt.V[16],5.757680086212623e-007);
```

```
mpf_set_d(x_korrekt.V[17],7.027092293756611e-008);
mpf_set_d(x_korrekt.V[18],6.783540356553512e-007);
mpf_set_d(x_korrekt.V[19],4.564978156399474e-007);
mpf_set_d(x_korrekt.V[20],6.330127558576494e-007);
mpf_set_d(x_korrekt.V[21],6.868112494222045e-007);
mpf_set_d(x_korrekt.V[22],3.413542147883727e-007);
mpf_set_d(x_korrekt.V[23],1.770721128012483e-007);
mpf_set_d(x_korrekt.V[24],2.22787309190812e-007);
mpf_set_d(x_korrekt.V[25],1.175104062500625e-006);
mpf_set_d(x_korrekt.V[26],8.526231255068886e-008);
mpf_set_d(x_korrekt.V[27],8.526331893755719e-008);
mpf_set_d(x_korrekt.V[28],4.068538943923275e-008);
mpf_set_d(x_korrekt.V[29],3.025944775041957e-008);
mpf_set_d(x_korrekt.V[30],0.0);
mpf_set_d(x_korrekt.V[31],5.128982654019395e-007);
mpf_set_d(x_korrekt.V[32],0.0);
mpf_set_d(x_korrekt.V[33],1.0666046072128e-007);
mpf_set_d(x_korrekt.V[34],2.611872522149039e-007);
mpf_set_d(x_korrekt.V[35],1.607784444513643e-007);
mpf_set_d(x_korrekt.V[36],7.543509619131898e-008);
mpf_set_d(x_korrekt.V[37],2.460179116686543e-007);

double b_korrekt=-0.290858;
mpf_t f_korrekt;
mpf_init_set_d(f_korrekt,-0.000007);
#endif

mpz_t eins;
mpz_init_set_ui(eins,1);
int Anzahl_Gleichungsnebenbedingungen=1;

ifstream data;

#ifdef leu
data.open("leu.txt.scale");
#endif

#ifdef leu_quadratisch
data.open("leu.txt.scale");
#endif

#ifdef colon
data.open("cc-train.model");
#endif

#ifdef colon_quadratisch
data.open("cc-train.model");
#endif

#ifdef duke
data.open("duke.tr.scale");
#endif

#ifdef duke_quadratisch
data.open("duke.tr.scale");
#endif
```

```
if(!data)
    {
    cout<<"Datei konnte nicht geoeffnet werden. Abbruch"<<endl;
    exit(1);
    }
string Daten_String;
string Hilfsstring;
string Hilfsstring2;


char C;
while(data.get(C)){
    Daten_String+=C;//Input in Daten_String schreiben
    }

k=0;
int counter=0;
size_t found,found_d;
double hilfe=0.0;
int Zahl;
while(!Daten_String.empty()){
    found=Daten_String.find(' ');
    Hilfsstring=Daten_String.substr(0,(int)found);
    Daten_String.erase(0,found+(size_t)1);

    found_d=Hilfsstring.find(":");
    if(found_d==string::npos){
        hilfe=atof(Hilfsstring.c_str());
        Zahl=0;
        }
    else{
        Hilfsstring2=Hilfsstring.substr(0,(int)found_d);

        Hilfsstring.erase(0,(int)found_d+1);
        Zahl=atoi(Hilfsstring2.c_str());

        hilfe=atof(Hilfsstring.c_str());
        }

    mpf_set_d(Daten.M[counter][Zahl],hilfe);

    #ifdef leu
    if(Zahl==7129){
    #endif

    #ifdef leu_quadratisch
    if(Zahl==7129){
    #endif

    #ifdef colon
    if(Zahl==2000){
    #endif

    #ifdef colon_quadratisch
    if(Zahl==2000){
    #endif
```

```
    #ifdef duke
    if(Zahl==7129){
    #endif

    #ifdef duke_quadratisch
    if(Zahl==7129){
    #endif

        found=Daten_String.find('\n');
        Daten_String.erase(0,found+(size_t)1);
        counter++;
        k++;
        }
    }

data.close();
k=KK;

Vektor_mpf_t Daten_Y(Daten.Hoehe);
Daten_Y.getSpaltenvektor(0,Daten);//Klassenindikatoren auslesen
SVM Daten_verteilt(Daten.Hoehe,Daten.Breite-1,n,t,q,q_1,k,f);
Matrix_mpf_t Daten_rechts(Daten.Hoehe,Daten.Breite-1);
Daten_rechts.Kopiere_rechts(Daten);
Daten_verteilt.Daten=Daten_rechts;

Daten_verteilt.Y.Unskaliert_einlesen(Daten_Y);
Daten_verteilt.Lambda.Erzeugung_double(1.0,t,q);//Parameter lambda, der
    die Toleranz der SVM angibt

//Berechnung der Kernel-Matrix
#ifdef leu
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef leu_quadratisch
Daten_verteilt.Quadratischer_Kernel(0.0001,100.0);
#endif

#ifdef colon
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef colon_quadratisch
Daten_verteilt.Quadratischer_Kernel(0.01,1.0);
#endif

#ifdef duke
Daten_verteilt.Linearer_Kernel();
#endif

#ifdef duke_quadratisch
Daten_verteilt.Quadratischer_Kernel(1.0,1.0);
#endif

Vektor_LA c(Daten_verteilt.Hoehe_Daten,n,t,q,q_1,k,f);
for(i=0; i<c.d; i++){
    c[i].Erzeugung_double(-1.0,t,q);
```

```
    }

Daten_verteilt.Berechne_G();
Matrix_LA G=Daten_verteilt.G;
G.Ausgabe_skaliert("G");

Vektor_LA b(2*Daten_verteilt.K.Zeilenzahl+1,n,t,q,q_1,k,f);
for(i=1; i<Daten_verteilt.K.Zeilenzahl+1; i++){
    b[i]=-Daten_verteilt.Lambda;
    }

Matrix_LA A(b.d,Daten_verteilt.Y.d,n,t,q,q_1,k,f);
for(i=1; i<Daten_verteilt.K.Zeilenzahl+1; i++){
    A.M[i][i-1].Erzeugung_double(-1.0,t,q);
    }

for(i=Daten_verteilt.K.Zeilenzahl+1; i<A.Zeilenzahl; i++){
    A.M[i][i-Daten_verteilt.K.Zeilenzahl-1].Erzeugung_double(1.0,t,q);
    }

for(i=0; i<A.Spaltenzahl; i++){
    A.M[0][i]=Daten_verteilt.Y[i]*zwei_hoch_f;
    }

Vektor_LA x(A.Spaltenzahl,n,t,q,q_1,k,f);


mpz_t epsilon;
mpz_init_set_ui(epsilon,1);
mpz_mul_2exp(epsilon,epsilon,10);

Vektor_LA AM_Zq(b.d,n,t,q,q_1,k,f);
Vektor_LA Index_AM(b.d,n,t,q,q_1,k,f);
Vektor_LA NN(b.d,n,t,q,q_1,k,f);
Vektor_LA AM_kurz(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_kurz_alt(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_kurz_Arbeit(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_kurz_Arbeit_C(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA AM_Arbeit(b.d,n,t,q,q_1,k,f);///Aktive Menge bei der ausgewählte
    , verletzte NB provisorisch hinzugefügt wird
Vektor_LA AM_Arbeit_C(b.d,n,t,q,q_1,k,f);///Komplement der temporären
    aktiven Menge

Share ff(n,t,q);

Vektor_LA s(b.d,n,t,q,q_1,k,f);
Vektor_LA S=s;///s wird noch bei der Berechnung von t_2 gebraucht
Share s_p(n,t,q);

Vektor_LA u(A.Zeilenzahl,n,t,q,q_1,k,f);//Vektor der Lagrange-
    Multiplikatoren
Vektor_LA u_plus(A.Zeilenzahl,n,t,q,q_1,k,f);//Vektor der Änderungen der
    Lagrange-Multiplikatoren

Matrix_LA Kopie_G=G;

Vektor_LA z(G.Zeilenzahl,n,t,q,q_1,k,f);
```

```
Vektor_LA tz(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_Arbeit(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_GTZ(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_GTZ_C(G.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA r_gross(A.Zeilenzahl,n,t,q,q_1,k,f);//wird für die Berechnung
    der Lagrange-Multiplikatoren und ihre Zuordnung zu den einzelnen NBen
     benötigt.

Vektor_LA t_1_Vektor(A.Zeilenzahl,n,t,q,q_1,k,f);//Zur Bestimmung von t_1
    benötigt
Vektor_LA t_1_Index(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA t_1_Index_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA u_plus_kurz(A.Spaltenzahl,n,t,q,q_1,k,f);

Matrix_LA B(A.Spaltenzahl,A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA n_plus_minus(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA n_p(A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA y(A.Spaltenzahl,n,t,q,q_1,k,f);

Vektor_LA Hilfsvektor(A.Spaltenzahl,n,t,q,q_1,k,f);
Vektor_LA Hilfsvektor2(A.Spaltenzahl,n,t,q,q_1,k,f);

Share V(n,t,q);
Share Hilfe(n,t,q);
Share qq(n,t,q);///Anzahl der aktiven Gleichungen
Share t_1(n,t,q);
Share t_1_2(n,t,q);
Share Norm_z(n,t,q);
Share t_2(n,t,q);
Share zn(n,t,q);
Share T(n,t,q);
Share T_Hilfe(n,t,q);
Share T_Vergleich_t_2(n,t,q);
Share T_Vergleich_t_1(n,t,q);
Share eins_minus_T_Vergleich_t1(n,t,q);
Share eins_minus_T_Vergleich_t2(n,t,q);
Share t_2_Vergleich_unendlich(n,t,q);//Wird in 1. Iteration noch nicht
    gebraucht
Share eins_minus_t_2_Vergleich_unendlich(n,t,q);

Share Schalter_Schritt_1(n,t,q);
Share eins_minus_Schalter_Schritt_1=Schalter_Schritt_1;
Share q_Vergleich_Null(n,t,q);
Share r_kleiner_Null(n,t,q);
Share Schalter_t_1(n,t,q);
Share eins_minus_Schalter_t_1(n,t,q);

Vektor_LA nn(A.Zeilenzahl,n,t,q,q_1,k,f);
Vektor_LA Signum_GHNB(Anzahl_Gleichungsnebenbedingungen,n,t,q,q_1,k,f);
Vektor_LA Additionshilfe(A.Zeilenzahl,n,t,q,q_1,k,f);

Vektor_LA beta(x.d-1,n,t,q,q_1,k,f);

Matrix_LA P(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
```

```
Matrix_LA P2(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_t(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA P_alt(A.Zeilenzahl,A.Zeilenzahl,n,t,q,q_1,k,f);

Matrix_LA N(A.Spaltenzahl, A.Zeilenzahl,n,t,q,q_1,k,f);///Matrizen, die
    für die Berechnung von z und r benötigt werden
Matrix_LA J(G.Zeilenzahl,G.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA R(G.Zeilenzahl,G.Zeilenzahl,n,t,q,q_1,k,f);
Matrix_LA RR(1,1,n,t,q,q_1,k,f);

Vektor_LA Nummerierung(A.Zeilenzahl,n,t,q,q_1,k,f);
mpz_t Zahl_mpz;
mpz_init(Zahl_mpz);
for(i=0; i<A.Zeilenzahl; i++){
    mpz_set_ui(Zahl_mpz,i+1);//Nummerierung startet bei 1; so wird bei
        nicht-besetzten Einträgen in Nummern_AM erkannt, dass keine
        Änderung vorzunehmen ist
    Nummerierung[i].Erzeugung(Zahl_mpz,t,q);//Für jede NB eine Nummer
        erzeugen
    }
Share Nummer(n,t,q);
Vektor_LA Nummern_AM(J.Spaltenzahl,n,t,q,q_1,k,f);

start=clock();

/**Schritt 0: Bestimme als Startwert das Minimum ohne Nebenbedingungen x=G
    ^{-1}c */

x=c;
Kopie_G.Loese_GLS_Cholesky(x.Vector_LA,x.d);
x*=-1;

c.SkalarMult(x,ff);
ff.TruncPr(k,f,t,q,q_1);
ff.Div_Konst_int(2,q,q_1,k); ///ff=1/2*c^tx

/*H=G^{-1} wird nicht bestimmt; ist in Kopie_Q implizit gespeichert.
    Aktive Menge ist leer*/

/** Schritt 1: Aktive Menge bestimmen*/

///Bestimme verletzte Nebenbedingungen
x.MatrixMult(A,s);
s.TruncPr();
s=s-b;///s=Ax-b

S=s;
for(i=0; i<Anzahl_Gleichungsnebenbedingungen; i++){
    Signum_GHNB[i]=s[i];
    Signum_GHNB[i].Signum(t,q,q_1,k);
    Signum_GHNB[i]*=-1;//neg. VZ der GHNB
    s[i]*=Signum_GHNB[i];//wenn GHNB nicht erfüllt ist, stelle sie so als
        UGL dar, dass das VZ negativ ist
    s[i]=S[i];
    S[i].Quadrat();
    S[i].TruncPr(k,f,t,q,q_1);
    S[i].LT(epsilon,t,q,q_1,k);
```

```
    S[i]=(-S[i]+1);
    V+=S[i];
    }

for(i=Anzahl_Gleichungsnebenbedingungen; i<s.d; i++)
    {
    S[i].LTZ_approximativ(t,q,q_1,k);
    V=V+S[i];//V zählt Anzahl der verletzten NBen
    }

i=V.EQZPub();
if(i==1){///Ist keine Nebenbedingung verletzt, Ende.
    stop=clock();
    Zeit=stop-start;
    printf("\nLoesung gefunden:");
    x.Ausgabe_skaliert("x");
    ff.Rekonstrukt_Ausgabe("f");
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Programmende(start,stop,x,Daten_verteilt.b,x_korrekt,b_korrekt,u,ff,
        f_korrekt,Iterationszaehler,t,q);
    //Daten_verteilt.Entscheidungsregel_Test();
    }

Iterationszaehler++;

#ifdef G_Norm
Max_G_Norm(A,Kopie_G,b,x,nn,S,s,NN,n,t,k,q,q_1,Iterationszaehler);//S: Die
      Stellen an denen eine auswählbare NB sitzt
#endif

#ifdef Euklid
Max_Eukl_Norm(A,b,x,nn,S,Signum_GHNB,n,t,k,q,q_1,
     Anzahl_Gleichungsnebenbedingungen);
#endif

for(i=0; i<A.Zeilenzahl; i++){
    for(j=0; j<A.Spaltenzahl; j++){
        n_p[j]=A.M[i][j]*nn[i]+n_p[j];
        if(i<Anzahl_Gleichungsnebenbedingungen){
            n_p[j]=n_p[j]*Signum_GHNB[i];
            }
        }
    }
Nummer=Nummerierung[nn];
Nummer.Rekonstrukt_Ausgabe_unskaliert("#");

///*Berechne Schrittrichtung (Schritt 2a)*/

y=n_p;

y.VWSubs_mit_Div(Kopie_G);
z.RWSubs(Kopie_G,y);

///*Berechne Schrittweite; im 1. Schritt ist t_1 immer gleich unendlich!*/
z.Gleich_Null_approximativ(Norm_z);
```

```
z.SkalarMult(n_p,zn);
zn.TruncPr(k,f,t,q,q_1);

s_p=s[nn];
s_p.FPDiv(zn,t,q,q_1,k);
s_p*=-1;
t_2=Norm_z*unendlich;
Hilfe=-Norm_z;
Hilfe+=1;

t_2+=Hilfe*s_p;
T=t_2;


///*Schritt 2c*/
T_Hilfe=T-unendlich;


T_Vergleich_t_2=T;
T_Vergleich_t_1=T;
eins_minus_T_Vergleich_t1=T_Vergleich_t_1;
eins_minus_T_Vergleich_t2=T_Vergleich_t_2;

i=T_Hilfe.EQZPub();
if(i==1){
    printf("\nNebenbedingungen widerspruechlich. Abbruch");
    stop=clock();
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Programmende(start,stop,x,Daten_verteilt.b,x_korrekt,b_korrekt,u,ff,
        f_korrekt,Iterationszaehler,t,q);
    exit(1);
    }

t_2_Vergleich_unendlich=t_2;//Wird in 1. Iteration noch nicht gebraucht
eins_minus_t_2_Vergleich_unendlich=-t_2_Vergleich_unendlich+1;

tz=z;
tz*=T;
tz.TruncPr(); //tz=T*z
x=x+tz;

//Bestimme verletzte Nebenbedingungen
x.MatrixMult(A,s);
s.TruncPr();
s=s-b;//s=Ax-b

///Bestimme verletzte NB gleich nach Neuberechnung von x
S=s;//s wird noch bei der Berechnung von t_2 gebraucht
V=0;
for(i=0; i<Anzahl_Gleichungsnebenbedingungen; i++){
    Signum_GHNB[i]=s[i];
    Signum_GHNB[i].Signum(t,q,q_1,k);
    Signum_GHNB[i]*=-1;//neg. VZ der GHNB
```

```
    s[i]*=Signum_GHNB[i];//wenn GHNB nicht erfüllt ist, stelle sie so als
        UGL dar, dass das VZ negativ ist
    S[i]=s[i];
    S[i].LTZ_approximativ(t,q,q_1,k);
    V+=S[i];
    }

for(i=Anzahl_Gleichungsnebenbedingungen; i<s.d; i++)
    {
    S[i].LTZ_approximativ(t,q,q_1,k);
    V=V+S[i];//V zählt Anzahl der verletzten NBen
    }

i=V.EQZPub();

zn=zn*T*T;
zn.Div_Konst_int(2,q,q_1,2*k);//Division vor TruncPr ist teurer, aber
    genauer
zn.TruncPr(2*k,2*f,t,q,q_1);


ff+=zn;//f=f+tz^tn_p(0.5t+u); (u=0) in der ersten Iteration!!!

u_plus=nn*T;//Lagrange-Multiplikator der aktivierten Gleichung wird auf T
    gesetzt (u_plus ist am Anfang Null)

u=u_plus;

/** Aktive Menge aktualisieren*/
Additionshilfe=nn;
Additionshilfe+=AM_Zq;
AM_Zq=nn;

if(i==1){//Ist keine Nebenbedingung verletzt, Ende.
    stop=clock();
    Zeit=stop-start;
    printf("\nOptimale Loesung gefunden. Ende");
    x.Ausgabe_skaliert("x");
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Programmende(start,stop,x,Daten_verteilt.b,x_korrekt,b_korrekt,u,ff,
        f_korrekt,Iterationszaehler,t,q);
    }

qq=1;

///Aktive Menge kompaktifizieren
/** N^{\ast},H aktualisieren!!!*/
P.Ersetze_Zeile(AM_Zq.Vector_LA,0,AM_Zq.d);//Das so definierte P schiebt
    als Permutationsmatrix die 1. aktivierte NB an die Stelle 1
AM_kurz[0].Erzeugung(eins,t,q);//Nach dem 1. Schritt is genau eine NB
    aktiv; im verkürzten Vektor AM_kurz steht diese an Stelle 0

int check;
check=Berechne_J(J,RR,A,B,Kopie_G,nn,n,t,k,q,q_1,Iterationszaehler);//nn
    statt AM_Zq, da im ersten Schritt nur eine NB aktiv ist
```

```
Nummern_AM[0]=Nummer;

AM_Arbeit=AM_Zq;///Aktive Menge bei der ausgewählte, verletzte NB
    provisorisch hinzugefügt wird
AM_Arbeit_C=AM_Arbeit;///Komplement der temporären aktiven Menge

for(Iterationszaehler=3; Iterationszaehler<50; Iterationszaehler++){

///Bestimme verletzte Nebenbedingungen – Verschoben ans Ende; nach der
    Neuberechnung

#ifdef G_Norm
Max_G_Norm(A,Kopie_G,b,x,nn,S,s,NN,n,t,k,q,q_1,Iterationszaehler);
#endif

#ifdef Euklid
Max_Eukl_Norm(A,b,x,nn,S,Signum_GHNB,n,t,k,q,q_1,
    Anzahl_Gleichungsnebenbedingungen);
#endif

#ifdef FindFirst
S.FindFirst(nn);
#endif

AM_Arbeit=AM_Zq+nn;
for(i=0; i<AM_Arbeit.d; i++){
    AM_Arbeit_C[i]=AM_Arbeit[i];
    AM_Arbeit_C[i]*=-1;
    AM_Arbeit_C[i]+=1;
    }//Bestimmt Komplement der temporären aktiven Menge

Index_AM=AM_Arbeit;
Index_AM.Vektor_Zusammenschieben_binaer(AM_Arbeit.Vector_LA,P2);//
    Aktualisiere die Matrix P, mit deren Hilfe die aktiven Ungleichungen
    kompaktifziert werden

AM_kurz_Arbeit.Oben_einpassen(Index_AM);

for(i=0; i<AM_kurz_Arbeit.d; i++){
    AM_kurz_Arbeit_C[i]=AM_kurz_Arbeit[i];
    AM_kurz_Arbeit_C[i]*=-1;
    AM_kurz_Arbeit_C[i]+=1;
    }

n_p=0;
for(i=0; i<A.Zeilenzahl; i++){
    for(j=0; j<A.Spaltenzahl; j++){
        n_p[j]=A.M[i][j]*nn[i]+n_p[j];
        if(i<Anzahl_Gleichungsnebenbedingungen){
            n_p[j]=n_p[j]*Signum_GHNB[i];//VZ bei GHNB ggf umdrehen
            }
        }
    }
Nummer=Nummerierung[nn];//Nummer der NB auslesen

/** Berechne Schrittrichtung (Schritt 2a)*/
```

```
int v=haupt::min(Iterationszaehler,G.Zeilenzahl);
Matrix_LA R_klein(v-1,v-1,n,t,q,q_1,k,f);//Größe hängt von Iterationszahl
    ab; muß deshalb in jeder Runde neu erzeugt werden
R_klein.Links_oben_einpassen(R);
if(Iterationszaehler==3){
    check=Berechne_z_r(J,RR,n_p,AM_kurz,z,r,n,t,q,q_1,k,Iterationszaehler)
        ;
    }
else
    check=Berechne_z_r(J,R_klein,n_p,AM_kurz,z,r,n,t,q,q_1,k,
        Iterationszaehler);
/** Berechne Schrittweite; im 1. Schritt ist t_1 immer gleich unendlich!*/

z.Gleich_Null_approximativ(Norm_z);

z.SkalarMult(n_p,zn);
zn.TruncPr(k,f,t,q,q_1);
Hilfe=zn;

/** Berechne t_1*/
q_Vergleich_Null=qq;
q_Vergleich_Null.EQZ(t,q,q_1,k);
r_kleiner_Null=r[0];
r_kleiner_Null.GTZ_approximativ(t,q,q_1,k);
r_GTZ[0]=r_kleiner_Null;
for(i=1; i<r.d; i++){
    r_GTZ[i]=r[i];
    r_GTZ[i].GTZ_approximativ(t,q,q_1,k);
    r_kleiner_Null=r_kleiner_Null+r_GTZ[i]-r_kleiner_Null*r_GTZ[i];//
        r_kleiner_Null ist 1, wenn mindestens eine Komponente größer als
        0 ist
    }

Schalter_t_1=r_kleiner_Null*(-q_Vergleich_Null+1);

r_gross=0;
r_gross.Oben_einpassen(r);

P_t=P;
P_t.Transponiere_quadratisch();

r_gross.MatrixMult(P_t,r_gross);//Ordne die r's ihren Indizes zu

t_1_Vektor=u_plus;
t_1_Vektor.MatrixMult(P,t_1_Vektor);//permutiere die Einträge von u^+ an
    die oberen Stellen

u_plus_kurz.Oben_einpassen(t_1_Vektor);

for(i=0; i<r.d; i++){
    r_GTZ_C[i]=r_GTZ[i];
    r_GTZ_C[i]*=-1;
    r_GTZ_C[i]+=1;
    }

for(i=0; i<A.Spaltenzahl; i++){// TODO (LocalAdmin#1#): Kriteriumsvektor
    verwenden!
```

```
    u_plus_kurz[i]=u_plus_kurz[i]*AM_kurz_Arbeit[i]+AM_kurz_Arbeit_C[i]*
        unendlich;
    u_plus_kurz[i]=r_GTZ[i]*u_plus_kurz[i]+r_GTZ_C[i]*unendlich;
    r_Arbeit[i]=r[i]*AM_kurz_Arbeit[i]+AM_kurz_Arbeit_C[i]*zwei_hoch_f;//
        r_gross wird später noch für die Aktualisierung von u benötigt
    r_Arbeit[i]=r_Arbeit[i]*r_GTZ[i]+r_GTZ_C[i]*zwei_hoch_f;
    }


u_plus_kurz.Min_Bruch(r_Arbeit,t_1_Index_kurz);
t_1_2=u_plus_kurz[t_1_Index_kurz];
t_1_2.FPDiv(r_Arbeit[t_1_Index_kurz],t,q,q_1,k);
t_1=t_1_2;

t_1_Index=0;
t_1_Index.Oben_einpassen(t_1_Index_kurz);
t_1_Index.MatrixMult(P_t,t_1_Index);//Index von t_1 an die richtige Stelle
    permutieren

/** Berechne t_2*/
s_p=s[nn];
s_p.FPDiv(zn,t,q,q_1,k);
s_p*=-1;
t_2=Norm_z*unendlich;
Hilfe=-Norm_z;
Hilfe+=1;

t_2+=Hilfe*s_p;

T.min(t_1,t_2,q,q_1,t,k,Hilfe);

/**Schritt 2c*/
T_Hilfe=T;
T_Hilfe-=unendlich;
i=T_Hilfe.EQZPub();

if(i==1){
    stop=clock();
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Programmende(start,stop,x,Daten_verteilt.b,x_korrekt,b_korrekt,u,ff,
        f_korrekt,Iterationszaehler,t,q);
    printf("\nNebenbedingungen␣widerspruechlich.␣Abbruch");
    exit(1);
    }

t_2_Vergleich_unendlich=t_2;
t_2_Vergleich_unendlich.EQ_mpz(unendlich,t,q,q_1,k);//t_2=?unendlich
eins_minus_t_2_Vergleich_unendlich=-t_2_Vergleich_unendlich;
eins_minus_t_2_Vergleich_unendlich+=1;

T_Vergleich_t_1=Hilfe;//Hilfe==1 <=> T=t_1, sonst 0
T_Vergleich_t_2=-Hilfe+1;

eins_minus_T_Vergleich_t1=T_Vergleich_t_1;
eins_minus_T_Vergleich_t2=T_Vergleich_t_2;
```

```
eins_minus_T_Vergleich_t1*=-1;
eins_minus_T_Vergleich_t1+=1;
eins_minus_T_Vergleich_t2*=-1;
eins_minus_T_Vergleich_t2+=1;

///xNeu
tz=z;
tz*=T;
tz.TruncPr(); //tz=T*z

x+=tz;
x*=eins_minus_t_2_Vergleich_unendlich;
Hilfsvektor=x*t_2_Vergleich_unendlich;
x+=Hilfsvektor;

///fNeu
zn*=T;//wird benötigt für die Aktualisierung von f
zn.TruncPr(k,f,t,q,q_1);
Hilfe=u_plus[nn];//Hilfe=u^+_{q+1}
Hilfe*=2;
Hilfe+=T;
Hilfe.Div_Konst_int(2,q,q_1,k);
zn*=Hilfe;
zn.TruncPr(k,f,t,q,q_1);

ff=eins_minus_t_2_Vergleich_unendlich*(ff+zn)+t_2_Vergleich_unendlich*ff;;
     //f=f+tz^tn_p(0.5t+u); (u=0) in der ersten Iteration!!!

///uNeu
u_plus=u_plus+nn*T;

r_gross*=T;
r_gross.TruncPr(k,f);

u_plus-=r_gross;//u^+=u^+t(-r 1)^t

u=u_plus*T_Vergleich_t_2+u*eins_minus_T_Vergleich_t2;//If T=t_2 ->u=u_plus

///Aktualisierung der aktiven Menge
Additionshilfe=nn+AM_Zq;

AM_Zq=AM_Zq*eins_minus_T_Vergleich_t2+Additionshilfe*T_Vergleich_t_2;//
     neuen Index zur aktiven Menge hinzufügen

Additionshilfe=AM_Zq-t_1_Index;
AM_Zq=AM_Zq*eins_minus_T_Vergleich_t1+Additionshilfe*T_Vergleich_t_1;

qq=(qq+1)*T_Vergleich_t_2+(qq-1)*eins_minus_T_Vergleich_t2; //
     Aktualisierung des Zählers der aktiven Menge

///Überprüfen der Zulässigkeit des neuen Punkts

x.MatrixMult(A,s);
s.TruncPr();
s=s-b;//s=Ax-b

S=s;//s wird noch bei der Berechnung von t_2 gebraucht
```

```
V=0;
for(i=0; i<Anzahl_Gleichungsnebenbedingungen; i++){
    Signum_GHNB[i]=s[i];
    Signum_GHNB[i].Signum(t,q,q_1,k);
    Signum_GHNB[i]*=-1;//neg. VZ der GHNB
    s[i]*=Signum_GHNB[i];//wenn GHNB nicht erfüllt ist, stelle sie so als
        UGL dar, dass das VZ negativ ist
    S[i]=s[i];
    S[i].LTZ_approximativ(t,q,q_1,k);
    V+=S[i];
    }

for(i=Anzahl_Gleichungsnebenbedingungen; i<s.d; i++)
    {
    S[i].LTZ_approximativ(t,q,q_1,k);
    V=V+S[i];//V zählt Anzahl der verletzten NBen
    }


i=V.EQZPub();
if(i==1){//Ist keine Nebenbedingung verletzt, Ende.
    stop=clock();
    Zeit=stop-start;
    printf("\nOptimaler␣Punkt␣erreicht.");
    x.Ausgabe_skaliert("x");
    ff.Rekonstrukt_Ausgabe("f");
    Daten_verteilt.Alpha=x;
    Daten_verteilt.Berechne_SV();
    Daten_verteilt.Berechne_b();
    Programmende(start,stop,x,Daten_verteilt.b,x_korrekt,b_korrekt,u,ff,
        f_korrekt,Iterationszaehler,t,q);
    break;
    }

/** N^{\ast},H aktualisieren!!!*/
int check;

Index_AM=AM_Zq;
P_alt=P;
Index_AM.Vektor_Zusammenschieben_binaer(AM_Zq.Vector_LA,P);

AM_kurz_alt=AM_kurz;
AM_kurz.Oben_einpassen(Index_AM);

Matrix_LA R_gross(v,v,n,t,q,q_1,k,f);
if(A.Spaltenzahl<=3){//Bei Größe 3 müssten bei Aktualisierung Matrizen der
      Größe 2 UND 3 berechnet werden. Da ist Neuberechnung billiger
    check=Berechne_J(J,R_gross,A,B,Kopie_G,AM_Zq,n,t,k,q,q_1,
        Iterationszaehler);
    }
else{
    if(Iterationszaehler==3)
        J.Ausgabe_skaliert("J");
    Hilfe=qq-1;//qq ist um eins größer als die Spalte, die es indiziert
    Hilfsvektor.BitMask(Hilfe);//HV ist Bitvektorkodierung der Größe der
        aktiven Menge
```

```
    t_1_Index.MatrixMult(P_alt,Additionshilfe);//Additionshilfe=P*
        t_1_Index; multipliziere t_1_Index an eine vordere Position;
        verwende P_alt, da neues P auf t_1_I nicht anspricht

    Hilfsvektor2.Oben_einpassen(Additionshilfe);//AH ist einer der ersten
        Einträge

    Hilfsvektor=Hilfsvektor*T_Vergleich_t_2;//HV ist Bitvektorkodierung
        von qq, wenn neue NB hinzugefügt wird (dann steht die Spalte - qq
         - an der sie in J eingetragen wir, fest)
    Hilfe=-T_Vergleich_t_2+1;
    Hilfsvektor+=(Hilfsvektor2*Hilfe);//If T=t_2 -> HV<-HV else HV<-HV2

    n_plus_minus=(n_p*T_Vergleich_t_2);//T!=t_2 ->n+- = 0 und somit wird
        in B die 0-Spalte substituiert

    AM_kurz_Arbeit.Oben_einpassen(Additionshilfe);
    AM_kurz_Arbeit=AM_kurz_alt-AM_kurz_Arbeit;//Aus alter, verkürzter AM
        gelöschten Index entfernen
    AM_kurz_Arbeit=AM_kurz*T_Vergleich_t_2+AM_kurz_Arbeit*T_Vergleich_t_1;
        //neue verkürzte AM übergeben, wenn NB hinzugefügt wurde; sonst
        manipulierte alte, verkürzte AM
    Nummer=Nummer*T_Vergleich_t_2+t_1_Index[Nummerierung]*T_Vergleich_t_1;
        //Index der zu löschenden Spalte übergeben, wenn T=t_1
    T_Vergleich_t_1.Rekonstrukt_Ausgabe_unskaliert("T=?t_1");
    check=Aktualisiere_J(J,R_gross,A,Kopie_G,B,P,n_plus_minus,Nummer,
        Nummern_AM,Hilfsvektor,AM_kurz_Arbeit,n,t,k,q,q_1,
        Iterationszaehler);
    }
R.Links_oben_einpassen(R_gross);
}

Daten_verteilt.Alpha=x;//Lösungsvektor in den Scope der SVM schreiben
Daten_verteilt.Berechne_SV();
Daten_verteilt.SV.Ausgabe("SV");
Daten_verteilt.Entscheidungsregel_Test("Ausgabe.txt");

ifstream data_test;

#ifdef leu
Daten_verteilt.Save_SVM("leu");
Matrix_mpf_t Daten_Test(34,7130);
data_test.open("leu.t.scale");
#endif

#ifdef leu_quadratisch
Daten_verteilt.Save_SVM("leu");
Matrix_mpf_t Daten_Test(34,7130);
data_test.open("leu.t.scale");
#endif

#ifdef colon
Daten_verteilt.Save_SVM("colon");
Matrix_mpf_t Daten_Test(32,2001);
data_test.open("cc-test.scale");
#endif
```

```
#ifdef colon_quadratisch
Daten_verteilt.Save_SVM("colon");
Matrix_mpf_t Daten_Test(32,2001);
data_test.open("cc-test.scale");
#endif

#ifdef duke
Daten_verteilt.Save_SVM("duke");
Matrix_mpf_t Daten_Test(4,7130);
data_test.open("duke.val.scale");
#endif

#ifdef duke_quadratisch
Daten_verteilt.Save_SVM("duke");
Matrix_mpf_t Daten_Test(4,7130);
data_test.open("duke.val.scale");
#endif

if(!data_test)
    {
    cout<<"Datei konnte nicht geoeffnet werden. Abbruch"<<endl;
    exit(1);
    }

string Daten_String_Test;
string Hilfsstring_Test;
string Hilfsstring2_Test;


while(data_test.get(C)){
    Daten_String_Test+=C;//Input in Daten_String schreiben
    }

//cout<<Daten_String_Test;

k=0;
counter=0;
hilfe=0.0;

while(!Daten_String_Test.empty()){
    found=Daten_String_Test.find(' ');
    Hilfsstring_Test=Daten_String_Test.substr(0,(int)found);
    Daten_String_Test.erase(0,found+(size_t)1);

    found_d=Hilfsstring_Test.find(":");
    if(found_d==string::npos){
        hilfe=atof(Hilfsstring_Test.c_str());
        Zahl=0;
        }
    else{
        Hilfsstring2_Test=Hilfsstring_Test.substr(0,(int)found_d);

        Hilfsstring_Test.erase(0,(int)found_d+1);
        Zahl=atoi(Hilfsstring2_Test.c_str());

        hilfe=atof(Hilfsstring_Test.c_str());
        }
```

```
    mpf_set_d(Daten_Test.M[counter][Zahl],hilfe);

    #ifdef leu
    if(Zahl==7129){
    #endif

    #ifdef leu_quadratisch
    if(Zahl==7129){
    #endif

    #ifdef colon
    if(Zahl==2000){
    #endif

    #ifdef colon_quadratisch
    if(Zahl==2000){
    #endif

    #ifdef duke
    if(Zahl==7129){
    #endif

    #ifdef duke_quadratisch
    if(Zahl==7129){
    #endif

        found=Daten_String_Test.find('\n');
        Daten_String_Test.erase(0,found+(size_t)1);
        counter++;
        k++;
        }
    }

data_test.close();

Vektor_mpf_t Daten_Y_Test(Daten_Test.Hoehe);
Daten_Y_Test.getSpaltenvektor(0,Daten_Test);//Klassenindikatoren auslesen
SVM Daten_verteilt_Test(Daten_Test.Hoehe,Daten_Test.Breite-1,n,t,q,q_1,k,f
    );
Matrix_mpf_t Daten_rechts_Test(Daten_Test.Hoehe,Daten_Test.Breite-1);
Daten_rechts_Test.Kopiere_rechts(Daten_Test);
Daten_verteilt_Test.Daten=Daten_rechts_Test;

Daten_Y_Test.Ausgabe("Y");
Daten_verteilt_Test.Y.Unskaliert_einlesen(Daten_Y_Test);
Daten_verteilt_Test.Y.Ausgabe("Y_s");
Daten_verteilt_Test.Lambda=Daten_verteilt.Lambda;//Parameter lambda, der
    die Toleranz der SVM angibt

printf("\n\nZeit fuer Berechnung der SVM: %f s",(double)Zeit/
    CLOCKS_PER_SEC);

start=clock();
#ifdef leu
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
    Daten_verteilt_Test.Y,"Ausgabe.txt");
```

```
#elif colon
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
     Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif duke
Daten_verteilt.SVM_Test_linear(Daten_verteilt_Test.Daten,
     Daten_verteilt_Test.Y,"Ausgabe.txt");
#endif

#ifdef leu_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
     Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif colon_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
     Daten_verteilt_Test.Y,"Ausgabe.txt");

#elif duke_quadratisch
Daten_verteilt.SVM_Test_quadratisch(Daten_verteilt_Test.Daten,
     Daten_verteilt_Test.Y,"Ausgabe.txt");
#endif
stop=clock();

FILE * datei1;
printf("\n\nZeit␣fuer␣Verifikation:␣%f␣s",((double)stop-(double)start)/
     CLOCKS_PER_SEC);
datei1=fopen("Ausgabe.txt","a");
fprintf(datei1,"\nZeit␣fuer␣Verifikation:␣%f␣s",((double)stop-(double)
     start)/CLOCKS_PER_SEC);
getchar();
return 0;
}
```