Eine Implementierung von Algorithmen der linearen Algebra über $\mathbb{Z}$

Diplomarbeit

von

Patrick Theobald

nach einem Thema von

Prof. Dr. Buchmann

am Fachbereich 14

Informatik

der Universität des Saarlandes

25. September 1995

Eidesstattliche Erklärung

„Hiermit versichere ich an Eides statt, daß ich diese Arbeit selbständig und nur unter Verwendung der angegebenen Quellen angefertigt habe.“

Saarbrücken, den 28. Februar 1995 Patrick Theobald

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während der Entstehung dieser Diplomarbeit unterstützt haben.

Mein Dank gebührt zunächst Prof. Dr. Buchmann, der dieses interessante Thema vergeben und mich durch Hinweise und Anregungen beraten hat.

Desweiteren möchte ich mich bei Thomas Papanikolaou und Stefan Neis bedanken. Sie haben mich kontinuierlich durch Ideen und Anregungen bei der Implementierung und der Erstellung dieser Arbeit vorangebracht.

Schließlich gilt mein Dank insbesondere Anja Jantke und meiner Familie. Sie haben mir während des Studiums den nötigen Rückhalt gegeben und mich nach besten Kräften unterstützt.

Inhaltsverzeichnis

Teil I

Einführung

Kapitel 1

Einleitung

Gegenstand der vorliegenden Arbeit ist eine C++ Implementierung von Algorithmen der linearen Algebra über dem Ring der rationalen ganzen Zahlen.

Als theoretische Grundlage dieser Implementierung dient uns die Diplomarbeit „*Effiziente Algorithmen für Probleme der linearen Algebra über* $\mathbb{Z}$“ von Achim Müller. Wir verwenden seine Notationen, Definitionen und Algorithmen als Basis unserer weiteren Betrachtungen.

Die grundlegenden mathematischen Strukturen der linearen Algebra über $\mathbb{Z}$ sind *$\mathbb{Z}$-Moduln*. Untersuchungsgegenstand sind *Modulhomomorphismen* und ihre Eigenschaften. Da sowohl Moduln als auch Homomorphismen *mathematische Objekte* sind, sind sie für genauere algorithmische Untersuchungen ungeeignet. Deshalb müssen wir Fragestellungen und Untersuchungen, die sich aus der Theorie dieser Objekte ergeben, auf eine angemessene *Darstellung* derselben übersetzen.

In der Literatur hat sich die Darstellung eines Homomorphismus bzw. einer linearen Abbildung durch die zugehörige *Abbildungsmatrix* und die Darstellung eines Moduls durch eine im allgemeinen nicht eindeutig bestimmte *Basis* durchgesetzt. In Anlehnung hieran haben wir zur Darstellung von Homomorphismen und Moduln in unserer Implementierung als Basisdatenstruktur das Anordnungsschema *Matrix* gewählt.

Im Hinblick auf Homomorphismen symbolisiert eine Matrix die entsprechende Abbildungsmatrix bzgl. der Standardbasis. Dagegen soll sie im Bezug auf Moduln als eine Sammlung von Vektoren, die den Modul erzeugen, aufgefaßt werden. Bei der Implementierung von Algorithmen der linearen Algebra über $\mathbb{Z}$ handelt es sich also nach unserem Verständnis um die Implementierung einer Matrixklasse über $\mathbb{Z}$. Unsere Matrixklasse haben wir *bigint_matrix* genannt. Der Name entstand in Anlehnung an die zugrundeliegende Arithmetik für große Zahlen *bigint* und unter Berücksichtigung der Nomenklatur des Computer–Algebra–Systems LiDIA, von dem diese Implementierung ein Teil ist.

Die Matrixklasse *bigint_matrix* ist konzeptionell für die Bewältigung vieler Problemstellungen der linearen Algebra über $\mathbb{Z}$ ausgelegt. Wir stellen nun von einem eher anwendungsbezogenen Standpunkt aus einige Aspekte des Leistungsumfangs dieser Klasse vor.

Seien ϕ und φ Homomorphismen und A und B $\mathbb{Z}$-Moduln.

- Im Hinblick auf Homomorphismen ist die Klasse *bigint_matrix* in der Lage,

  1. die Hintereinanderausführung $\phi \circ \varphi$,
  2. die Summe $\phi + \varphi$,
  3. die Differenz $\phi - \varphi$,
  4. den Rang $rank(\phi)$,
  5. den adjungierten Homomorphismus $adj(\phi)$,
  6. die Determinante $det(\phi)$ und
  7. das charakteristische Polynom $charpoly(\phi)$

  zu berechnen.

- Desweiteren kann sie für Moduln

  1. die Summe $A + B$,
  2. den Rang $rank(A)$ und
  3. die Indizes der linear unabhängigen Vektoren $lininc(A)$

  berechnen.

- Schließlich erlaubt die Klasse *bigint_matrix* im Hinblick auf die Beziehungen zwischen Modulhomomorphismen und Moduln die Berechnung

  1. einer Basis des Kerns,
  2. einer Basis des Bildes und
  3. einer Darstellung des Urbildes

  bezüglich eines Homomorphismus.

Neben den Funktionen, die dem hier angegebenen Leistungsspektrum entsprechen, haben wir im Zuge einer geeigneten Modularisierung eine Vielzahl zusätzlicher Funktionen implementiert. Dies hat zum einen speicherverwaltungstechnische Gründe, zum anderen verfolgen wir damit die Zielsetzung, die Implementierung abgeleiteter Klassen, wie zum Beispiel einer Modulklasse mit eindeutig bestimmter Basis (Hermite–Normalform–Basis und Gitterdeterminante), weitgehend zu unterstützen.

Die C++ Implementierung von Algorithmen der linearen Algebra über $\mathbb{Z}$ in der Matrixklasse *bigint_matrix* orientiert sich an zwei Zielsetzungen:

1. Die Berechnungen in dieser Klasse sollen für solche Homomorphismen bzw. Moduln optimiert sein, deren Darstellungsmatrizen dicht besetzt sind. Dies ist, wie wir im Teil *Datenstruktur und Basisfunktionen einer Matrixklasse* sehen werden, für die interne Datenstruktur von großer Bedeutung.

2. Desweiteren sollen die implementierten Funktionen effizienter als entsprechende Funktionen anderer Implementierungen zu diesem Themengebiet wie PARI, MATHEMATICA und MAPLE sein.

Wir werden im Verlauf der Arbeit anhand von Laufzeitmessungen bezüglich Funktionsausführungen auf dicht besetzten Matrizen zeigen, daß wir diese beiden Zielsetzungen durchaus erfüllt haben.

Kapitel 2

Gliederung der Arbeit und Bezeichnungen

2.1 Gliederung

Die Gliederung der weiteren Arbeit lehnt sich sehr stark an die Aufschlüsselung des Funktionsumfangs der Klasse *bigint_matrix* aus der Sicht des Programmierers an. Die Gesamtarbeit unterteilt sich wie folgt in fünf Teile.

Im weiteren Verlauf dieses Einleitungsteils werden wir zunächst einige Bezeichnungen vereinbaren, die wir in den folgenden Teilen dann nicht mehr explizit einführen werden. Desweiteren werden wir das Computer–Algebra–System LiDIA vorstellen, in welchem die Klasse *bigint_matrix* enthalten ist und welches zur Zeit (1995) am Lehrstuhl von Prof. Dr. J. Buchmann an der Universität des Saarlandes entsteht.

Im zweiten Teil der Arbeit stellen wir die Datenstruktur vor und erläutern, weshalb wir uns gerade für diese Darstellungsform entschieden haben. Ebenso beschreiben wir die Basisfunktionen der Matrixklasse. Hinter dem Begriff Basisfunktionen verbergen sich dabei Konstruktoren, Destruktoren, Struktur–, Zugriffs– und Tauschfunktionen. Diese bilden die Grundlage aller weiteren Funktionen und sind somit für die folgenden Teile von Bedeutung.

Im dritten Teil beschreiben wir die implementierten arithmetischen Funktionen und Operatoren der Klasse *bigint_matrix*. Dies sind im wesentlichen Funktionen zur Addition, Subtraktion, Multiplikation, Restbildung, Zuweisung und einige Boolesche Funktionen. Außerdem geben wir erste tabellarische Laufzeitvergleiche mit anderen Computer–Algebra–Systemen wie PARI, MATHEMATICA und MAPLE an.

Im vierten Teil kommen wir zum eigentlichen Kern der Arbeit. Wir erläutern anhand eines Beispiels das Hauptproblem *entry explosion*, daß sich bei Algorithmen der linearen Algebra über $\mathbb{Z}$ stellt. Obwohl dieser Themenkreis in der Diplomarbeit von Achim Müller bereits ausführlich behandelt wurde, gehen wir an dieser Stelle nochmals darauf ein. Einerseits geschieht dies, um die Unterteilung dieses Teils in die Kapitel *Modulare Arithmetik I*, *Modulare Arithmetik II* und *Nichmodulare Algorithmen* zu motivieren. Andererseits wollen wir neuere Erkenntnisse aus [12] und [14] vorstellen. Wir streben dabei keine theoretische Detailbetrachtung an, sondern wollen lediglich dem Leser drei Methoden zur Lösung des Problems der *entry explosion* vorstellen und auf Funktionen eingehen, die mittels dieser Methoden implementiert wurden. Wie im dritten Teil belegen wir die Effizienz der implementierten Funktionen durch Laufzeitvergleiche mit anderen Systemen.

Im fünften Teil widmen wir uns der Mensch–Maschine–Schnittstelle dieser Klasse oder anders gesagt den Ein– und Ausgabefunktionen bzw. -operatoren. Wir legen die unterstützten Formate offen und zeigen anhand einfacher Beispiele deren Verwendung im Zusammenspiel mit anderen Computer–Algebra–Systemen.

Im Anhang dieser Arbeit gehen wir kurz auf Tests und Handler im Zusammenhang mit der Einbettung der Klasse *bigint_matrix* in LiDIA ein. Desweiteren geben wir eine Beispielmatrix an, die sich im Zusammenhang mit *entry explosion* bei der Berechnung von Normalformen ganzzahliger Matrizen bewährt haben. Ebenfalls sind die Seiten des LiDIA–Handbuches angefügt, die sich mit dieser hier vorgestellten Klasse beschäftigen.

2.2 Bezeichnungen

Im weiteren Verlauf der Arbeit verwenden wir die folgenden Bezeichnungen, die wir später nicht mehr explizit einführen werden.

- Zur Bezeichnung verschiedener Zahlbereiche verwenden wir die folgenden Symbole.

$\mathbb{N}$	Menge der natürlichen Zahlen (ohne Null)
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{P}$	Menge der Primzahlen

- Für eine Matrix $A \in \mathbb{Z}^{m \times n}$ und $1 \leq i \leq m$, $1 \leq j \leq n$ bezeichnen wir mit
  - $a_{i,j}$ den Eintrag der i–ten Spalte und j–ten Zeile der Matrix A,
  - $a_{*,j}$ die j–te Spalte der Matrix A,
  - $a_{i,*}$ die i–te Zeile der Matrix A und
  - $|A|$ den betragsmäßig größten Eintrag der Matrix A.
- Für eine Menge M ganzer Zahlen sei $\#M$ die Anzahl der Elemente der Menge M.

- Für einen Vektor $v = (v_0, \ldots, v_{n-1})$ und eine ganze Zahl $k \geq 0$ sei die L_k–Metrik:

$$L_k(v) = \left(\sum_{l=0}^{n-1} |v_l|^k\right)^{\frac{1}{k}}$$

- Für einen Eintrag $a_{i,j} \in A$ und $k \geq 0$ bezeichnen wir mit

  – $|a_{i,j}|_k^R$ .... die Berechnung $\left(\sum_{l=0}^{n-1} |a_{i,l}|^k\right)^{\frac{1}{k}}$,

  – $|a_{i,j}|_k^C$ .... die Berechnung $\left(\sum_{l=1}^{m-1} |a_{l,j}|^k\right)^{\frac{1}{k}}$.

Kapitel 3

LiDIA

In diesem Kapitel befassen wir uns mit LiDIA, einer neuen **Bibliothek für algorithmische Zahlentheorie** [1]. Wir unterteilen dieses Kapitel in vier Abschnitte. Im ersten Abschnitt werden wir ein Problem beleuchten, welches letztlich die Zielsetzung und die Konzeption des Pakets LiDIA prägt. Anschließend werden wir im zweiten Abschnitt die Zielsetzung und im dritten das Konzept der Bibliothek LiDIA vorstellen. Im letzten Abschnitt beschreiben wir kurz die Grundbausteine dieser hier vorgestellten neuen Bibliothek. Aus diesen Betrachtungen soll klar werden, welche Aufgabe die Klasse *bigint_matrix* in diesem Paket erfüllt und auf welcher Software–Grundlage die Klasse steht.

3.1 Einführung

In den letzten Jahren wurde am Lehrstuhl von Prof. Dr. J. Buchmann an vielen Problemen der algorithmischen Zahlentheorie gearbeitet. Zu diesen Problemen gehören neben anderen

- die Faktorisierung ganzer Zahlen,
- das Zählen der Anzahl von Punkten auf elliptischen Kurven,
- die Berechnung der Klassenzahl eines Zahlkörper und
- die Berechnung diskreter Logarithmen über endlichen Körpern.

Dazu wurde Software für viele grundlegende Aufgaben benötigt, wozu unter anderem eine Multi–Precision–Arithmetik und Routinen der linearen Algebra über $\mathbb{Z}$ zählen. Die Wiederverwendung von Softwaremodulen eines Projekts in einem anderen war jedoch äußerst schwierig. Entweder war die Dokumentation unzureichend oder nicht vorhanden, oder in unterschiedlichen Projekten wurden unterschiedliche Multi–Precision–Arithmetik–Pakete und Speicherverwaltungen verwendet. Daher gab es oft zu einem Algorithmus verschiedene Implementierungen, die nicht immer auf die effizienteste Art und Weise vorgenommen wurden.

3.2 Zielsetzung

Um das Problem der Wiederverwendbarkeit von Software zu lösen, wurde Anfang 1994 das LiDIA–Projekt ins Leben gerufen und beschlossen, das Softwarepaket LiDIA als eine C++ Bibliothek zu konzipieren.

Das Hauptanliegen des LiDIA–Projektes ist es, möglichst effizienten Code zu entwickeln. Dazu werden die Laufzeiten von LiDIA mit denen anderer Computer-Algebra-Systeme, wie z.B. PARI, MATHEMATICA, MAPLE, verglichen.

3.3 Konzept

Mittels einer objektorientierten Sprache ist es möglich, Objekte zu kreieren, die mathematischen Objekten entsprechen. Damit können Algorithmen auf einem geeigneten Abstraktionsgrad implementiert werden, die dann dem mathematischen Formalismus sehr ähnlich sehen. Bis zu diesem Zeitpunkt wurde ein Großteil der Software, die am Lehrstuhl entwickelt wurde, in C geschrieben. Daher wird die Entscheidung, das LiDIA–Paket in C++ zu realisieren, verständlich.

Um eine möglichst einfache Portabilität zu garantieren, sieht das LiDIA–Konzept einen relativ geringen maschinenabhängigen Kern vor. Gegenwärtig enthält der Kern die Multi–Precision–Arithmetik für Integers. In der nächsten Version soll außerdem eine Speicherverwaltung darin enthalten sein.

Es ist ein ernstes Problem, ein geeignetes Multi–Precions–Arithmetik-Paket für Integers und eine geeignete Speicherverwaltung für LiDIA auszuwählen. Einige Pakete laufen auf bestimmten Architekturen effizienter, andere auf anderen Architekturen. Außerdem werden neue Architekturen zu neuen Multi–Precision–Arithmetik–Paketen führen. Daher haben wir beschlossen, LiDIA unabhängig von speziellen Kernen zu gestalten, aber dennoch ein einfaches Ersetzen des LiDIA–Kerns zu ermöglichen. Um das zu realisieren, wurde der Kern von den Anwenderprogrammen mittels einer Schnittstelle, in der sich Deklarationen, Operatoren, Funktionen und Prozeduren mit Multi–Precion–Arithmetik beschäftigen, getrennt und die Speicherverwaltung (memory management) standardisiert. Alle LiDIA–Klassen nutzen den Kern über diese Schnittstelle, d.h. die Kernfunktionen werden nicht direkt angesteuert. Damit wird ein Wechsel vom aktuellen Kern zu einem neuen Kern relativ einfach, weil nur die Schnittstelle in geeigneter Art und Weise modifiziert werden muß, nicht aber die LiDIA–Klassen.

3.4 Überblick

Im weiteren werden wir kurz den Inhalt von LiDIA in der Version 1.0 beleuchten. Der Leser wird dabei erkennen, daß die Klasse *bigint_matrix*, die wir ausführlich in dieser Arbeit behandeln, ein Bestandteil dieses Funktionsumfangs ist. Eine etwas ausführlichere Beschreibung des LiDIA–Pakets findet der daran interessierte Leser im Handbuch zu LiDIA, welches im LiDIA–Paket enthalten ist, oder in [1].

3.4.1 Der Kern

Der Kern von LiDIA enthält das Multi–Precision–Arithmetik–Paket **libI** [6] für Integers, das von allen LiDIA–Klassen genutzt wird. Die minimalen Anforderungen, die LiDIA an dieses Paket stellt, sind folgende.
Das Paket muß Zuweisungen, arithmetische Operationen (Addition, Subtraktion, Multiplikation, Division), Vergleichsoperationen ($==, \neq, >, <, \geq, \leq$), den Vergleich mit Null und Bit–Operationen (not, and, or, xor) unterstützen. Zudem soll es Routinen für das Ziehen der Quadratwurzel, für den (erweiterten) Euklidischen Algorithmus und für das Erzeugen von (Pseudo–) Zufallszahlen bereitstellen.

3.4.2 Die Schnittstelle

Die Schnittstelle ist die Klasse *bigint*. Sie unterstützt die Multi–Precision–Integers und ruft Funktionen des Kerns auf. Alle Operatoren, die in C++ für den Typ *long* existieren, werden in *bigint* so überladen, daß sie dasselbe Verhalten wie *long* Operatoren zeigen. Außerdem stellt *bigint* zu vielen dieser Operatoren eine prozedurale Version zu Verfügung.

3.4.3 Die Klassen

Es folgt nun eine kurze Charakterisierung der Klassen , die zur Zeit in LiDIA enthalten sind.

- Die Klasse **bigrational** unterstützt die Arithmetik der rationalen Zahlen.
- Die Klasse **bigmod** unterstützt die Arithmetik in $\mathbb{Z}/m\mathbb{Z}$ für eine positive ganze Zahl m.
- Die Klasse **bigfloat** unterstützt Multi–Precision–Floating–Point–Arithmetik, einschließlich der Berechnung elementarer Funktionen (exp, log, sin, cos, etc.).
- Die Klasse **bigcomplex** unterstützt die Arithmetik der komplexen Zahlen.

In allen Klassen sind die Standard-Arithmetik-Operationen und die Standard- Vergleichsoperatoren überladen.

Es gibt auch Klassen, in denen komplexere Algorithmen implementiert wurden. Dazu gehören folgende Klassen:

- Die Klasse **bigint_matrix** unterstützt die lineare Algebra über dem Ring der ganzen Zahlen. In dieser Klasse ist es möglich, den Kern, das Bild, die Determinante, das charakteristische Polynom, die Hermite–Normalform etc. einer ganzzahligen Matrix zu berechnen.

- Die Klassen **lattice_gensys** und **lattice_basis** berechnen Gitterbasen von Erzeugendensystemen und unterstützen die Anwendung von LLL–Reduktionen auf die Gitterbasen. Später soll weiterhin möglich sein, die kürzesten Gittervektoren ungleich Null, bzw. Minkowski– und Korkine–Zolotaref reduzierte Basen zu finden.

- Die Klasse **rational_factorization** unterstützt die Faktorisierung ganzer Zahlen. Insbesondere kann die Prim–Faktor–Zerlegung einer ganzen Zahl durch die Auswahl einer Strategie (ECS, QS u.a.) berechnet werden.

Nach dieser Betrachtung der LiDIA–Bibliothek wissen wir nun, wie die Klasse *bigint_matrix* in das Gesamtkonzept integriert ist. Die entsprechenden Seiten des LiDIA–Handbuches, die sich mit der Klasse *bigint_matrix* beschäftigen, findet man im Anhang dieser Arbeit.

Teil II

Datenstruktur und Basisfunktionen einer Matrixklasse

Allgemeines

Wie bereits im Titel *Datenstruktur und Basisfunktionen einer Matrixklasse* angedeutet wird, gliedert sich dieser Teil in zwei Kapitel.

Im anschließenden Kapitel **Interne Datenstruktur** beschreiben wir ausführlich, welche Datenrepräsentation wir für die Implementierung der Matrixklasse *bigint_matrix* verwendet haben. Wir begründen, weshalb gerade diese Darstellung für unsere Belange am besten geeignet ist und zeigen die Vorteile gegenüber anderen Datenstrukturen auf.

Im Kapitel **Basisfunktionen** gehen wir auf Funktionen ein, deren Funktionalität sich an der einer Klasse zugrundeliegenden *Datenstuktur* orientiert und die die Grundlage aller weiteren Funktionen bilden. Wir werden erkennen, daß *Datenstruktur* und *Basisfunktionen* eng zusammengehören und sich gegenseitig beeinflussen.

Wir unterteilen die Basisfunktionen gemäß ihrer Funktion in der Matrixklasse in folgende Gruppen:

- *Konstruktoren* und *Destruktoren* sind für die speichertechnische Realisierung der Erzeugung einer Variablen bzw. der Freigabe des in einem Konstruktor allokierten Speichers am Ende eines Gültigkeitsbereiches einer Variablen verantwortlich.

- *Strukturfunktionen* sind Funktionen, die mehrere Variablen des Typs *bigint_matrix* strukturell miteinander verbinden oder eine Variable dieses Typs strukturell in mehrere aufteilen können.

- Unter dem Sammelbegriff *Zugriffsfunktionen* verbergen sich alle Operatoren und Funktionen, die dem Nutzer dieser Matrixklasse den Zugriff auf die interne Datenstruktur einer Variablen vom Typ *bigint_matrix* gewähren und Manipulationen an der Struktur erlauben.

- *Tauschfunktionen* erlauben zum einen den Tausch zweier Spalten bzw. Zeilen innerhalb einer Matrix und zum anderen den Tausch zweier ganzer Matrizen.

Entsprechend dieser Gliederung unterteilt sich das Kapitel in vier Abschnitte.

Kapitel 4

Interne Datenstruktur

Die Auswahl der Datenstruktur ist, wie im Kapitel *Einleitung* bereits kurz erwähnt wurde, für die Laufzeit der auf dieser Struktur basierenden Algorithmen von außerordentlicher Bedeutung.

Für die Klasse **bigint_matrix** haben wir uns für folgende interne Datenstruktur entschieden:

- **bigint **value** Zeiger auf ein zweidimensionales Array über dem Typ *bigint*.
- **long rows** .... Integer–Variable zur Speicherung der Anzahl der Zeilen.
- **long columns** ... Integer–Variable zur Speicherung der Anzahl der Spalten.

Demzufolge ist ein Element des Typs *bigint_matrix* ein zweidimensionales Array (**Wertearray**) mit getrennt abgespeicherten Dimensionen. Die Einträge der Matrix werden hierbei *zeilenweise* abgelegt.

Aus folgenden Gründen haben wir uns für die oben beschriebene Datenstruktur zur Implementierung der Matrixklasse entschieden.

1. Die Klasse *bigint_matrix* soll gemäß den in der Einleitung angeführten Zielsetzungen den Umgang mit *dichtbesetzten Matrizen* unterstützen.
   Das bedeutet, daß wir in der Regel mit einem geringen Prozentsatz an Nulleinträgen innerhalb einer Matrix zu rechnen haben. Jeder Eintrag ist von Bedeutung und muß möglichst **schnell adressierbar** und **referenzierbar** sein. Somit scheidet eine Implementierung über verkettete Listen, bei der lediglich Elemente ungleich Null via Zeiger miteinander verbunden werden, von vornherein aus.
   Bei Zugriffsoperationen müßten wir die verkettete Liste bis zum gewünschten Element entlang laufen, was wesentlich langsamer als die Adressierung im zweidimensionalen Array ist.

Wegen des geringen Prozentsatzes an Nulleinträgen wäre der **Speicherbedarf** der Listenstruktur sogar größer als der der Arraystruktur, weil sowohl der eigentliche Wert als auch der Zeiger auf das nächste Element abgespeichert werden müssen.

2. Aus den gleichen Gründen scheidet die Implementierung einer Matrix als ein Vektor von Vektoren vom Typ *bigint* und die Implementierung mittels Multidarstellung (z.B. Arrayimplementierung und verkettete Liste) aus. Der hierbei entstehende Overhead an Operationen und Speicherbedarf hätte fatale Auswirkungen auf die Laufzeit der auf dieser Datenstruktur operierenden Algorithmen.

Somit eignet sich ein zweidimensionales Array für unsere Belange am besten. Ein weiterer Vorteil ist, daß ein zweidimensionales Array dem Bild, das sich der Anwender im allgemeinen von einer Matrix macht, am nächsten kommt.

Aufgrund dessen, daß wir das Wertearray einer Matrix dynamisch anlegen und verwalten wollen und müssen, ist zusätzliche eine Entscheidung darüber zu treffen, ob das Wertearray zeilen– oder spaltenweise im Speicher angelegt und mit Werten belegt wird. Wir müssen uns zwischen folgenden Ablageschemata entscheiden:

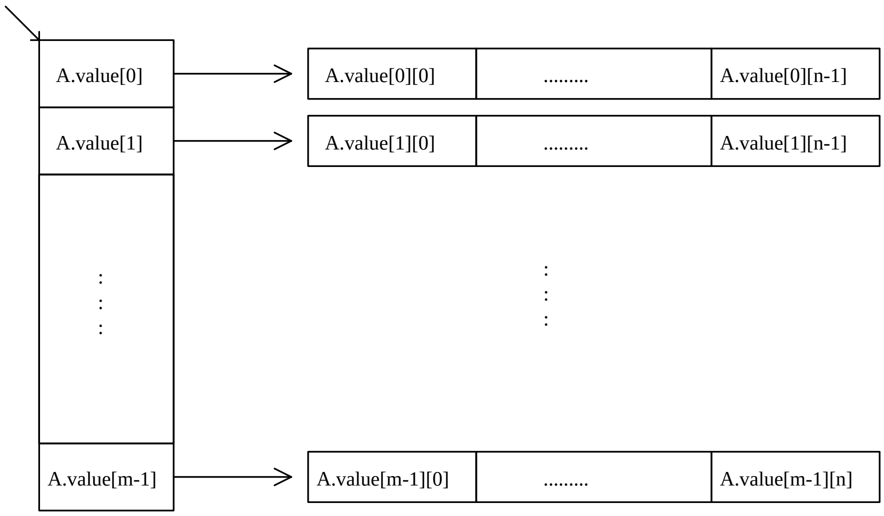

Abbildung 4.1: Ablageschema 1 – zeilenweise

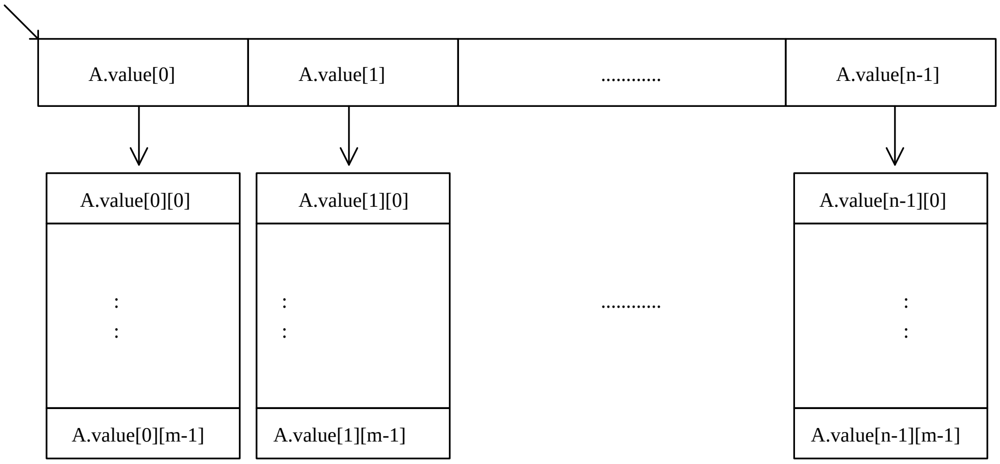

Abbildung 4.2: Ablageschema 2 – spaltenweise

Hierbei bezeichnen wir mit n die Anzahl der Spalten und mit m die Anzahl der Zeilen.

Auf dem Wertearray werden in der Regel nur Zugriffs- und Tauschoperationen ausgeführt. Dabei spielt das Ablageformat eine wesentliche Rolle. Demzufolge sollte uns die Betrachtung dieser Operationen einen Hinweis darauf liefern, welche der beiden Varianten für unsere Implementierung die bessere ist.

- Für Zugriffsoperationen auf dem Wertearray spielt der Unterschied zwischen den beiden Speicherungstypen keine Rolle. Dies erkennt man sofort, wenn man sich den Zeigerverlauf auf ein bestimmtes Element in beiden Varianten ansieht.
- Bei Tauschoperationen unterstützt die erste Variante den Tausch zweier Zeilen und die zweite den Tausch zweier Spalten. Die jeweilige Operation läßt sich auf der jeweiligen Ablagestruktur mittels drei Zeigeroperationen implementieren, was leicht aus obigen Skizzen erkennbar ist.

Falls man nicht nur Zeilenvertauschungen oder nur Spaltenvertauschungen verwenden will, liefern uns die Zugriffs– und Tauschoperationen also nicht den gewünschten Hinweis. Die Wahl zwischen diesen beiden Ablageformaten scheint in diesem Fall keinen offensichtlichen Einfluß auf die Laufzeit zu haben.

Um zu einer Entscheidung zu kommen, haben wir uns dem Wunsch, mit A.value[i][j] das Element der i-ten Zeile und j-ten Spalte zu bezeichnen, gebeugt und uns für die erste Variante entschieden. Diese Anlehnung an die Mathematik, in der mit $a_{i,j}$ der Eintrag in der i–ten Zeile und j–ten Spalte der Matrix A bezeichnet wird, erleichtert uns die Umsetzung mathematischer Konzepte in Algorithmen. Ebenso wird dadurch nach unserer Meinung die Lesbarkeit unserer Implementierung erhöht.

Kapitel 5

Basisfunktionen

Die Basisfunktionen der Matrixklasse *bigint_matrix* unterteilen sich gemäß ihrer Funktionalität in Konstruktoren, Destruktoren, Struktur– und Zugriffsfunktionen.

5.1 Konstruktoren und Destruktoren

Allgemein betrachtet bilden Konstruktoren und Destruktoren die Basis einer jeden C++ Klasse. Konstruktoren ermöglichen das (ordnungsgemäße) Erzeugen eines Objektes des jeweiligen Klassentyps. Destruktoren regeln die (ordnungsgemäße) Freigabe des durch die Konstruktoren allokierten Speichers. Sie werden dabei am Ende des jeweiligen Variablengültigkeitsbereiches automatisch aufgerufen.

Die Klasse *bigint_matrix* verfügt über folgende Konstruktoren und Destruktoren:

Prototypen:

- `void bigint_matrix()`
  konstruiert eine mit Null vorbelegte (1×1)–Matrix.
- `void bigint_matrix(long n)`
  konstruiert eine mit Nullen vorbelegte $(n \times 1)$–Matrix.
- `void bigint_matrix(long n, long m)`
  konstruiert eine mit Nullen vorbelegte $(n \times m)$–Matrix.
- `void bigint_matrix(long n, long m, const long **v)`
  konstruiert eine mit den Werten des zweidimensionalen Array v vorbelegte $(n \times m)$–Matrix.

- `void bigint_matrix(long n, long m, const bigint **v)`
  konstruiert eine mit den Werten des zweidimensionalen Array v vorbelegte $(n \times m)$–Matrix.

- `void bigint_matrix(const bigint_matrix &A)`
  konstruiert eine Kopie der Matrix A (copy - constructor).

- `void ~ bigint_matrix()`
  ist der Destruktor in dieser Matrixklasse.

Implementierung:

- Intern wird bei diesen Konstruktoren Speicherplatz in der notwendigen (oben in der Erklärung angeführten) Größe allokiert, die Anzahl der gewünschten Zeilen bzw. Spalten in den jeweiligen Membervariablen gespeichert und eine dem Konstruktor entsprechende Datenbelegung der Matrix durchgeführt.

  In einigen der oben aufgezählten Konstruktoren werden die Parameter n, m und v verwendet. Gilt beim Aufruf eines dieser Konstruktoren $n \leq 0$ oder $m \leq 0$, dann endet die Funktion mit der Fehlermeldung:

  constructor :: Parameter must be greater than zero !

  Falls v innerhalb eines Konstruktoraufrufs ein Nullzeiger ist, bricht die Funktion mit der Meldung

  constructor :: Error in parameter !

  ab.

- Im Destruktor werden mittels *delete[]* die im Konstruktor allokierten Speicherbereiche freigegeben.

WARNUNG !

Bei der Verwendung der Konstruktoren

`void bigint_matrix(long n, long m, const long **v)`

oder

`void bigint_matrix(long n, long m, const bigint **v)`

wobei v ein $(k \times l)$–Array sei, sollte der Anwender dieser Matrixklasse sicherstellen, daß gilt:

$$n \leq k \quad \wedge \quad m \leq l.$$

Sind diese Bedingungen nicht erfüllt, ist die Funktionsweise der Konstruktoren undefiniert !

Bemerkung:

Nach dem Aufruf des Konstruktors *bigint_matrix(m,n)* zeigt der Zeiger *value* auf ein Array mit m Elementen vom Typ *Zeiger auf bigint*. Diese Zeiger zeigen ihrerseits jeweils auf ein Array mit n Elementen vom Typ *bigint*. Die Nummerierung der einzelnen Elemente innerhalb der Arrays beginnt der C–Konvention entsprechend mit 0. Im weiteren Fortgang der Arbeit halten wir uns an diese Konvention, d.h. für uns existiert die 0–te Zeile und die 0–te Spalte einer Variablen vom Typ *bigint_matrix*, aber nicht die m–te Zeile und n–te Spalte.
In den Membervariablen *rows* und *columns* liegen nach dem obigen Konstruktoraufruf die mathematischen Dimensionen vor, d.h. $rows = m$ und $columns = n$.

5.2 Strukturfunktionen

Wie bereits zu Beginn dieses Teils erwähnt wurde, dienen Strukturfunktionen der Zusammenfassung der Daten mehrerer Variablen des Typs *bigint_matrix* zu einer Variablen gleichen Typs nach einem fest vordefinierten Schema. Oder umgekehrt dienen die Strukturfunktionen dazu, die Daten einer Variablen vom Typ *bigint_matrix* auf verschiedene Variablen gleichen Typs aufzuteilen. Dementsprechend unterscheiden wir Kombinations- und Teilungsfunktionen.

Sei im folgenden A eine Variable vom Typ *bigint_matrix*.

5.2.1 Kombinationsfunktionen

Prototypen:

- ```
  void
  A.compose( const bigint_matrix &B, const bigint_matrix &C,
             const bigint_matrix &D, const bigint_matrix &E )
  ```

  kombiniert die Daten der Matrizen B, C, D und E in der Matrix A, so daß:

  $$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}.$$
  ```

- `void A.compose_h(const bigint_matrix &B, const bigint_matrix &C)` kombiniert die Daten der Matrizen B und C in der Matrix A, so daß:

$$A = \begin{pmatrix} B & C \end{pmatrix}.$$

- `void A.compose_v(const bigint_matrix &B, const bigint_matrix &C)` kombiniert die Daten der Matrizen B und C in der Matrix A, so daß:

$$A = \begin{pmatrix} B \\ C \end{pmatrix}.$$

Implementierung:

- In der Funktion *A.compose(B,C,D,E)* wird zuerst getestet, ob die Dimensionen der Argumentmatrizen B, C, D und E kleiner gleich den Dimensionen der Matrix A sind. Ist dies nicht der Fall, endet die Funktion mit der Fehlermeldung:

  compose :: Error in matrix dimension !

  Ansonsten werden nacheinander die Einträge der Wertearrays der Matrizen B, C, D und E in dieser Reihenfolge an eine entsprechende Stelle innerhalb des Wertearrays der Matrix A kopiert. Dabei wird die genaue Position durch

$$\begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

  bestimmt. Das heißt, daß die Eckelemente der Matrizen B, C, D und E nach der Ausführung der Funktion gemäß folgender Tabelle in der Matrix A enthalten sind.

Eckpunkt	Position im Wertearray der Matrix A
$b_{0,0}$	$(0, 0)$
$c_{0,C.columns-1}$	$(0, A.columns - 1)$
$d_{D.rows-1,0}$	$(A.rows - 1, 0)$
$e_{E.rows-1,E.columns-1}$	$(A.rows - 1, A.columns - 1)$

Tabelle 5.2.1: Kombination von vier Matrizen in einer Matrix

Bei Überschneidungen der Komponentenmatrizen enthält das Wertearray der Matrix A das Element, das der obigen Reihenfolge gemäß zuletzt in das Wertearray geschrieben wird. Falls die Komponentenmatrizen die Matrix A nicht völlig überdecken, behalten die nicht überdeckten Einträge der Matrix A ihren ursprünglichen Wert.

- Die Funktionen *A.compose_h(B,C)* und *A.compose_v(B,C)* wurden in analoger Art und Weise zu der oben beschriebenen Funktion implementiert.

Beispiel:

Betrachten wir die Matrizen

$$B = \begin{pmatrix} 12 & 23 & 30 \\ 90 & 19 & 40 \end{pmatrix}, C = \begin{pmatrix} 24 & 89 \\ 67 & 73 \end{pmatrix}, D = \begin{pmatrix} 21 \\ 92 \\ 45 \end{pmatrix}, E = \begin{pmatrix} 12 & 88 & 73 \\ 99 & 33 & 91 \end{pmatrix}.$$

Desweiteren sei A eine (3×4)–Matrix.
Nach Ausführung der Funktion $A.compose(B, C, D, E)$ gilt:

$$A = \begin{pmatrix} 21 & 23 & 24 & 89 \\ 92 & 12 & 88 & 73 \\ 45 & 99 & 33 & 91 \end{pmatrix}.$$

Die Matrizen B, C, D und E bleiben unverändert.

5.2.2 Teilungsfunktionen

Prototypen:

- `void A.divide( bigint_matrix &B, bigint_matrix &C,`
  `bigint_matrix &D, bigint_matrix &E ) const`

  teilt die Matrix A in die Submatrizen B, C, D und E, so daß:

  $$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}.$$

- `void A.divide_h(bigint_matrix &B, bigint_matrix &C) const`
  teilt die Matrix A in die Submatrizen B und C, so daß

  $$A = \begin{pmatrix} B & C \end{pmatrix}.$$

- `A.divide_v(bigint_matrix &B, bigint_matrix &C) const`
  teilt die Matriz A in die Submatrizen B und C, so daß

$$A = \begin{pmatrix} B \\ C \end{pmatrix}.$$

Implementierung:

Wie bereits bei den Kombinationsfunktionen beschreiben wir lediglich die erste Teilungsfunktion. Die Implementierung der beiden folgenden verläuft analog.

Zunächst wird in der Funktion $A.divide(B, C, D, E)$ überprüft, ob die Dimensionen der Teilmatrizen B, C, D und E kleiner gleich den Dimensionen der Matrix A sind. Ist dies nicht der Fall, endet die Funktion mit der Fehlermeldung:

$$\boxed{\textbf{divide :: Error in matrix dimension !}}$$

Ist die Bedingung erfüllt, werden die Einträge der Matrix A gemäß des Schemas

$$\begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

in die Submatrizen B, C, D und E kopiert. Nach der Funktionsausführung gilt demnach:

$$\begin{aligned}
b_{0,0} &= a_{0,0} \\
c_{0,C.columns-1} &= a_{0,A.columns-1} \\
d_{D.rows-1,0} &= a_{A.rows-1,0} \\
e_{E.rows-1,E.columns-1} &= a_{A.rows-1,A.columns-1}
\end{aligned}$$

Bei Bereichsüberschneidungen der Submatrizen werden entsprechende Elemente der Matrix A mehrfach kopiert.

Beispiel:

Betrachten wir die Matrix

$$A = \begin{pmatrix} 12 & 23 & 62 & 89 \\ 90 & 19 & 80 & 73 \\ 45 & 90 & 32 & 91 \end{pmatrix}.$$

Desweiteren seien B eine (2×3)–Matrix, C eine (2×2)–Matrix, D eine (3×2)–Matrix und E eine (3×3)–Matrix.
Nach Ausführung der Funktion $A.divide(B, C, D, E)$ gilt:

$$A = \begin{pmatrix} 12 & 23 & 62 & 89 \\ 90 & 19 & 80 & 73 \\ 45 & 90 & 32 & 91 \end{pmatrix}$$

und

$$B = \begin{pmatrix} 12 & 23 & 62 \\ 90 & 19 & 80 \end{pmatrix}, C = \begin{pmatrix} 62 & 89 \\ 80 & 73 \end{pmatrix}, D = \begin{pmatrix} 12 & 23 \\ 90 & 19 \\ 45 & 90 \end{pmatrix}, E = \begin{pmatrix} 23 & 62 & 89 \\ 19 & 80 & 73 \\ 90 & 32 & 91 \end{pmatrix}.$$

5.3 Zugriffsfunktionen

Wie bereits im Einleitungsabschnitt dieses Teils erwähnt wurde, verbergen sich hinter dem Sammelbegriff *Zugriffsfunktionen* alle Funktionen und Operatoren, die dem Anwender Zugang zur internen Struktur der Matrixdarstellung einer Variablen vom Typ *bigint_matrix* gewähren und Manipulationen direkt an der Struktur erlauben. Die verwendete Datenstruktur besitzt gemäß der Betrachtungen im vorherigen Kapitel drei Komponenten. Wir sollten demnach im folgenden die Zugriffs– und Manipulationsfunktionen für jede dieser Komponenten betrachten.

5.3.1 Wertearray

Auf das Wertearray einer Variablen A vom Typ *bigint_matrix* kann man folgendermaßen zugreifen:

Prototypen:

- `bigint A.member(long i, long j) const`
  liefert den Wert der i–ten Zeile und j–ten Spalte der Matrix A zurück.

- `void A.sto(long i, long j, const bigint &e)`
  schreibt den Wert e an die Position (i, j) der Matrix A.

- `(bigint *) A.row(long i) const`
  gibt die Werte der i–ten Zeile der Matrix A als Array zurück.

- `void A.sto_row(const bigint *v, long l, long i)`
  schreibt l Werte des Arrays v von der Spalte Null beginnend in die i–te Zeile der Matrix A.

- `(bigint *) A.column(long i) const`
  liefert die Werte der i–ten Spalte der Matrix A als Array zurück.

- `void A.sto_column(const bigint *v, long l, long i)`
  schreibt l Werte des Arrays v von der Zeile Null beginnend in die i–te Spalte der Matrix A.

Darüber hinaus wurden die Operatoren () und [] wie folgt überladen:

- $A[i]$; ....überladen mit `A.row(i);`
- $A(i)$; ....überladen mit `A.column(i);`
- $A(i,j)$; ....überladen mit `A.member(i,j);`

Aufgrund dessen, daß die Klasse *bigint_matrix* als Basisklasse für weitere Klassen gedacht und konzipiert ist, müssen die Zugriffsfunktionen und –operatoren möglichst effizient sein. Deshalb liefern die Funktionen *A.row(i)* und *A.column(i)* bzw. die Operatoren *A[i]* und *A(i)* ein Zeiger auf *bigint* anstatt eines Vektors mit Elementen des Typs *bigint* zurück.

Implementierung:

- Bei dem Aufruf der Memberfunktion *A.member(i,j)* wird zunächst überprüft, ob die Parameter i und j innerhalb der zulässigen Wertebereiche liegen, d.h. es wird getestet, ob die Bedingungen $0 \leq i < A.rows$ und $0 \leq j < A.columns$ erfüllt sind. Trifft dies zu, wird der Wert $a_{i,j}$ zurückgegeben. Andernfalls bricht die Funktion mit der Fehlermeldung

  member :: Parameter out of range !

  ab.

- Die Implementierung der Funktion *A.sto(e,i,j)* verläuft analog zur Implementierung der Funktion *A.member(i,j)*. Anstatt aber das Element $a_{i,j}$ zurückzugeben, wird der Wert e an die Position (i,j) im Wertearray der Matrix A gespeichert. Als Fehlermeldung erscheint im Unterschied zu oben

  sto :: Parameter out of range !

- Die Memberfunktion *A.row(i)* überprüft zunächst die Bedingung $0 \leq i < A.rows$. Ist diese erfüllt, wird ein Array mit *A.columns* Elementen des Typs *bigint* allokiert, die Werte der i–ten Zeile des Wertearrays der Matrix A in dieses Array kopiert und schließlich ein Zeiger auf dieses Array zurückgegeben.
  Ist die Bedingung nicht erfüllt, bricht die Funktionsausführung mit der Meldung

row :: Parameter out of range !

ab.

- Die Funktion *A.sto_row(v,l,i)* überprüft zunächst, ob die Bedingungen $0 \leq i < A.rows$ und $0 \leq l < A.columns$ erfüllt sind. Gilt eine dieser Bedingungen nicht, bricht die Funktionsausführung mit der Fehlermeldung

  sto_row :: Parameter out of range !

  ab. Gelten diese Bedingungen, werden l Werte des eindimensionalen Arrays v von Position 0 beginnend in die Zeile i kopiert.

- Die Memberfunktionen *A.sto_column(v,l,i)* bzw. *A.column(i)* wurden analog zu den Funktionen *A.sto_row(v,l,i)* bzw. *A.row(i)* implementiert.

WARNUNG !

In den Funktionen

```
void A.sto_row(const bigint *v, long l, long i)
```

und

```
void A.sto_column(const bigint *v, long l, long i)
```

findet keine Überprüfung statt, ob und wieviele Werte das Array v enthält. Enthält das Array weniger als l Werte ist die Funktionsweise dieser Funktionen undefiniert !

Bemerkungen:

- Beachten Sie, daß die Funktionen *A.row(i)* und *A.column(i)* jeweils ein Array allokieren und einen Zeiger darauf zurückgeben. Bei Verwendung dieser Funktionen sollte nicht vergessen werden, diesen Speicherbereich durch *delete[]* wieder freizugeben.

- Desweiteren sollten wir bemerken, daß die Funktionalität des Operators $A(i,j)$ und die von $A[i][j]$ identisch sind. $A[i]$ gibt die i–te Zeile der Matrix A als Array zurück. Dessen j–te Komponente wird durch die zweite Klammerung $[j]$ ausgewählt.

5.3.2 Zeilenanzahl und Spaltenanzahl

Prototypen:

- `long A.get_no_of_rows() const`
  gibt die Anzahl der Zeilen der Matrix A zurück.
- `void A.set_no_of_rows(long r)`
  ändert die Anzahl der Zeilen der Matrix A in r.
- `long A.get_no_of_columns() const`
  gibt die Anzahl der Spalten der Matrix A zurück.
- `A.set_no_of_columns(long c)`
  änderte die Anzahl der Spalten der Matrix A in c.

Implementierung:

- Die Memberfunktionen *A.get_no_of_rows()* und *A.get_no_of_columns()* liefern lediglich den Wert der Membervariablen *A.rows* bzw. *A.columns* zurück.
- Bei den Funktionen *A.set_no_of_rows(r)* und *A.set_no_of_columns(c)* ist die Lage etwas komplizierter.
  Die intuitive Vorgehensweise sieht bei diesen Funktionen wie folgt aus:
  1. Wir allokieren zunächst ein neues Wertearray in der gewünschten Größe. Dies bedeutet, daß wir durch die Funktion *A.set_no_of_rows(r)* ein Array der Größe ($r \times A.columns$) und mittels der Funktion *A.set_no_of_columns(c)* ein ($A.rows \times c$)–Array allokieren.
  2. Dann kopieren wir die Einträge des alten Wertearrays an entsprechende Positionen des neuen Wertearrays. Dabei beachten wir, daß wir stets innerhalb der definierten Dimensionen beider Wertearrays bleiben. Sind nach diesem Kopiervorgang Positionen innerhalb des neuen Wertearrays noch unbelegt, werden diese mit Null aufgefüllt.
  3. Anschließend setzen wir den Zeiger vom alten Wertearray zum neuen Wertearray um und löschen den für das alte Wertearray reservierten Speicherbereich.

Bei dieser Schrittfolge fällt auf, daß viele Kopieraktionen benötigt werden, die bei großen Zahlen teuer sind. Deshalb versuchen wir zwecks Optimierung in unserer Implementierung einen möglichst großen Speicherbereich des alten Wertearrays in der veränderten Matrix wiederzuverwenden. Hier spielt die Speicherorganisation eine Rolle, was dazu führt, daß wir die Funktionen *set_no_of_rows* und *set_no_of_columns* getrennt betrachten müssen.

- Widmen wir uns zuerst der Funktion *A.set_no_of_columns(c)*. Übertragen auf die Datenstruktur beinhaltet die Ausführung dieser Funktion folgende Veränderung:

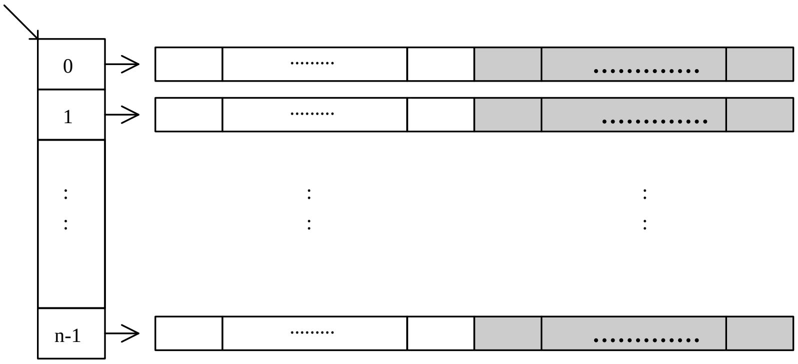

Abbildung 5.3.1: Speicherorganisation – set_no_of_columns

Zunächst überprüfen wir ob $c < 1$ ist. Ist dies der Fall, endet die Funktion mit Ausgabe der Fehlermeldung:

set_no_of_columns :: Error in parameter !

Ist dies nicht der Fall, müssen wir dafür sorgen, daß die Wertearrays, auf die die Zeiger $A.value[0], \ldots, A.value[A.rows - 1]$ zeigen, neu organisiert werden. Dazu unterscheiden wir folgende Möglichkeiten:

$c = columns$: Ist c gleich $A.columns$, dann ist nichts weiter zu tun.

$c < columns$: Ist c kleiner als $A.columns$, geben wir den Speicherbereich der Einträge $A.value[i][c], \ldots, A.value[i][A.columns - 1]$ für $0 \leq i < A.rows$ frei.

$c > columns$: Ist c größer als $A.columns$, reallokieren wir die Arrays, auf welche die Zeiger $A.value[0], \ldots, A.value[A.rows - 1]$ zeigen, auf die neue Größe und belegen die neu allokierten Eintragsbereiche mit Null.

Nach dieser Reorganisation setzen wir den Wert der Membervariablen $A.columns$ auf c und beenden die Funktion.

- Die Funktion *A.set_no_of_rows(r)* änderte die Anzahl der Zeilen. Auf der Ebene der Datenstruktur geschieht also folgendes:

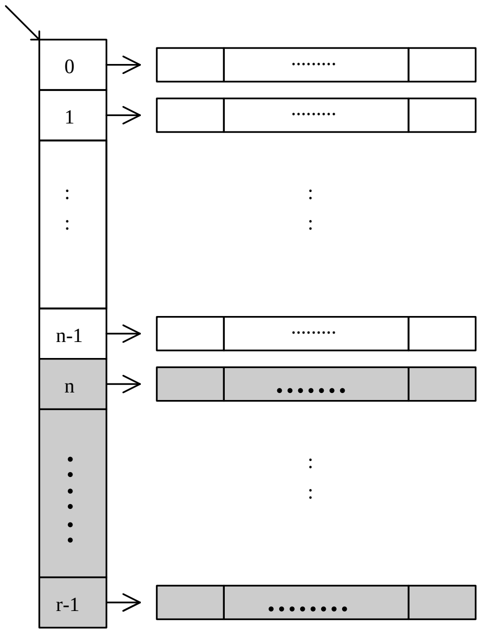

Abbildung 5.3.2: Speicherorganisation – set_no_of_rows

Bei der Ausführung der obigen Funktion überprüfen wir zunächst, ob $r < 1$ ist. Ist dies der Fall, beenden wir die Funktion mit der Fehlermeldung

set_no_of_rows :: Error in parameter !.

Ist dies nicht der Fall, reorganisieren wir das Array, auf das der Zeiger *A.value* zeigt. Hierzu gibts es folgende Möglichkeiten:

$r = rows$: Ist r gleich $A.rows$, beenden wir die Funktionsausführung.

$r < rows$: Ist r kleiner als $A.rows$, geben wir die Speicherbereiche der Arrays, auf welche die Zeiger $A.value[r], \ldots, A.value[A.rows - 1]$ zeigen, und den Speicherbereich, in dem diese Zeiger liegen, frei.

$r > rows$: Ist r größer als $A.rows$, reallokieren wir das Array auf die Größe r, auf das der Zeiger $A.value$ zeigt. Anschließend allokieren wir $(r - A.rows)$ Arrays der Größe $A.columns$ vom Typ *bigint*, belegen diese Arrays mit Nullen und setzen die neuen Zeiger *A.value[A.rows],. . . , A.value[r-1]* auf die Basisadressen dieser Arrays.

Nachdem diese Operationen ausgeführt wurden, wird der Wert der Variablen $A.rows$ auf r gesetzt und die Funktion beendet.

5.4 Tauschfunktionen

Die folgenden Tauschfunktionen bilden die letzte Gruppe, die wir zu den Basisfunktionen zählen.

Prototypen:

- `void A.swap_columns(long i, long j)`
  tauscht die Spalten i und j in der Matrix A.

- `void A.swap_rows(long i, long j)`
  tauscht die Zeilen i und j in der Matrix A.

- `void swap(bigint_matrix &A, bigint_matrix &B)`
  tauscht die Matrizen A und B.

Implementierung:

- Die Funktion *A.swap_columns(i,j)* überprüft zunächst, ob die Parameter i und j innerhalb des zulässigen Bereichs liegen, d.h. es wird überprüft, ob $0 \leq i, j < A.columns$. Ist dies nicht der Fall, endet die Funktion mit der Ausgabe der Fehlermeldung:

  swap_columns :: Error in parameters !

  Ist die Bedingung erfüllt, werden komponentenweise die Werte der i–ten Spalte und die der j–te Spalten miteinander getauscht.

- Bei der Ausführung der Funktion *A.swap_rows(i,j)* wird zunächst getestet, ob die Bedingung $0 \leq i, j A.rows$ erfüllt ist. Ist diese Bedingung nicht erfüllt, gibt die Funktion die Fehlermeldung

  swap_rows :: Error in parameters !

  aus und wird beendet. Ist die Bedingung erfüllt, werden die Werte der beiden Zeiger $A.value[i]$ und $A.value[j]$ miteinander getauscht.

- Die Funktion *swap(A,B)* tauscht zunächst die Werte von $A.columns$ und $B.columns$, dann die Werte von $A.rows$ und $B.rows$ und schließlich die Werte der Zeiger $A.value$ und $B.value$.

Aus der Darstellung der Datenstruktur im vorherigen Kapitel *Interne Datenstruktur* erkennt man, daß diese Funktionen das gewünschte Resultat liefern.

Teil III

Matrix – Arithmetik

Allgemeines

In den folgenden Kapiteln stellen wir die implementierten arithmetischen Funktionen und Operatoren unserer Matrixklasse *bigint_matrix* vor. Die Operatoren dienen hierbei lediglich als benutzerfreundliche Schnittstelle zu den jeweiligen Funktionen. Für zeitkritische Anwendungen sind deshalb die Funktionen den Operatoren jederzeit vorzuziehen.

Im folgenden Kapitel gehen wir zunächst ausführlich auf die Funktionen der Matrix–Arithmetik ein, wobei wir kurz die jeweilige Implementierung ansprechen und Laufzeitvergleiche mit entsprechenden Funktionen der Computer–Algebra–Systeme PARI, MAPLE und MATHEMATICA durchführen.

Die Laufzeitvergleiche mit diesen Systemen basieren auf folgender Vorgehensweise.

1. In jedem der betrachteten Systeme werden Argumentmatrizen mit vorbestimmten Dimensionen und zufälligen Einträgen im Bereich von -2^{31} bis $2^{31}-1$ generiert bzw. eingelesen.
2. Anschließend werden ausgewählte Funktionen, die nach unserer Meinung die Leistungsfähigkeit der Klasse *bigint_matrix* widerspiegeln, mit diesen Argumentmatrizen ausgeführt. Dabei messen wir die Laufzeiten dieser Funktionen innerhalb der verschiedenen Computer–Algebra–Systeme.

Der Leser wird im folgenden erkennen, daß viele der dargestellten Funktionen unter gewissen Umständen Dimensionsanpassungen mittels der Funktionen *set_no_of_rows* und *set_no_of_columns* ausführen. Um aussagekräftige und faire Testzeiten zu erhalten, haben wir alle Testläufe der Klasse *bigint_matrix* in zwei Varianten durchgeführt. In der ersten Variante wird die Dimensionsanpassung aus der Laufzeitberechnung ausgeklammert. In der zweiten Variante wird sie in der Laufzeit berücksichtigt. Desweiteren wurden alle Testläufe und Zeitmessungen mehrfach und auf Maschinen gleicher Bauart und gleicher Leistungsfähigkeit ausgeführt. In unserem Fall ist dies eine SUN Sparc 7 ELC mit 21,5 MIPS (Millions Instructions Per Second).

3. Die Ergebnisse, die wir bei Zeitmessungen obiger Art erhalten haben, werden jeweils am Ende eines jeden Abschnitts in Form einer Tabelle präsentiert. In der Spalte mit der Bezeichung LiDIA erscheinen in jeder Zeile zwei Laufzeiten.

Die erste Laufzeit erhalten wir ohne Berücksichtigung der Dimensionsanpassung. In der zweiten Laufzeit ist die interne Dimensionsanpassung enthalten. Das Symbol „–“ in den Tabellen bedeutet, daß die Berechnung innerhalb eines angemessenen Zeit– und Speicherverbrauchs nicht ausführbar waren.

Als Vergleichssysteme dienten uns:

- PARI Version 1.39
- MATHEMATICA Version 2.1
- MAPLE-V-R2

Kapitel 6

Funktionen der Matrix–Arithmetik

6.1 Addition und Subtraktion

Die Klasse *bigint_matrix* unterstützt zum einen die Addition und die Subtraktion zweier Matrizen, und zum anderen die komponentenweise Addition eines Skalars zu einer Matrix bzw. die komponentenweise Subtraktion eines Skalars von einer Matrix.

Prototypen:

- Addition:
  - ```
    void add( bigint_matrix &A, const bigint_matrix &B,
              const bigint_matrix &C )
    ```

    $$A = B + C$$

  - ```
    void add( bigint_matrix &A, const bigint_matrix &B,
              const bigint &e )
    ```

    $$\begin{pmatrix} a_{0,0} & \dots & a_{0,n} \\ \vdots & \ddots & \vdots \\ a_{m,0} & \dots & a_{m,n} \end{pmatrix} = \begin{pmatrix} b_{0,0}+e & \dots & b_{0,n}+e \\ \vdots & \ddots & \vdots \\ b_{m,0}+e & \dots & b_{m,n}+e \end{pmatrix}$$

    mit $m = B.rows - 1$ und $n = B.columns - 1$.

- Subtraktion:

  - `void subtract( bigint_matrix &A, const bigint_matrix &B, const bigint_matrix &C )`

$$A = B - C$$

  - `void subtract( bigint_matrix &A, const bigint_matrix &B, const bigint &e )`

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,n} \\ \vdots & \ddots & \vdots \\ a_{m,0} & \dots & a_{m,n} \end{pmatrix} = \begin{pmatrix} b_{0,0} - e & \dots & b_{0,n} - e \\ \vdots & \ddots & \vdots \\ b_{m,0} - e & \dots & b_{m,n} - e \end{pmatrix}$$

mit $m = B.rows - 1$ und $n = B.columns - 1$.

Implementierung:

Im folgenden werden wir nur die Implementierung der Addition beschreiben. Die Implementierung der Subtraktion ist dazu analog.

- Die Funktion *add(A,B,C)* realisiert die Matrixaddition.
  Zunächst wird überprüft, ob die Matrizen B und C die gleichen Dimensionen besitzen. Ist dies nicht der Fall, endet die Funktion mit der Ausgabe der Fehlermeldung:

  add :: Error in matrix dimensions !

  Andernfalls wird überprüft, ob die Matrix A die gleichen Dimensionen wie die Matrizen B und C hat. Ist dies nicht der Fall, werden die Dimensionen der Matrix A unter Benutzung der Funktionen *set_no_of_rows* und *set_no_of_columns* angepaßt. Anschließend werden die Wertearrays der Matrizen B und C mittels der Schulmethode addiert und das Ergebnis in das Wertearray der Matrix A geschrieben.

- Die Funktion *add(A,B,e)* überprüft zunächst, ob die Dimensionen der Matrizen A und B identisch sind. Ist diese Bedingung erfüllt, wird komponentenweise zu jedem Eintrag des Wertearrays der Matrix B das Skalar e addiert und das Ergebnis an die entsprechende Position innerhalb des Wertearrays der Matrix A geschrieben. Ist die Bedingung nicht erfüllt, werden zuerst die Dimensionen der Matrix A angepaßt und anschließend obige Addition ausgeführt.

Laufzeitvergleich:

Für diesen Komplex sehen wir die Laufzeiten der Matrixaddition als charakteristisch an. Die Laufzeiten für diese Funktion sind für die betrachteten Systeme in der folgenden Tabelle zusammengestellt:

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:07 00:00:15	00:00:13	00:02:57	00:25:33
200×200	00:00:32 00:00:56	00:00:52	00:10:32	02:36:58
300×300	00:00:65 00:01:19	00:01:21	00:23:48	–
400×400	00.01:17 00:02:22	00:02:35	00:42:03	–
500×500	00:01:81 00:03:26	00:03:30		–

Tabelle 6.1.1: Laufzeitvergleich – Matrixaddition

Den Versuch, mit MAPLE zwei Matrizen mit Dimensionen größer als (300×300) zu addieren, hat das System mit der Meldung *Sytem error – ran out of memory* abgebrochen. Der Speicherbedarf betrug mehr als 16 MB.

6.2 Multiplikation

Die Klasse *bigint_matrix* unterstützt die Matrixmultiplikation, die Skalarmultiplikation und die Matrix–Vektor–Multiplikation. Dabei bezeichnen wir ein eindimensionales Array als Vektor.

Prototypen:

- `void multiply( bigint_matrix &A, const bigint_matrix &B,`
  `const bigint_matrix &C )`

$$A = B \cdot C$$

- ```
  void multiply( bigint_matrix &A, const bigint_matrix &B,
                 const bigint &e )
  ```

$$A = e \cdot B$$

- ```
  void multiply( bigint *w, const bigint_matrix &A,
                 const bigint *v )
  ```

$$w = A \cdot v$$

(im Sinne einer Matrix–Vektor–Multiplikation)

Implementierung:

- Die Funktion *multiply(A,B,C)* führt eine Matrixmultiplikation durch.
  Zuerst wird getestet, ob die Dimensionen der Matrizen B und C für die Matrixmultiplikation verträglich sind, d.h. es wird überprüft, ob gilt:

  $$B.columns = C.rows.$$

  Ist diese Bedingungen nicht erfüllt, bricht diese Funktion mit der Fehlermeldung

  $$\boxed{\textbf{multiply :: Error in matrix dimensions !}}$$

  ab. Ist diese Bedingung erfüllt, wird überprüft, ob der Zeiger $B.value$ oder der Zeiger $C.value$ auf dasselbe Objekt wie der Zeiger $A.value$ zeigen. Wir unterscheiden folgende zwei Fälle:

  1. Der Zeiger $B.value$ und der Zeiger $C.value$ zeigen auf andere Objekte, als der Zeiger $A.value$.
     Nun wird überprüft, ob die Dimensionen der Ausgabematrix A den Anforderungen

     $$A.rows = B.rows$$
     $$A.columns = C.columns$$

     genügen. Tun sie dies nicht, werden die Dimensionen der Matrix A entsprechend angepaßt. Nun werden die Wertearrays der Matrizen B und C nach der Schulmethode im Sinne einer Matrixmultiplikation miteinander multipliziert und das Ergebnis in das Wertearray der Matrix A geschrieben.

2. Zeigt der Zeiger *B.value* oder der Zeiger *C.value* auf dasselbe Objekt wie der Zeiger *A.value*, erzeugen wir eine Hilfsvariable *TMP* vom Typ *bigint_matrix* mit *B.rows* Zeilen und *C.columns* Spalten. Damit verhindern wir, daß während der folgenden Durchführung der Multiplikation, noch benötigte Werte in der Matrix A, die dieselbe wie die Matrix B oder C ist, überschrieben werden. Anschließend multiplizieren wir die Wertearrays der Matrizen B und C nach der Schulmethode im Sinne einer Matrixmultiplikation miteinander und schreiben das Ergebnis in die Hilfsvariable *TMP*. Dann rufen wir die Funktion *swap(A,TMP)* auf. Nun beenden wir die Funktion.Die Variable *TMP* wird automatisch beim Verlassen der Funktion *add* durch den Destruktor gelöscht.

- *multiply(A,B,e)* multipliziert ein Skalar an eine Matrix heran.
  Die Funktion überprüft zunächst, ob die Dimensionen der Matrix A den Dimensionen der Matrix B entsprechen. Ist dies nicht der Fall, werden zuerst die Dimensionen der Matrix A den Dimensionen der Matrix B angepaßt. Ist dies nun der Fall, werden die Einträge des Wertearrays der Matrix B komponentenweise mit dem Skalar e multipliziert und das Produkt an die entsprechende Position in das Wertearray der Matrix A geschrieben.

- *multiply(w,A,v)* multipliziert die Matrix A mit dem Array v im Sinne einer Matrix–Vektor–Multiplikation.
  Zuerst wird getestet, ob Speicher für die Zeiger v und w allokiert wurde. Wurde für einen der beiden Zeiger kein Speicher allokiert, wird die Funktion mit der Fehlermeldung

  multiply :: No memory allocated for parameters !

  beendet. Ansonsten wird das Array v mit dem Wertearray der Matrix A multipliziert und das Ergebnis an die entsprechende Position des Arrays w gespeichert.

WARNUNG:
Es wird nicht überprüft, ob die Anzahl der Elemente des Arrays v gleich der Anzahl der Spalten, oder ob die Anzahl der Elemente des Arrays w gleich der Anzahl der Zeilen der Matrix A ist !

Laufzeitvergleich:

Im Bezug auf die Multiplikation betrachten wir die Laufzeiten der Matrixmultiplikation.

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:12:55 00:13:44	00:16:92	04:07:65	10:24:95
200×200	01:35:84 01:44:77	02:18:61	32:57:5	127:29:33
300×300	05:36:78 06:01:16	08:06:52	110:43:5	–
400×400	13:23:07 14:08:66	19:24:66	–	–
500×500	26:12:79 26:51:48	–	–	–

Tabelle 6.2.1: Laufzeitvergleich – Matrixmultiplikation

Die Laufzeiten lassen sich aber direkt auf nicht–quadratische Matrizen übertragen.

6.3 Negation

Prototyp:

- `void negate(bigint_matrix &A, const bigint_matrix &B)`

$$A = -B$$

Implementierung:

In der Implementierung der Funktion *negate(A,B)* wird zuerst überprüft, ob die Matrizen A und B die gleichen Dimensionen haben. Wenn dies der Fall ist, werden die komponentenweise negierten Einträge des Wertearrays der Matrix B an die jeweilige Position in das Wertearray der Matrix A geschrieben. Andernfalls werden zunächst die Dimensionen der Matrix A den Dimensionen der Matrix B angepaßt und dann obige Zuweisungen ausgeführt.

Laufzeitvergleich:

Das Computer–Algebra–System PARI stellt unseres Wissens nach keine Funktion dieser Funktionalität bereit.

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:15 00:00:22	–	00:02:32	00:26:68
200×200	00:00:61 00:00:93	–	00:09:36	03:03:78
300×300	00:01:75 00:02:00	–	00:21:65	–
400×400	00:02:41 00:03:32	–	00:38:62	–
500×500	00:03:80 00:05:12	–	–	–

Tabelle 6.3.1: Laufzeitvergleich – Negation einer Matrix

6.4 Restbildung

Eine weitere, nützliche Funktion, die in vielen der in Teil IV beschriebenen Funktionen verwendet wird, ist die folgende:

Prototyp:

- `void remainder( bigint_matrix &A, const bigint_matrix &B,`
  `const bigint &e )`

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{pmatrix} = \begin{pmatrix} b_{1,1} \bmod e & \dots & b_{1,n} \bmod e \\ \vdots & \ddots & \vdots \\ b_{m,1} \bmod e & \dots & b_{m,n} \bmod e \end{pmatrix}$$

Dabei berechnet $b_{i,j}$ *mod* e für $0 \leq i < m = B.rows$ und $0 \leq j < n = B.columns$ den absolut kleinsten Rest.

Implementierung:

Die Funktion *remainder(A,B,e)* überprüft zunächst, ob die Dimensionen der Matrix A den Dimensionen der Matrix B entsprechen. Ist diese Bedingung nicht erfüllt, werden die Dimensionen der Matrix A mittels der Funktionen *set_no_of_rows* und *set_no_of_columns* an die Dimensionen der Matrix B angepaßt. Anschließend werden alle Einträge des Wertearrays der Matrix B komponentenweise modulo e reduziert und das Resultat an die jeweilige Position in das Wertearrays der Matrix A geschrieben. Die Moduloreduktion berechnet dabei den absolut kleinsten Rest.

Laufzeitvergleich:

Die Reduktion einer Matrix modulo einer ganzen Zahl ist unseres Wissens mit einer PARI–eigenen Funktion nicht möglich.

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:47 00:00:55	–	00:01:19	01:03:27
200×200	00:02:07 00:02:02	–	00:04:75	08:30:92
300×300	00:05:38 00:04:78	–	00:10:90	–
400×400	00:07:51 00:07:82	–	00:19:37	–
500×500	00:11:84 00:12:28	–	–	–

Tabelle 6.4.1: Laufzeitvergleich – Reduktion einer Matrix

6.5 Zuweisung

Um nun die Grundarithmetik zu komplettieren stellen wir die Funktionen vor, die die Zuweisung in der Matrixklasse realisieren.

Prototypen:

- `void assign(bigint_matrix &A, const bigint_matrix &B)`
- `void A.assign(const bigint_matrix &B)`

$$A = B$$

Implementierung:

Die Funktionen *A.assign(B)* und *assign(A,B)* prüfen zunächst, ob die Dimensionen der Matrizen A und B identisch sind. Ist dies nicht der Fall, werden die Dimensionen der Matrix A mit Hilfe der Funktionen *set_no_of_rows* bzw. *set_no_of_columns* angepaßt. Wurde die Anpassung durchgeführt oder sind die Dimensionen von vornherein identisch, werden die Werte des Wertearrays der Matrix B komponentenweise an die entsprechende Position in das Wertearray der Matrix A kopiert.

Laufzeitvergleich:

PARI führt implizit Zuweisungen durch. Eine separate Funktion ist unseres Wissens nach nicht vorhanden.

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:05	–	00:00:06	00:19:98
200×200	00:00:16	–	00:00:16	02:35:26
300×300	00:00:36	–	00:00:40	–
400×400	00:00:63	–	00:00:76	–
500×500	00:01:02	–	–	–

Tabelle 6.5.1: Laufzeitvergleich – Zuweisung

6.6 Boolesche Funktionen

Die Funktionen der Matrix–Arithmetik haben wir durch folgende boolesche Funktionen erweitert. Jede dieser Funktionen liegt in zwei Varianten vor. Die zweite Variante stellt von der Seite der Implementierung her lediglich eine Benutzerschnittstelle dar. Wir haben sie der Klasse *bigint_matrix* beigefügt, um bekannte Schreibweisen aus anderen Systemen auch in LiDIA zu ermöglichen.

Prototypen:

- Die Funktionen
  - `int A.equal(const bigint_matrix &B) const`
  - `int equal(const bigint_matrix &A, const bigint_matrix &B)`

  geben 1 zurück, wenn die Dimensionen und die Werte der Matrizen A und B identisch sind. Ansonsten geben die Funktionen 0 zurück.

- Die Funktionen
  - `int A.unequal(const bigint_matrix & B) const`
  - `int unequal(const bigint_matrix &A, const bigint_matrix &B)`

  geben 0 zurück, wenn die Dimensionen und die Werte der Matrizen A und B identisch sind. Ansonsten geben die Funktionen 1 zurück.

- Die Funktionen
  - `int A.is_column_zero(int c) const`
  - `int is_column_zero(const bigint_matrix &A, int c)`

  geben 1 zurück, wenn die c–te Spalte des Wertearrays der Matrix A nur Nullen enthält. Ansonsten geben die Funktionen 0 zurück.

- Die Funktionen
  - `int A.is_row_zero(int r) const`
  - `int is_row_zero(const bigint_matrix &A, int r)`

  geben 1 zurück, wenn die r–te Zeile des Wertearrays der Matrix A nur Nullen enthält. Ansonsten geben die Funktionen 0 zurück.

Implementierung:

- Bei der Ausführung der Funktionen *A.equal(B)* oder *equal(A,B)* wird zunächst überprüft, ob die Matrizen A und B gleiche Dimensionen besitzen. Ist dies der Fall, werden die Einträge des Wertearrays der Matrix A mit den Einträgen an der korrespondierenden Stelle im Wertearray der Matrix B verglichen. Stellen wir dabei fest, daß alle Einträge identisch sind, wird 1 zurückgegeben. Finden wir einen Eintrag, der sich von dem im Wertearray der Matrix B unterscheidet, oder sind die Dimensionen unterschiedlich, liefern wir 0 zurück.

- Die Funktionen *A.unequal(B)* und *unequal(A,B)* arbeiten analog zu den beiden eben beschriebenen Funktionen. Der einzige Unterschied besteht darin, daß die beiden Rückgabewerte vertauscht wurden.

- *A.is_column_zero(c)* und *is_column_zero(A,c)* überprüfen zuerst, ob für den Parameter c
  $$0 \leq c < A.columns$$
  gilt. Ist diese Bedingung nicht erfüllt, werden die Funktionen mit der Fehlermeldung

  $$\boxed{\textbf{is\_column\_zero :: Parameter out of range !}}$$

  beendet. Ist die Bedingung erfüllt, überprüfen sie, ob alle Einträge der c–ten Spalte im Wertearray der Matrix A gleich 0 sind. Ist dies der Fall, wird 1 zurückgegeben, sonst wird 0 zurückgegeben.

- Die Funktionen *A.is_row_zero(r)* und *is_row_zero(A,r)* arbeiten analog auf den Zeilen des Wertearrays der Matrix A. Als Fehlermeldung wird

  $$\boxed{\textbf{is\_row\_zero :: Parameter out of range !}}$$

verwendet.

Laufzeitvergleich:

Aus der Liste der obigen booleschen Funktionen haben wir die Funktion *equal* ausgewählt. Diese Funktion besucht alle Elemente einer Matrix, wenn ihr identische Matrizen als Argumente übergeben werden. Deshalb messen wir die Laufzeit der Funktion *equal(A,B)*, wobei wir mit A und B zwei Variablen von Typ *bigint_matrix* mit gleichen Dimensionen und gleichen Werten im Wertearray bezeichnen.
In PARI konnten wir keine Funktion dieser Funktionalität finden.

Dimension	LiDIA	PARI	MATHEMATICA	Maple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:04	–	00:00:01	00:03:21
200×200	00:00:15	–	00:00:01	00:29:11
300×300	00:00:35	–	00:00:01	–
400×400	00:00:62	–	00:00:01	–
500×500	00:00:98	–	–	–

Tabelle 6.6.1: Laufzeitvergleich – Gleichheit

Kapitel 7

Operatoren der Matrix-Arithmetik

In diesem Kapitel stellen wir die Operatoren vor. Die Operatoren bilden lediglich eine Benutzerschnittstelle zu den im Kapitel *Funktionen der Matrix–Arithmetik* vorgestellten Funktionen.

In der folgenden Zusammenstellung wird jedem Operator die jeweilige Funktion, mit der er überladen ist, zugeordnet. Dabei sei *TMP* eine Hilsvariable vom Typ *bigint_matrix* und *tmp* ein Array "richtiger" Dimension vom Typ *bigint*, die in der jeweils betrachteten Operatorimplementierung deklariert wurden. Desweiteren seien B, C Variablen vom Typ *bigint_matrix* und e eine Variable vom Typ *bigint*.

1. Addition
   - $A+=B$; .... *add(A, A, B);*
   - $A+=e$; .... *add(A, A, e);*
   - $B+C$; .... *add(TMP, B, C); return TMP;*
   - $B+e$; .... *add(TMP, B, e); return TMP;*
2. Subtraktion
   - $A-=B$; .... *subtract(A, A, B);*
   - $A-=e$; .... *subtract(A, A, e);*
   - $B-C$; .... *subtract(TMP, B, C); return TMP;*
   - $B-e$; .... *subtract(TMP, B, e); return TMP;*
3. Multiplikation
   - $A*=B$; .... *multiply(A, A, B);*
   - $A*=e$; .... *multiply(A, A, e);*

- $B * C$; .... *multiply(TMP, B, C); return TMP;*
- $B * e$; .... *multiply(TMP, B, e); return TMP;*
- $A * v$; .... *multiply(tmp, A, v); return tmp;*

4. Restbildung

- $B\% = e$; .... *remainder(B, B, e);*
- $B\%e$; .... *remainder(TMP, B, e); return TMP;*

5. Zuweisung

- $A = B$; .... *A.assign(B);*

6. Boolesche Operatoren

- $A == B$; .... *A.equal(B);*
- $A! = B$; .... *A.unequal(B);*

Teil IV

Weitere Algorithmen der linearen Algebra

Kapitel 8

Das Problem entry explosion

Nachdem die Implementierung der bisher beschriebenen Operatoren und Funktionen keine nennenswerten Probleme aufgeworfen hat, und wir in der Regel Schulmethoden zur Berechnung verwenden konnten, könnte man der Ansicht verfallen, daß die Implementierung von komplizierteren Algorithmen der linearen Algebra über $\mathbb{Z}$ ebenfalls kein Problem darstellen dürfte. Doch dem ist nicht so.
Um zu verdeutlichen, welche Probleme bei etwas komplizierteren Algorithmen der linearen Algebra über $\mathbb{Z}$ auftreten, wollen wir an dieser Stelle ein kleines Beispiel anführen. Folgende (20×20)–Matrix mit Einträgen kleiner als 10 wollen wir mittels Gauß–Elimination in eine obere Dreiecksmatrix verwandeln:

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

Abbildung 8.1: (20×20) – Matrix

Während dieser Berechnung tritt ein interessantes Phänomen auf. Als maximales Zwischenergebnis erhalten wir eine Zahl der Größenordnung 10^{720}.

Obwohl die Einträge der Startmatrix von moderater Größe sind, wachsen die Zwischenergebnisse in unkontrollierter Art und Weise an. Dieses Phänomen wird in der Literatur als **entry explosion** bezeichnet und tritt in vielen Algorithmen der linearen Algebra über $\mathbb{Z}$ auf.

Betrachtet man folgende Tabelle, in der die maximalen Werte der Zwischenmatrizen eingetragen sind, die jeweils nach einer Zeilenelimination in obigem Algorithmus entstehen, erkennt man, daß bereits nach wenigen Eliminationsschritten der Wertebereich einer *long*–Variablen verlassen wird.

Eliminationsschritt	maximaler Wert
1	78
2	189
3	418
4	252404
5	7576858302412
6	508959989329979740
7	1143378883295124024912426
8	217066201996088304598824698546028236007162181528556484
9	$\approx 10^{63}$
10	$\approx 10^{73}$
11	$\approx 10^{83}$
12	$\approx 10^{93}$
13	$\approx 10^{106}$
14	$\approx 10^{324}$
15	$\approx 10^{338}$
16	$\approx 10^{690}$
17	$\approx 10^{706}$
18	$\approx 10^{720}$
19	$\approx 10^{354}$
20	$\approx 10^{354}$

Tabelle 8.1: Zwischenergebnisse – Gauß–Elimination

Um Probleme der linearen Algebra über $\mathbb{Z}$, wie Determinantenberechnung u.a. überhaupt algorithmisch lösen zu können, scheint der Gebrauch einer Multi–Precision–Arithmetik unumgänglich zu sein.

Desweiteren ist es einleuchtend, daß dieses Phänomen unangenehme Folgen für die Laufzeit solcher Algorithmen hat, in denen es auftritt. Bei der obigen Gauß–Elimination führen wir eine polynomielle Anzahl von Operationen auf Operanden „explodierender“ Länge aus. Bei unvorsichtiger Implementierung führt dies zu langen Laufzeiten.

Diese Vorbetrachtungen zeigen deutlich, daß *entry explosion* das Hauptproblem verschiedener Algorithmen über $\mathbb{Z}$ darstellt. Wenn es uns gelingt, sie in den Griff zu bekommen, steht einer Implementierung effizienter Algorithmen über $\mathbb{Z}$ nichts mehr im Wege.

Folglich müssen wir zunächst nach Methoden zur Lösung oder zur Einschränkung dises Problems suchen, diese den gewünschten Berechnungen anpassen und sie schließlich möglichst effizient implementieren.

Im weiteren Verlauf dieses Teils wollen wir drei Methoden zur Lösung des Problems der *entry explosion* vorstellen und jeweils Funktionen angeben, die mittels der dargestellten Methode in der Klasse *bigint_matrix* implementiert wurden. Dabei streben wir keine theoretisch detaillierte Betrachtung der jeweiligen Methoden an, sondern wollen lediglich den Mechanismus skizzieren.

Die Effizienz der implementierten Funktionen belegen wir an geeigneter Stelle jeweils durch tabellarische Laufzeitvergleiche mit anderen Computer–Algebra–Systemen, sofern dies möglich ist.

Kapitel 9

Modulare Arithmetik I

9.1 Der Chinesische Restsatz

Die Idee des ersten Ansatzes, den wir hier als *modulare Arithmetik I* bezeichnen und der in der Literatur häufig lediglich *modulare Arithmetik* genannt wird, beruht auf folgendem Satz:

9.1. Satz (Chinesischer Restsatz)
Es seien $m_1, \ldots, m_n \in \mathbb{N}$ paarweise teilerfremd und es seien $x_1, \ldots, x_n \in \mathbb{Z}$. Dann existiert ein $x \in \mathbb{Z}$ mit:

$$x \equiv x_i \bmod m_i \qquad \textit{für } 1 \leq i \leq n$$

Dieses x ist eindeutig bestimmt modulo $M = \prod_{i=1}^{n} m_i$.

Beweis:

Setze $\quad M_i = \frac{M}{m_i} \quad$ für $1 \leq i \leq n$.
Dann löse $a_i \cdot M_i \equiv 1 \bmod m_i \quad \forall i \in \{1, \ldots, n\}$.
Die Lösbarkeit folgt aus der Tatsache, daß $\mathrm{ggT}(M_i, m_i) = 1 \quad \forall i \in \{1, \ldots, n\}$.
Setzt man nun $e_i = a_i \cdot M_i \bmod M \quad \forall i \in \{1, \ldots, n\}$, dann gilt:

$$e_i \equiv \left\{ \begin{array}{ll} 0 \; mod \; m_j & , \; j \neq i \\ 1 \; mod \; m_i & \end{array} \right\}, \; \forall i \in \{1, \ldots, n\}$$

Dann ist $x = \sum_{l=1}^{n} e_i \cdot x_i \bmod M$ die gesuchte Lösung.

Die Eindeutigkeut beweist man wie folgt:
Nehmen wir an, es gäbe eine zweite Lösung x'. Dann gilt:

$$x' \equiv x_i \equiv mod\ m_i \quad \forall i \in \{1,\ldots,n\}.$$

Daraus folgt $x \equiv x'\ mod\ M$, weil die m_i paarweise teilerfremd sind. ■

Wählen wir die Paramter $m_1,\ldots,m_n$ aus der Menge aller Primzahlen ($\mathbb{P}$), bietet uns dieser Satz die Möglichkeit, die Berechnung eines *eindeutig* definierten Objektes Obj über $\mathbb{Z}$ auf die Berechnung kongruenter, eindeutig definierter Objekte Obj_i in mehreren endlichen Körpern $K_i = \mathbb{F}_{m_i},\ i = 1,\ldots,n$ zu reduzieren. Die Berechnung dieses Objektes bzw. die Lösung eines Problems in $\mathbb{Z}$ erfolgt dann nach folgendem Schema:

$$\begin{array}{ccccccc} & \nearrow & \text{Problem in } \mathbb{F}_{m_1} & \longrightarrow & \text{Lösung in } \mathbb{F}_{m_1} & \searrow & \\ \text{Problem in } \mathbb{Z} & & \vdots & \vdots & \vdots & & \text{Lösung in } \mathbb{Z} \\ & \searrow & \text{Problem in } \mathbb{F}_{m_n} & \longrightarrow & \text{Lösung in } \mathbb{F}_{m_n} & \nearrow & \end{array}$$

Abbildung 9.1.1: Schema der modularen Arithmetik I

Um diese Vorgehensweise noch zu verdeutlichen, betrachten wir folgendes Beispiel.

Beispiel:

Nehmen wir an, daß wir die Determinante der folgenden (2×2)–Matrix berechnen wollen.

$$A = \begin{pmatrix} 3 & 7 \\ -5 & 8 \end{pmatrix}$$

Desweiteren wissen wir aus [21] (Seite 99), daß die Determinante einer $(n \times n)$–Matrix B kleiner der Hadamard–Schranke $H = \max\{\prod_{j=0}^{n} L_2(b_{*,j}), \prod_{j=0}^{n} L_2(b_{j,*})\}$ dieser Matrix ist.
Die Matrix A besitzt eine Hadamard–Schranke $H < 100$. Somit wissen wir, daß die Determinante der Matrix A im symmetrischen Intervall $I = [-100; +100]$ liegt.

Die folgenden Berechnungsschritte orientieren sich an dem eben geführten Beweis des Chinesischen Restsatzes:

1. Zuerst bestimmen wir eine Folge von Primzahlen, deren Produkt größer als die Anzahl der Elemente im symetrischen Intervall I ist.

$$m_1 = 2,\ m_2 = 3,\ m_3 = 5,\ m_4 = 7 \qquad \text{mit} \prod_{i=1}^{4} m_i = 210 = M > 200$$

   Damit garantieren wir, daß in dem Intervall I die Lösung eindeutig wird, weil aus im Intervall I keine zwei Elemente den gleichen Wert modulo M besitzten.

2. Dann folgt:

$$M_1 = 105,\ M_2 = 70,\ M_3 = 42,\ M_4 = 30$$

3. Nun berechnen wir die Determinante der Matrix A modulo m_i innerhalb des Körpers $\mathbb{F}_{m_i}$ für $i = 1, \ldots, 4$.

$$\det(A \bmod m_1) = x_1 = 1 \bmod m_1 \tag{9.1}$$
$$\det(A \bmod m_2) = x_2 = 2 \bmod m_2 \tag{9.2}$$
$$\det(A \bmod m_3) = x_3 = 4 \bmod m_3 \tag{9.3}$$
$$\det(A \bmod m_4) = x_4 = 3 \bmod m_4 \tag{9.4}$$

4. Anschließend bestimmen wir $a_1,\ a_2,\ a_3,\ a_4$ mittels des erweiterten Euklidischen Algorithmus. Wir erhalten:

$$a_1 = 1,\ a_2 = 1,\ a_3 = 3,\ a_4 = 4$$

   und damit

$$e_1 = 105,\ e_2 = 70,\ e_3 = 126,\ e_4 = 120$$

5. Als Lösung in $\mathbb{Z}$ erhalten wir dann:

$$x = (\sum_{i=1}^{4} e_i \cdot x_i) \bmod M = (105 + 140 + 504 + 360) \bmod 210 = 59$$

   Man kann leicht durch nachrechnen überprüfen, daß wir das richtige Ergebnis erhalten haben.

Es stellen sich an diesem Punkt zwei Fragen:

- Kann man alle Berechnungen über $\mathbb{Z}$ auf die beschriebene Art und Weise auf Berechnungen über endlichen Körpern verlagern ?
- Wenn man eine Berechnung verlagern kann, wie erhält man das symmetrische Intervall I ?

Aus dem Chinesischen Restsatz erhalten wir implizit zwei Kriterien, die uns bei der Beantwortung dieser Fragestellungen helfen.

1. Um eine Berechnung auf endliche Körper zu verlagern, muß die Lösung der Berechnung über $\mathbb{Z}$ modulo m_i kongruent zu den Lösungen adäquater, eindeutig definierter Berechnungen über den endlichen Körpern $\mathbb{F}_{m_i}$ sein.
2. Desweiteren müssen wir für die Komponenten der Lösung über $\mathbb{Z}$ eine obere Schranke S angeben oder berechnen können, sodaß $I = [-S, S]$ das gesuchte Intervall bildet.

Achim Müller hat in seiner Arbeit „*Effiziente Algorithmen für Problem der linearen Algebra über* $\mathbb{Z}$" gezeigt, daß die in der folgenden Tabelle aufgeführten Berechnungen in Verbindung mit der jeweils angegebenen Schranke diese beiden Kriterien erfüllen. Sei nun $A \in \mathbb{Z}^{n \times n}$ und $B \in \mathbb{Z}^{n \times m}$.

Berechnung	Schranke
Determinatenberechnung einer Matrix A	Hadamard–Schranke der Matrix A
Berechnung der adjungierten Matrix zu A	Hadamard–Schranke der Matrix A
Berechnung des Rangs einer Matrix B	Hadamard–Schranke der Matrix B
Berechnung des Rangs und der Indizes der linear unabhänigen Zeilen einer Matrix B	Hadamard–Schranke der Matrix B
Berechnung des Rangs und der Indizes der linear unabhänigen Spalten einer Matrix B	Hadamard–Schranke der Matrix B
Berechnung der Koeffizienten des charakteristischen Polynoms einer Matrix A	$n \cdot \|A\|^n$

Tabelle 9.1.1: Zusammenstellung: Problem – Schranke

Außerdem hat er in seiner Arbeit Gründe angeführt, weshalb die Berechnung der Hermite–Normalform, die als Basisbaustein der Bild– und Kernberechnung fungiert, mittels dieses Ansatzes nicht durchgeführt werden kann.

Die Implementierung der Klasse *bigint_matrix* ist auf die Arbeit von Achim Müller aufgesetzt. Wir haben bisher keine Funktionen eingebaut, die den Chinesischen Restsatz verwenden und nicht in der erwähnten Arbeit bewiesenermaßen den beiden obigen Kriterien, mit der jeweils angegebenen oberen Schranke, genügen. Aus diesem Grund werden wir an dieser Stelle auf diese theoretischen Aspekte nicht näher eingehen.

Betrachtet man den Chinesischen Restsatz und seine Anwendung unter dem Gesichtspunkt einer geeigneten Modularisierung, erhalten wir eine Dreiteilung des Problemlösevorgangs, wie schon in [21] erläutert wurde.
Sei A ein Variable vom Typ *bigint_matrix*, die unser „Problemmodul" bzw. unsere „Problemabbildung" beschreibt.

Phase 1: Vorberechnung
Zunächst berechnen wir zwecks Abschätzung der Größe der Komponenten des erwarteten Resultats Obj eine obere Schranke S. Anschließend ermitteln wir eine Liste von Primzahlen $m_1, \ldots, m_n$, deren Produkt größer als das Doppelte der berechneten Schranke ist. Damit haben wir die Voraussetzungen geschaffen, um mittels des Chinesischen Restsatzes die richtige Lösung in $\mathbb{Z}$ zu finden.

Phase 2: Reduktion
Die in Phase 1 ermittelten Primzahlen dienen uns als Charakteristika m_i der Primkörpern $\mathbb{F}_{m_i}$, auf die unsere Problemstellung jetzt übertragen wird. Anschließend wird das Problem innerhalb der Primkörper der jeweiligen Charakteristik gelöst.

Phase 3: Rekonstruktion
Die Ergebnisse $Obj_1, \ldots, Obj_n$ der Berechnungen aus Phase 2 werden gesammelt und in Phase 3 mittels des oben angegebenen Chinesischen Restsatzes zu einer Lösung in $\mathbb{Z}$ zusammengesetzt.

Betrachten wir diesen Ansatz der *modularen Arithmetik I* unter dem Blickwinkel der *entry explosion*, so ist der Vorteil in Bezug auf die Größe der Zwischenergebnisse unübersehbar. In Phase 1 wird keine Veränderung an der Eingabematrix vorgenommen. Die Größe des maximalen Eintrages bleibt also unverändert. In Phase 2 ist die Größe der entstehenden Zwischenresultate durch

$$\max_{i=1,\ldots,n} \{Charakteristik\ von\ K_i\}$$

beschränkt. Lediglich in Phase 3 werden die in Phase 2 berechneten „kleinen“ Resultate zum Endergebnis zusammengesetzt. Das wiederum bereitet uns auch keine Probleme, weil keine weitere „explodierende“ Berechnung auf dem Ergebnis stattfindet.

Das Problem der *entry explosion* haben wir hiermit soweit unter Kontrolle. Doch wie kann man dieses Verfahren effizient gestalten ?

Betrachtet man die oben angeführte Beschreibung des Problemlösevorgangs unter dem Gesichtspunkt der Effizienzsteigerung, so fallen zwei Ansatzpunkte ins Auge.

1. Ändert man die Konstruktion der Primzahlliste in Phase 1 dahingehend ab, daß weniger Primzahlen verwendet werden, werden folglich auch weniger endliche Körper zur Berechnung benötigt. Daraus folgt, daß weniger Problemlösungen in $\mathbb{F}_{m_i}$ stattfinden, was zu einem Geschwindigkeitszuwachs führt. Führt man diese Überlegung weiter, könnte man mit nur einer Berechnung über $\mathbb{F}_p$ auskommen, vorausgesetzt die verwendet Primzahl ist groß genug, d.h. $p > 2 \cdot S$.

2. Betrachtet man den Zeitverbrauch innerhalb einer Berechnung nach obigem Schema, so fällt auf, daß ca. 80% - 90% der Gesamtlaufzeit im Mittelabschnitt (Phase 2) verbraucht wird. Der Grund dafür ist, daß die Algorithmen, die im Mittelabschnitt benutzt werden, in der Regel mehrfach zu Berechnungen aufgerufen werden. Dementsprechend wirken sich selbst kleine Verbesserungen innerhalb der Algorithmen, die die Objekte Obj_i im Körper $\mathbb{F}_{m_i}$ berechnen, sehr stark auf die Gesamtlaufzeit aus.

   Diese Erkenntnis hat uns zu folgender Überlegung veranlaßt, bei welcher wir natürlich voraussetzen, daß wir den besten, uns bekannten Algorithmus zur Lösung des betrachteten Problems verwenden.

   Wir betrachten die Laufzeiten eines Algorithmus, der die Determinante einer Matrix über $\mathbb{F}_{m_i}$ berechnet, unter dem Gesichtspunkt, daß die Körperoperationen einerseits über *bigint* und andererseits über *long* implementiert wurden. Anhand der folgenden Tabelle erkennt man, daß der gleiche Algorithmus mit Körperoperationen über *bigint* deutlich langsamer als die entsprechende *long*–Variante ist. Vorraussetzen müssen wir bei dieser Betrachtung allerdings, daß die bei diesen Berechnungen verwendete Charakteristik des Körpers klein genug ist, um eine korrekte Berechnung ohne Overflow im Integerbereich zu erlauben.

Dimension	Charakteristik	long–Variante	bigint–Varinate
10×10	32749	00:00:01	00:00:07
20×20	32749	00:00:06	00:00:50
30×30	32749	00:00:20	00:01:64
40×40	32749	00:00:47	00:03:90
50×50	32749	00:00:93	00:07:61
100×100	32749	00:07:38	01:00:31

   Tabelle 9.1.2: Laufzeitvergleich *long*–Vaiante – *bigint*–Variante

   Die Matrizen, die dieser Tabelle zugrundeliegen, wurden mit Einträgen betragsmäßig kleiner als $2^{31} - 1$ zufällig generiert. Ausgeführt wurden die Laufzeittests auf einer Sparc V7 ELC mit einer Integer–Bitlänge von 32. Bei den Berechnungen wurde als Charakteristik die größte Primzahl kleiner 2^{15} verwendet, um die Laufzeitunterschiede zwischen *bigint*– und *long*–Variante zu verdeutlichen.

   Demnach wäre die *long*–Variante gegenüber der *bigint*–Variante jederzeit zu bevorzugen. Oder anders gesagt, sollte man nur die *long*–Variante implementieren?

   Die Benutzung der *long*–Variante ist, wie oben bereits erwähnt wurde, abhängig von der Größe der Charakteristik des jeweiligen Körpers und somit von der Primzahlliste. Es kann nur für solche Körperberechnungen die *long*–Variante benutzt werden, für die der endlicher Körper eine Charakteristik mit einer Bitlänge kleiner als die Hälfte der Bitlänge eines maximalen

Integer–Eintrages besitzt. Die Halbierung der Bitlänge stellt sicher, daß die Multiplikation in $\mathbb{F}_{m_i}$ ohne Overflow durchgeführt werden kann.

Nach diesen Betrachtungen ergibt sich für uns kein klares Bild. Der erste Ansatz zur Effizienzsteigerung favorisiert die Benutzung der *bigint*–Variante, während die zweite der *long*–Variante den Vorzug geben würde.

Aufgrund dieser gespaltenen Situation haben wir für Funktionen, die sich der *modularen Arithmetik I* bedienen, Module zur Berechnung des Zielobjektes in endlichen Körper $\mathbb{F}_{m_i}$ sowohl in einer *long*–Variante als auch in einer *bigint*–Variante implementiert. Der jeweilige Algorithmus entscheidet dynamisch zur Laufzeit in Abhängigkeit von der Primzahlliste, welche Variante er zur Berechnung aufruft. Die Effizienz des Algorithmus ist durch diese Doppelvariante nur von der Wahl der Primzahlen abhängig. Effizienzveränderungen können wir somit als direkte Folge einer Veränderung der Primzahlliste betrachten.

Desweiteren haben wir implizit aus obiger Betrachtung zwei Zielsetzungen zur Konstruktion der Primzahllisten erhalten:

1. Minimiere die Anzahl der benutzten endlichen Körper!
2. Bevorzuge durch die Wahl der Primzahlen die *long*–Variante!

Doch welche der beiden Zielsetzungen sollte nun den Vorrang erhalten?

An diesen Stelle möchten wir diese Frage zurückstellen und verweisen auf eine Untersuchung dieses Problems im Abschnitt *Primzahlliste*.

9.2 Grundalgorithmen

In diesem Abschnitt gehen wir auf die Grundalgorithmen ein, die bei der *modularen Arithmetik I* benötigt werden. Dabei klammern wir die Berechnung der Primzahlliste aus und verweisen auf den nächsten Abschnitt in welchem wir detailliert darauf eingehen. Es bleiben also folgende Aufgaben übrig, die mittels geeigneter Funktionen erledigt werden müssen.

Aus Phase 1: Wir benötigen eine Funktion, die die obere Schranke S oder zumindest eine geeignete Approximation berechnet. Aufgrund dessen, daß die Berechnung der Schranke $n \cdot |A|^n$ völlig problemlos erfolgen kann, betrachten wir im weiteren lediglich die Berechnung der Hadamard–Schranke.

Aus Phase 2: Wir brauchen verschiedene Implementierungen des Chinesischen Restsatzes.

Der Plural bei Phase 2 ist hier mit Bedacht gewählt. In der Praxis hat es sich bewährt, die Funktion zur Ausführung des Chinesischen Restsatzes in mehreren Varianten zu implementieren. Doch darauf werden wir etwas später noch eingehen.

Für den Leser mag sich an diesem Punkt die Frage stellen, wozu er Informationen über diese Grundalgorithmen der *modularen Arithmetik I* benötigt. Wir verfolgen dabei die Zielsetzung, auf lange Sicht den Mechanismus der modularen Technik nach außen sichtbar und nutzbar zu machen. Damit wollen wir den Anwender dieser Klasse in die Lage versetzen, eigene Tests und Experimente mit diesem Schema zur Bekämpfung der *entry explosion* auszuführen.

9.2.1 Die Funktion `hadamard`

Laut der Beschreibung des Schemas der *modularen Arithmetik I* müssen wir in Phase 1 eine Schranke zur Abschätzung der Größe des Ergebnisses bzw. der Komponenten des Ergebnisses berechnen. Mit dieser Schranke haben wir ein Maß dafür, wie groß das Produkt der zu wählenden Primzahlen sein muß, damit die Anwendung des Chinesischen Restsatzes die korrekte Lösung in $\mathbb{Z}$ liefert.
Zu diesem Zweck wird des öfteren die Hadamard–Schranke einer Matrix als obere Schranke herangezogen. Aufgrund dessen, daß wir bei der Berechnung der Hadamard–Schranke, wie schon in [21] angeführt wurde, Wurzeln bestimmen müssen, berechnen wir hier lediglich eine Approximation dieser Schranke. Wir wollen auf die Theorie zu diesem Punkt nicht weiter eingehen und verweisen auf [21] (Seiten 99–104).

Die Klasse *bigint_matrix* stellt zur Berechnung einer Approximation der Hadamard–Schranke folgende Funktionen zur Verfügung.

Prototypen:

Sei A eine Variable vom Typ *bigint_matrix*.

- Die Funktion
  - `void A.hadamard(bigint &H)`

  schreibt die Approximation der Hadamard–Schranke der Matrix A in die Variable H.

- Die Funktionen
  - `bigint A.hadamard()`
  - `bigint hadamard(const bigint_matrix &A)`

  geben die Approximation der Hadamard–Schranke der Matrix A als Funktionswert zurück.

Implementierung:

- Die Funktion `A.hadamard(H)` ist eine Implementierung des Algorithmus in [21] (Seite 103). Die Verwendung eines Funktionsarguments zur Rückgabe des Resultats der Berechnung vermeidet eine Kopieraktion und ist somit schneller als die beiden folgenden Funktionen.

- Die Funktionen `bigint A.hadamard()` und `bigint hadamard(A)` rufen die erste Funktion auf und setzen die Parameter und Rückgabewerte in die gewünschte Form um. Sie sind einprägsamer und intuitiver als die erste Funktion und wurden aus diesem Grund implementiert.

9.2.2 Die Funktion `chinrest`

Der Chinesische Restsatz bildet die Grundlage des in diesem Kapitel betrachteten Ansatzes zur Lösung des Problems der *entry explosion*. Wie bereits häufiger erwähnt wurde, dient er der Zusammenfassung der Objekte $OBJ = (Obj_1, \ldots Obj_n)$ zu einer Lösung in $\mathbb{Z}$, die in Phase 2 als Ergebnisse von Berechnungen in den endlichen Körpern $\mathbb{F}_{m_1}, \ldots, \mathbb{F}_{m_n}$ entstanden sind.

Im folgenden sei *OBJ* der Vektor der in Phase 2 berechneten Objekte. *PRIM* sei die Liste der Primzahlen (Charakteristika) $m_1, \ldots, m_n$, die in Phase 2 zur Berechnung benutzt wurden. Weiterhin gelte, daß Obj_i im Körper $\mathbb{F}_{m_i}$ berechnet wurde.
Die Klasse *bigint_matrix* enhält folgende Implementierungen des Chinesichen Restsatzes:

Prototypen:

- Die Funktionen

  - `bigint chinrest(const bigint *OBJ, const bigint *PRIM)`
  - ```
    void chinrest( bigint &RES, const bigint *OBJ,
                   const bigint *PRIM )
    ```

  setzen die Objekte vom Typ *bigint* zu einer Lösung in $\mathbb{Z}$ vom Typ *bigint* zusammen und geben diese Lösung zurück bzw. speichern sie in der Variablen *RES* ab.

- Die Funktionen

  - ```
    void RES.chinrest( const bigint_matrix *OBJ,
                       const bigint *PRIM )
    ```

  - ```
    bigint_matrix chinrest( const bigint_matrix *OBJ,
                            const bigint *PRIM )
    ```
    ```

setzen die Elemente des Vektors *OBJ* vom Typ *bigint_matrix* zu einer Lösung in $\mathbb{Z}$ vom Typ *bigint_matrix* zusammen und speichern sie in der Variablen *RES* ab bzw. geben sie als Resultat zurück.

Implementierung:

- Die Funktionen der ersten Gruppe erwarten Objekte vom Typ *bigint* als Argumente. Resultat ist hierbei ebenfalls ein Element vom Typ *bigint*. Als Algorithmus haben wir den Algorithmus aus [21] (Seite 51) verwendet, der im wesentlichen die gleichen Berechnungen wie der Beweis zum Chinesischen Restsatz durchführt.
- Die Funktionen der zweiten Gruppe erhalten Objekte vom Typ *bigint_matrix* als Argumente. Theoretisch könnten wir nun komponentenweise den Chinesischen Restsatz anwenden, um eine Lösung in $\mathbb{Z}$ zu finden. Dies wäre allerdings aus dem Blickwinkel der Effizienz nicht ratsam. Deshalb haben wir hier eine Variante des Algorithmus aus [21] (Seite 53) benutzt, die alle Vorberechnungen, die lediglich auf der Primzahlliste basieren, nur einmal zu Beginn und nicht für jedes Element der Matrix ausführt.

9.3 Primzahlliste

Wie wir bereits in den vorangegangenen Abschnitten angesprochen haben, widmen wir uns nun der Aufgabe, eine geeignete Primzahlliste zu finden, die eine möglichst gute Effizienz bei Berechnungen nach dem Schema der *modularen Arithmetik I* verspricht.

Im Abschnitt *„Der Chinesische Restsatz“* haben wir uns bereits kurz mit der Primzahlliste beschäftigt. Dort haben wir die Forderungen aufgestellt, daß die Primzahlliste gemäß der folgenden Zielsetzungen konzipiert werden soll:

1. Minimiere die Anzahl der benutzten endlichen Körper!
2. Bevorzuge durch die Wahl der Primzahlen die *long*–Variante!

Zusammen mit der Vorgabe, Berechnungen auf dichtbesetzten Matrizen zu unterstützen, entstanden dazu eine Reihe von Versuchen und Überlegungen, die wir im weiteren Fortgang dieses Abschnitts beschreiben werden.

Zunächst möchten wir jedoch einige Vorbemerkungen machen.

1. Es ist leicht einzusehen, daß es sinnvoller ist, eine größere Liste von Primzahlen auf Platte zu hinterlegen, von der eine gewisse Anzahl von Primzahlen in eine

lokale Primzahlliste gelesen wird, als die lokale Primzahlliste nach den gleichen Vorgaben jeweils neu zu berechnen. Dabei hängt die Anzahl der einzulesenden Primzahlen vom jeweiligen Problem und von der Konstruktionsvorschrift für die lokale Primzahlliste ab. Aus diesem Grund setzen wir bei allen folgenden Tests voraus, daß die Primzahlen gelesen und nicht berechnet werden.

2. Desweiteren setzen wir voraus, daß jede der betrachteten Primzahllisten Primzahlen in absteigender Reihenfolge enthält. Damit tragen wir der Zielsetzung Rechnung, die Anzahl der verwendeten Primzahlen zu minimieren. Zwischen zwei aufeinanderfolgenden Einträgen innerhalb der verwendeten Liste existiert keine Primzahl. Man kann sich leicht überlegen, daß diese Vereinbarung sinnvoll ist und die Bemühung, möglichst wenige Primzahlen zu verwenden, unterstützt.

3. Die zufällig erzeugten Matrizen, die in den Testläufen der folgenden Abschnitte benutzt werden, besitzen Einträge betragsmäßig kleiner als $2^{31} - 1$.

4. Die Laufzeitmessungen und Untersuchungen führen wir exemplarisch am Beispiel der Determinantenberechnung durch. Eine Übertragung der Ergebnisse auf andere Problemstellungen der linearen Algebra, die den Anforderungen der *modularen Arithmetik I* genügen, ist aber jederzeit möglich.

9.3.1 long–Variante kontra bigint–Variante

Um das Zusammenspiel der *long*–Variante und der *bigint*–Variante innerhalb der Berechnung einer Determinanten näher zu untersuchen, haben wir folgenden Testlauf durchgeführt:

Zunächst wird zufällig eine Matrix der Dimensionen 20×20 erzeugt. Dann werden nacheinander Determinantenberechnungen durchgeführt, wobei die verwendete Primzahlliste variiert wird. Gemessen wird dabei

1. der Bereich der verwendeten Primzahlen,
2. die Gesamtlaufzeit des Algorithmus,
3. die Anzahl der verwendeten Primzahlen,
4. die Anzahl der beteiligten Ausführungen der *long*–Variante und
5. die Anzahl der beteiligten Ausführungen der *bigint*–Variante.

Damit die *long*–Variante korrekte Ergebnisse liefert, wird bei einer Charakteristik mit einer Bitlänge größer als der Hälfte der Bitlänge eines maximal möglichen *long*–Eintrages die *bigint*–Variante gewählt. Damit gehen wir sicher, daß bei der Multiplikation innerhalb dieses Körpers kein Overflow stattfindet. Die Maschine, auf der wir die der folgenden Tabelle zugrundeliegenden Testläufe ausgeführt haben, arbeitet mit einer 32–Bit–Integer-Darstellung. Der Wechsel zwischen den beiden Varianten findet somit bei einer Bitlänge von 16 Bit statt. Der maximal mögliche Integer–Eintrag mit 15–Bit–Länge ist 32767. Betonen möchten wir noch, daß die Meßergebnisse Durchschnittswerte aus 25 Messungen sind.

Bereich	Laufzeit (sec:hsec)	Gesamtanzahl	long–Anzahl	bigint–Anzahl
0 bis 33700	11:41	41	0	41
0 bis 33200	11:10	41	0	41
0 bis 33100	10:51	41	5	36
0 bis 33091	9:92	41	6	35
0 bis 33071	9:24	42	10	32
0 bis 33029	8:81	42	13	29
0 bis 32993	7:64	42	17	25
0 bis 32939	6:26	43	25	18
0 bis 32917	5:54	43	27	16
0 bis 32869	4:43	43	31	12
0 bis 32801	3:32	43	37	6
0 bis 32779	2:14	44	42	2
0 bis 32717	1:68	44	44	0

Tabelle 9.3.1: Laufzeitänderung bei veränderter Primzahlliste

Man erkennt deutlich, daß die Gesamtlaufzeit mit steigender Anzahl von Ausführungen der *long*–Variante beträchtlich kleiner wird.

Doch was geschieht, wenn die Anzahl der Aufrufe der *long*–Variante immer weiter steigt? Ist es in diesem Fall nicht besser, auf viele Aufrufe der *long*–Varianten zugunsten eines Aufrufs der *bigint*–Variante zu verzichten?

Aufgrund dieser Fragestellung haben wir folgende Überlegungen angestellt:
Die durchschnittliche Laufzeit der *bigint*–Varianten in obiger Konstellation, unter Vernachlässigung der Zeit für das Einlesen der Primzahlliste und die Ausführung des Chinesischen Restsatzes, beträgt ca. 27 hsec. Die durchschnittliche Laufzeit der *long*–Varianten beträgt 3.8 hsec. Somit wäre die Ausführung der *bigint*–Variante dann zu empfehlen, wenn wir dadurch mehr als 7 Ausführungen der *long*–Varianten einsparen würden. Dabei müssen wir sicherstellen, daß das Produkt der zu den Ausführungen der *long*–Variante korrespondierenden Einträge der Primzahlliste kleiner als die zu der *bigint*–Variante gehörende Primzahl ist.

Betrachten wir uns dies einmal genauer. Die erste Primzahl mit einer Bitlänge von 16 Bit, die die *bigint*–Variante auslöst, ist 32771. Das Produkt der 7 Primzahlen 2, 3, 5, 7, 11, 13 und 17 ist aber schon größer als diese Primzahl. Somit ist die Verwendung der Primzahl 32771 unter der Prämisse der Einsparung von Ausführungen der *long*–Varianten nicht zu empfehlen. Unter der unrealistischen Annahme, daß die *bigint*–Variante immer die gleiche Ausführungszeit besitzt, würde sich also die *bigint*–Variante erst bei einer Primzahl größer als 510510 ($= 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17$) bezahlt machen.

Diese Annahme ist deshalb unrealistisch, weil die Laufzeit von Operationen des Typs *bigint* im Gegensatz zu Operationen auf dem Typ *long* abhängig von der Anzahl der Bits der Operanden ist. Somit ist die Laufzeit der *bigint*–Varianten für eine Primzahl größer als 510510 größer als die für die Primzahl 32771. Dadurch veschiebt sich das obige Verhältnis 7:1 zwischen *long*–Variante und *bigint*–Variante zugunsten der *long*–Seite. Dies bedeutet, daß wir mehr Ausführungen der *long*–Varianten einsparen müssen, um Laufzeitgewinne zu verbuchen.

Diese Betrachtung belegt, daß die Verwendung der *long*–Varianten im Vergleich zur *bigint*–Varianten unter den oben angegebenen Rahmenbedingungen immer zu empfehlen ist.

Das Produkt aller Primzahlen mit einer Bitlänge kleiner gleich 15 ist eine Zahl mit 48484 Bit. Damit ist erst bei extrem großen Dimensionen damit zu rechnen, aufgrund eines Mangel an „long–Primzahlen“auf eine *bigint*–Variante zurückgreifen zu müssen.

Bemerkung:

Wir haben hier am Beispiel der Determinantenberechnung gezeigt, daß eine Primzahlliste, die Primzahlen kleiner als 2^{15} enthält und absteigend sortiert ist, eine sehr gute Laufzeit garantiert. Dieses Resultat erhält man auch, wenn man anstatt der Determinantenberechnung andere Berechnungen, die sich der *modularen Arithmetik I* bedienen, betrachtet.

Es stellt sich an dieser Stelle die Frage, ob wir das Verhältnis zwischen *long*– und *bigint*–Variante, das wir oben ermittelt haben, auf die eine oder andere Art und Weise beeinflussen können. Darauf gehen wir im nächsten Abschnitt ein.

9.3.2 Einflußfaktoren

Im vorherigen Abschnitt haben wir uns mit dem Verhältnis *long*–Variante zu *bigint*–Variante beschäftigt. Dieses Verhältnis ist jedoch keine statische Größe. Wir können durch Änderungen an den Körperoperationen dieses Verhältnis in die eine oder andere Richtung verschieben. In diesem Abschnitt stellen wir zwei Modifikationen vor, die beide in der Klasse *bigint_matrix* eingebaut wurden und zusammen zu erheblichen Geschwindigkeitszuwächsen geführt haben.

- Die Laufzeit von Operationen auf dem Typ *bigint* ist abhängig von der Anzahl der Bits der Operanden. Im Gegensatz dazu können wir bei der *long*–Variante davon ausgehen, daß die Laufzeit für den Bitbereich einer *long*–Variablen konstant ist.

Betrachten wir unter diesem Blickwinkel die *bigint*– und die *long*–Variante der Körperoperationen eines endlichen Körpers.

Alle Körperoperationen, sowohl in der *bigint*– als auch in der *long*–Variante basieren auf folgendem Schema:

1. Führe die Operation auf *bigint* bzw. *long* durch und
2. reduziere das Ergebnis *RES* modulo der Charakteristik des betrachteten endlichen Körpers.

Ansatzpunkt für die erste Modifikation ist die Reduktion modulo der Charakteristik. Wir wählen als Rest anstatt des kleinsten positiven Restes, den wir in den ersten Implementierungen verwendet hatten und der in der Literatur zumeist in der Beschreibung von Algorithmen verwendet wird, den betragsmäßig kleinsten Rest. Diese Variante, die auch in [14] und [12] im Zusammenhang mit dem Begriff *best remainder* erwähnt wird, verspricht kleinere Bitlängen der Operanden in der *bigint*–Variante und somit Geschwindigkeitsgewinne. Die *long*–Variante bleibt davon unbeeinflußt, weil es keine Rolle spielt, ob kleine oder große *long*–Werte addiert, subtrahiert oder multipliziert werden, solange kein Overflow stattfindet.

Diese Modifikation beeinflußt also das Verhältnis zugunsten der *bigint*–Variante.

Bei Messungen hat sich herausgestellt, daß sich diese Änderung nicht wesentlich, aber dennoch bemerkbar auf das Verhältnis zwischen *long*–Variante und *bigint*–Variante auswirkt.

- Die bisherigen Ergebnisse deuten darauf hin, daß viele Aufrufe der *long*–Varianten mit möglichst großen Charakteristika hohe Geschwindigkeit versprechen. Demzufolge haben wir uns damit beschäftigt, wie wir die bisherige obere Schranke für die Verwendung der *long*–Variante nach oben verschieben können.

  Die bisherige obere Schranke resultiert aus der Notwendigkeit, Overflow bei Berechnungen in endlichen Körpern hoher Charakteristik zu vermeiden. Deshalb müssen wir nach einer anderen Möglichkeiten suchen, Overflow zu vermeiden. Dadurch könnten wir die Schranke nach oben verschieben, d.h. den Wechsel von der *long*– zur *bigint*–Variante bei einer größeren Bitlänge durchführen und Laufzeitgewinne verbuchen.

  Eine naheliegende Idee dabei ist es, auf *double*–Hilfsvariablen auszuweichen. Diese besitzen zumindest auf unseren Maschinen doppelte *long*–Bitlänge und garantieren somit eine exakte Multiplikation zweier *long*–Werte größerer Bitlänge als bisher. Eine Verschiebung der bisherigen oberen Schranke auf die Bitlänge eines maximalen *long*–Eintrages ist im allgemeinen nicht möglich, weil in *double*–Variablen eine gewisse Anzahl an Bits zur Vorzeichenbehandlung und für den Exponenten reserviert werden. Die Zeit, die wir für den Wechsel der Werte in eine *double*–Variable und zurück benötigen, ist für die

Laufzeit ohne Bedeutung und kann deshalb vernachlässigt werden. Die Zeit, die wir für die Berechnung mit *double*–Variablen aufwenden müssen, wird durch den Laufzeitgewinn der Gesamtberechung um ein Vielfaches kompensiert.

Diese Idee beeinflußt das Verhältnis zugunsten der *long*–Varianten und wirkt sich bedeutend stärker auf die Laufzeit aus, als die erste Idee. Deshalb ist nach diesen Modifikationen an den Körperopertionen für endliche Körper die zu verwendende Primzahlliste immer noch so zu konzipierten, daß *long*–Varianten mit größtmöglicher Charakteristik bevorzugt zu Berechnungen herangezogen werden.

9.3.3 Zusammenfassung

Fassen wir alle bisherigen Untersuchungen zur Konstruktion einer Primzahlliste zusammen, so verspricht uns folgende Vorschrift gute Laufzeiten:

- Verwende solche Primzahlen bei Berechnungen nach dem Schema der *modularen Arithmetik I*, die eine Bitlänge kleiner gleich der Hälfte der Bitlänge eines maximalen ganzzahligen positiven Eintrages in einer *double*–Variablen haben.
- Verwende in der Liste diese Primzahlen in absteigender Reihenfolge und ohne Sprünge.

Im folgenden verwenden wir eine Primzahlliste, die nach dieser Vorschrift konstruiert und hinterlegt wurde. Von dieser Liste lesen wir mittels einer Funktion *get_primes* jeweils soviele Primzahlen in eine lokale Primzahlliste ein, wie wir zur Lösung des jeweiligen Problems benötigen. Man erkennt leicht, daß die lokale Primzahlliste dann ebenfalls der obigen Vorschrift genügt.

Bemerkung:

Dem Anwender der Klasse *bigint_matrix* würden wir empfehlen, sich eine eigene Primzahlliste nach den obigen Kriterien zu erzeugen, angepaßt an die Architektur, auf der diese Klasse eingesetzt wird. Die Laufzeitgewinne könnten enorm sein (z.B. bei 64–Bit Architekturen).

9.4 Funktionen der linearen Algebra

Nachdem wir die theoretischen Grundlagen gestreift haben, die Grundalgorithmen vorgestellt haben, und wissen, welche Primzahlliste in diesem Kontext geeignet ist, werden wir nun die Funktionen vorstellen, die wir mittels der *modularen Arithmetik I* in der Klasse *bigint_matrix* implementiert haben.

Der Beschreibung der Funktionen liegt nun folgendes Schema zugrunde:

- *Zielsetzung in der linearen Algebra*
  Unter diesem Punkt legen wir die Problemstellungen dar, die man mittels der gerade betrachteten Funktion(en) lösen kann.

- *Prototypen*
  An dieser Stelle werden die Prototypen der gerade betrachteten Funktionen angegeben. Dabei beziehen wir unsere Darstellung auf eine Variable A vom Typ *bigint_matrix*. d.h.:
  anstatt `void bigint_matrix::det(bigint &)` zu schreiben,
  schreiben wir `void A.det(bigint &)`.

- *Implementierung*
  Wie bereits erläutert wurde, bedienen sich alle in diesem Kapitel noch folgenden Funktionen der Methode der *modularen Arithmetik I*. Dementsprechend sind die Grundbausteine (Berechnung der oberen Schranke, Anwendung des Chinesischen Restsatzes) dieser Funktionen durch die Methode vorgegeben und für fast alle identisch. Demnach würde es in den folgenden Beschreibungen genügen, den Algorithmus anzugeben, der für die Berechnungen innerhalb der endlichen Körper $\mathbb{F}_{m_i}$ verwendet wird. Wir werden aber im Hinblick auf eine eventuelle Nutzung dieser Arbeit als Nachschlagewerk für die Klasse *bigint_matrix* im LiDIA–Paket jeweils kurz die Vorgehensweise skizzieren. Desweiteren erläutern wir das Auftauchen möglicher Fehlermeldungen und beschreiben kurz Besonderheiten der Implementierung, die für den Anwender von Bedeutung sind.

- *Laufzeit*
  In diesem Abschnitt führen wir tabellarisch einen Laufzeitvergleich der jeweiligen Funktion in verschiedenen Systemen an. Die dabei verwendeten Matrizen wurden mit Einträgen betragsmäßig kleiner als $2^{31} - 1$ zufällig generiert. Ausgeführt wurden die Laufzeitmessungen auf einer Sparc–V7–ELC–Maschine mit 21.5 MIPS. Als Primzahlliste wird eine Liste mit Primzahlen keiner als 2^{26} in absteigender Reihenfolge verwendet. Diese Liste trägt innerhalb des Computer–Algebra–Systems LiDIA den Namen *LIDIA_PRIMES* und wird in Zukunft als Standardprimzahlliste verwendet. Als Vergleichssysteme betrachten wir dieselben Systeme wie in Teil III. Das Symbol „–“ innerhalb der Laufzeittabellen bedeutet, das entweder die Berechnung einen Speicherbedarf von mehr als 15 MB hatte, oder die Berechnung unverhältnismäßig lange dauerte.

Bemerkung:

Zur Lösung der im folgenden angeführten Probleme der linearen Algebra wurden jeweils mehrere Funktionen gleichen Namens implementiert. Eine dieser Funktionen ist laufzeitoptimiert, d.h. beim Aufruf dieser Funktion werden unnötige Kopieraktionen vermieden. Die übrigen Funktionen sind Schnittstellen zu der laufzeitoptimierten Funktion. Sie setzen lediglich die Parameter und den Rückgabewert in die notwendige Form um und rufen dann die laufzeitoptimierte Funktion auf.

Beispiel:

- `void A.det(DET)`
  berechnet die Determinante der Matrix A. Dabei wird die Varibale *DET* intern schon zur Berechnung der Determinanten verwendet und dient nicht nur als Container für die Rückgabe.

- `bigint det(A)`
  ruft die Funktion `A.det(TMP)` auf und gibt *TMP* als Funktionswert zurück.

9.4.1 Rang

Zielsetzung der linearen Algebra:

- Berechne den Rang einer linearen Abbildung ϕ mit Abbildungsmatrix A, d.h. berechne die Anzahl der linear unabhängigen Spalten der Matrix A, die das Bild der linearen Abbildung ϕ erzeugen!

- Berechne den Rang eines Moduls, der von den Spalten der Matrix A erzeugt wird, d.h. berechne die Anzahl der linear unabhängigen Spalten der Matrix A, die den Modul erzeugen!

Prototypen:

Die Funktionen

- `long A.rank() const`

- `long rank(const bigint_matrix &A)`

liefern den Rang der Matrix A zurück.

Implementierung:

- *Vorgehensweise:*
  Die Berechnung des Rangs der Matrix A mittels der Funktion *A.rank()* verläuft wie folgt:

  1. Zuerst berechnen wir mittels der Funktion *hadamard* eine Approximation S der Hadamard–Schranke für die Matrix A.
  2. Anschließend lesen wir mittels der Funktion *get_primes* eine Liste von Primzahlen $m_1, \ldots, m_n \in \mathbb{P}$ ein, mit $m_1 \cdot \ldots \cdot m_n > 2 \cdot S$.
  3. Für jede dieser Primzahlen m_i berechnen wir den Rang $r(m_i)$ der Matrix $(A \bmod m_i)$. Dies tun wir essentiell durch die Berechnung einer Spalten–Stufen–Form.

4. Den Rang r der Matrix A erhalten wir nun durch die Berechnung von $r = \max_{i=0,\ldots,n} r(m_i)$.

5. Dann geben wir r als Funktionswert zurück.

Die detaillierte Beschreibung des hier verwendeten Algorithmus findet der interessierte Leser in [21] (Seite 90). Der dort dargestellte Algorithmus berechnet den Rang und die Indizes der linear unabhängigen Zeilen einer Matrix. Wir haben diesen Algorithmus für unsere Zielsetzung verändert, indem wir alle Berechnungschritte, die sich lediglich mit der Berechnung der Indizes der linear unabhängigen Zeilen beschäftigen, weggelassen haben.
Es ist leicht einzusehen, daß diese Modifikation keinen Einfluß auf die Korrektheit der Rangberechnung hat.

- *Fehlermeldungen:*
  keine

- *Besonderheiten:*
  keine

Laufzeitvergleich:

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:09	00:00:06	–	00:01:06
20×20	00:00:99	00:01:15	–	00:17:76
30×30	00:04:62	00:07:71	–	02:02:18
40×40	00:14:14	00:28:49	–	07:26:00
50×50	00:33:07	01:22:86	–	21:48:83
60×60	01:07:70	–	–	52:06:33
70×70	02:05:03	–	–	–

Tabelle 9.4.1: Laufzeitvergleich – Berechnung des Rangs einer Matrix

MATHEMATICA stellt unseres Wissens keine eigene Funktion zur Rangbestimmung bereit.

9.4.2 Linear unabhängige Zeilen und linear unabhängige Spalten

Zielsetzung der linearen Algebra:

- Berechne die Anzahl und die Indizes der linear unabhängigen Zeilen einer Matrix $A \in \mathbb{Z}^{m,n}$!
- Berechne die Anzahl und die Indizes der linear unabhängigen Spalten einer Matrix $A \in \mathbb{Z}^{m,n}$!

Prototypen:

- Die Funktionen
  - `long *A.lininr() const`
  - `long *lininr(const bigint_matrix &A)`

  geben ein Array über *long* zurück, sagen wir v, wobei gilt:

  1. $v[0]$ entspricht dem Rang der Matrix A.
  2. $v[1], \ldots, v[v[0]]$ sind die Indizes der linear unabhängigen Zeilen der Matrix A.

- Die Funktionen
  - `long *A.lininc() const`
  - `long *lininc(const bigint_matrix &A)`

  geben ein Array über *long* zurück, sagen wir v, wobei gilt:

  1. $v[0]$ entspricht dem Rang der Matrix A.
  2. $v[1], \ldots, v[v[0]]$ sind die Indizes der linear unabhängigen Spalten der Matrix A.

Implementierung:

- *Vorgehensweise:*
  Wir beschreiben an dieser Stelle lediglich die Vorgehensweise zur Berechnung der Anzahl und der Indizes der linear unabhängigen Zeilen. Um die Anzahl und die Indizes der linear unabhängigen Spalten zu berechnen, führen wir die im folgenden beschriebene Berechnung auf der transponierten Matrix aus.

  Die Berechnung des Rangs und der linear unabhängigen Zeilen der Matrix A mittels der Funktion *A.lininr()* verläuft wie folgt:

1. Zuerst berechnen wir mittels der Funktion *hadamard* eine Approximation S der Hadamard–Schranke für die Matrix A.
2. Anschließend lesen wir mittels der Funktion *get_primes* eine Liste von Primzahlen $m_1, \ldots, m_n \in \mathbb{P}$ ein, mit $m_1 \cdot \ldots \cdot m_n > 2 \cdot S$.
3. Für jede dieser Primzahlen m_i berechnen wir den Rang $r(m_i)$ und die Indizes $i_1(m_i), \ldots, i_{r(m_i)}(m_i)$ der linear unabhängigen Zeilen der Matrix $(A \bmod m_i)$. Dies tun wir essentiell durch die Berechnung einer Spalten–Stufen–Form.
4. Den Rang r der Matrix A erhalten wir nun durch die Berechnung von $r = \max\limits_{i=0\ldots n} r(m_i)$.
5. Falls $r(m_i) < r$ für ein Element der verwendeten Primzahlliste ist, setze $i_j(m_i) \;=\; 0 \quad j \in \{r(m_i)+1, \ldots, r\}$.
6. Die Indizes $i_1, \ldots, i_r$ erhalten wir nun durch:

$$\begin{array}{rcl} i_1 & = & \max\limits_{i=0,\ldots,n} i_1(m_i) \\ \vdots & \vdots & \\ i_r & = & \max\limits_{i=0,\ldots,n} i_r(m_i) \end{array}$$

7. Dann geben wir r und $i_1, \ldots, i_r$ als Array über *long* als Funktionswert zurück.

Wie bereits bei der Berechnung des Rangs ist die detaillierte Beschreibung des hier verwendeten Algorithmus in [21] (Seite 90) zu finden.

- *Bemerkung:*
  In [12] wird vorgeschlagen, den Rang modulo einer großen, zufällig gewählten Primzahl zu berechen und wenn nötig, eine neue Berechnung durchzuführen, wenn klar ist, daß der Rang nicht korrekt ist. Wir haben diesen Ansatz verworfen, um dem Anwender die Korrektheit des Ergebnisses garantieren zu können.

- *Fehlermeldungen:*
  keine

- *Besonderheiten:*
  Die Funktionen *A.lininr()*, *lininr(A)* *A.lininc()* und *lininc(A)* allokieren ein Array vom Typ *long*, um den Rang und die Indizes der über $\mathbb{Z}$ linear unabhängigen Zeilen zu speichern und zurückzugeben. Für den Anwender bedeutet dies, daß er sich um die Speicherfreigabe kümmern muß.

Laufzeitvergleich:

Aufgrund dessen, daß wir keine Funktionen mit gleicher bzw. ähnlicher Funktionalität in den Systemen MATHEMATICA, PARI und MAPLE finden konnten, geben wir an dieser Stelle lediglich die Laufzeiten der *bigint_matrix*–Routine an.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×20	00:00:22	–	–	–
20×40	00:02:43	–	–	–
30×60	00:11:64	–	–	–
40×80	00:35:72	–	–	–
50×100	01:23:28	–	–	–
60×120	02:50:98	–	–	–
70×140	05:14:56	–	–	–

Tabelle 9.4.2: Laufzeitvergleich – Berechnung der linear unabhängigen Zeilen

9.4.3 Determinante

Zielsetzung der linearen Algebra:

Berechne die Determinante einer linearen Abbildung ϕ von $\mathbb{Z}^n$ nach $\mathbb{Z}^n$ bzw. berechne die Determinante der Abbildungsmatrix von ϕ!

Prototypen:

- Die Funktion
  - `void A.det(bigint &DET) const`

  speichert die Determinante von A in der Variablen DET.
- Die Funktionen
  - `bigint A.det() const`
  - `bigint det(const bigint_matrix &A)`

  geben die Determinante der Matrix A als Funktionswert zurück.

Implementierung:

- *Vorgehensweise:*
  Die Funktion *A.det(DET)* berechnet die Determinante einer Matrix A. Die Vorgehensweise ist dabei wie folgt:

1. Zuerst überprüfen wir, ob die Matrix A quadratisch ist. Ist sie nicht quadratisch, geben wir die weiter unten unter dem Stichpunkt *Fehlermeldung* angegebene Meldung aus und beenden die Funktion.
2. Ist sie quadratisch, berechnen wir mittels der Funktion *hadamard* eine Approximation S der Hadamard–Schranke für die Matrix A.
3. Anschließend lesen wir mittels der Funktion *get_primes* eine Liste von Primzahlen $m_1, \ldots, m_n \in \mathbb{P}$ ein, mit $m_1 \cdot \ldots \cdot m_n > 2 \cdot S$.
4. Für jede dieser Primzahlen m_i berechnen wir die Determinante DET_{m_i} der Matrix $(A \bmod m_i)$. Dies tun wir mit dem Algorithmus von Gauß–Bareisse, der detailliert in [5] beschrieben ist.
5. Die Determinante DET der Matrix A erhalten wir jetzt durch Anwendung des Chinesischen Restsatzes in der Funktion *chinrest* auf die *bigint*–Werte $DET_1, \ldots, DET_n$.

Dieser Algorithmus basiert im wesentlichen auf dem Algorithmus aus [21] (Seite 106). Lediglich der Algorithmus zur Berechnung der Determinanten innerhalb endlicher Körper wurde durch den eben erwähnten Algorithmus von Gauß–Bareiss [5] ersetzt. Die Korrektheit des Gesamtalgorithmus ergibt sich aus der Korrektheit der verwendeten Teilalgorithmen.

- *Fehlermeldungen:*
  Wenn die Matrix A nicht quadratisch ist, endet die Funktion mit der Fehlermeldung:

  det :: non square matrix !

- *Besonderheiten:*
  keine

Laufzeitvergleich:

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:11	00:00:05	00:00:50	00:00:91
20×20	00:00:95	00:01:17	00:02:31	00:11:76
30×30	00:04:33	00:07:47	00:07:41	01:13:13
40×40	00:13:38	00:27:84	00:19:00	04:48:16
50×50	00:31:61	01:20:84	00:41:02	13:34:10
60×60	00:52:93	–	01:17:56	32:40:06
70×70	01:36:01	–	02:17:13	69:12:30
80×80	02:42:84	–	03:49:40	133:02:36
90×90	04:17:52	–	05:59:12	236:16:70
100×100	06:34:80	–	08:52:47	402:15:66

Tabelle 9.4.3: Laufzeitvergleich – Berechnung der Determinanten

9.4.4 Adjungierter Homomorphismus

Zielsetzung der linearen Algebra:

Berechne den adjungierten Homomorphismus zu ϕ d.h. die adjungierte Matrix zur Abbildungsmatrix von ϕ!

Prototypen:

- `void A.adj(const bigint_matrix &B) const`
  speichert in der Matrixvariablen A die adjungierte Matrix zur Matrix B.
- `bigint_matrix adj(const bigint_matrix &B)`
  gibt die adjungierte Matrix der Matrix B zurück.

Implementierung:

- *Vorgehensweise:*
  Die Berechnung der adjungierten Matrix A zu einer Matrix B mittels der Funktion *A.adj(B)* verläuft nach folgendem Schema:

  1. Zuerst überprüfen wir, ob die Matrix B quadratisch ist. Ist sie nicht quadratisch, geben wir die weiter unten folgende Fehlermeldung aus und beenden die Funktion.
  2. Ist sie quadratisch, berechnen wir mittels der Funktion *hadamard* eine Approximation S der Hadamard–Schranke für die Matrix B.
  3. Nun berechnen wir die Determinante DET der Matrix B.
  4. Anschließend lesen wir mittels der Funktion *get_primes* eine Liste von Primzahlen $m_1, \ldots, m_n \in \mathbb{P}$ ein, mit $m_1 \cdot \ldots \cdot m_n > 2 \cdot S$ und $m_i \nmid DET$.
  5. Für jede dieser Primzahlen m_i berechnen wir die adjungierte Matrix ADJ_{m_i} der Matrix $(A \bmod m_i)$. Dies tun wir essentiell durch die Berechnung der inversen Matrix und der Determinanten innerhalb der endlichen Körper.
  6. Die adjungierte Matrix A erhalten wir jetzt durch Anwendung des Chinesischen Restsatzes mittels der Funktion *chinrest* auf die Matrizen $ADJ_1, \ldots, ADJ_n$.

  Der Algorithmus wird etwas genauer in [21] (Seite 78) beschrieben.

- *Fehlermeldungen:*
  Wenn die Matrix B nicht quadratisch ist, endet die Funktion mit der Fehlermeldung:

  $\boxed{\textbf{adj :: non square matrix !}}$

- *Besonderheiten:*
  keine

Laufzeitvergleich:

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:48	00:00:24	–	01:00:15
20×20	00:05:58	00:06:40	–	–
30×30	00:26:01	00:45:88	–	–
40×40	01:25:07	03:08:37	–	–
50×50	03:18:84	09:30:40	–	–
60×60	06:42:87	–	–	–
70×70	14:51:06	–	–	–
80×80	25:09:67	–	–	–
90×90	39:28:87	–	–	–

Tabelle 9.4.4: Laufzeitvergleich – Berechnung der adjungierten Matrix

Nach unserem Wissen stellt MATHEMATICA keine eigene Funktion zur Berechnung der adjungierten Matrix bereit.

9.4.5 Charakteristisches Polynom

Zielsetzung der linearen Algebra:

Berechne das charakteristische Polynom der Abbildung ϕ bzw. der Abbildungsmatrix A zu ϕ!

Prototypen:

Die Funktionen

- `bigint *A.charpoly() const`
- `bigint *charpoly(const bigint_matrix &A)`

liefern ein Array über *bigint*, sagen wir v zurück, wobei gilt:
$char(\phi) = v[0] \cdot x^0 + v[1] \cdot x^1 + \cdots + v[k] \cdot x^k$ mit $k = A.rows = A.columns$.

Implementierung:

- *Vorgehensweise:*
  Die Berechnung des charakteristischen Polynoms einer Matrix A erfolgt nach folgender Vorgehensweise:

  1. Zuerst überprüfen wir, ob die Matrix A quadratisch ist. Ist sie nicht quadratisch, geben wir die unten folgende Fehlermeldung aus und beenden den Algorithmus.
  2. Ist sie quadratisch, berechnen wir $S = (k \cdot |A|)^k$, wobei wir mit $|A|$ den betragsmäßig größten Eintrag der Matrix A bezeichnen.
  3. Anschließend lesen wir mittels der Funktion *get_primes* eine Liste von Primzahlen $m_1, \ldots, m_n \in \mathbb{P}$ ein, mit $m_1 \cdot \ldots \cdot m_n > 2 \cdot S$.
  4. Für jede dieser Primzahlen m_i berechnen wir den Vektor der Koeffizienten $v(m_i)$ des charakteristischen Polynoms der Matrix $(A \bmod m_i)$. Dazu transformieren wir die Matrix $(A \bmod m_i)$ auf Hessenberg–Form und berechnen dann das charakteristische Polynom.
  5. Den Koeffizientenvektor v der Matrix A erhalten wir durch komponentenweise Anwendung des Chinesischen Restsatzes auf die Vektoren $v(m_1), \ldots, v(m_n)$.
  6. Den Vektor (ein eindimensionales Array) v geben wir nun zurück.

  Die detaillierte Beschreibung dieses Algorithmus findet man in [21] (Seite 86).

- *Fehlermeldungen:*
  Wenn die Matrix A nicht quadratisch ist, endet die Funktion mit der Fehlermeldung:

  charpoly :: non square matrix !

- *Besonderheiten:*
  Die Funktionen *A.charpoly()* und *charpoly(A)* allokieren ein Array vom Typ *bigint*, um die Koeffizienten des charakteristischen Polynoms zu speichern und zurückzugeben. Der Anwender dieser Funktion muß die Speicherfreigabe übernehmen.

Laufzeitvergleich:

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:36	00:00:23	–	27:43
20×20	00:04:19	00:06:32	–	8:05:56
30×30	00:19:70	00:46:66	–	51:47:63
40×40	01:01:23	03:10:53	–	214:44:38
50×50	02:28:77	09:25:19	–	667:39:73
60×60	05:10:21	–	–	–
70×70	09:55:40	–	–	–
80×80	16:42:75	–	–	–
90×90	28:05:29	–	–	–

Tabelle 9.4.5: Laufzeitvergleich – Berechnung des charakteristischen Polynoms

Die Berechnung des charakteristischen Polynoms läßt sich in MATHEMATICA über die Determinantenberechnung realisieren. Ein Laufzeitvergleiche erschien uns aber unfair, weil andere Systeme zur Berechnung des charakteristischen Polynoms eigene Funktionen bereit stellen. Aus diesem Grund haben wir darauf verzichtet.

Kapitel 10

Modulare Arithmetik II

Der zweite Ansatz zur Bewältigung des Problems der *entry explosion*, der hier die Bezeichnung *modulare Arithmetik II* trägt, ging aus Untersuchungen zur Berechnung von Matrix–Normalformen über $\mathbb{Z}$ hervor. Dementsprechend liegt das Hauptanwendungsgebiet dieses Ansatzes in der Normalformberechnung. Die Grundidee beruht auf der Tatsache, daß sich das von den Spalten einer Matrix $A \in \mathbb{Z}^{n \times m}$ erzeugte Modul $\mathbb{L}(A)$ nicht ändert, wenn man $\alpha \cdot I_n$ mit A „konkateniert". Hierbei bezeichnen wir mit I_n die Einheitsmatrix in $\mathbb{Z}^{n \times n}$ und mit α ein positives, ganzzahliges Vielfaches der sogenannten Gitterdeterminanten von $\mathbb{L}(A)$[21],d.h. des Volumens des von den Spalten der Matrix A aufgespannten Parallelotops. Es gilt also:

$$\mathbb{L}(A) = \mathbb{L}(A, \alpha \cdot I_n)$$

Hieraus resultiert folgende Vorgehensweise zur Begrenzung der Größe der Zwischenresultate bei der Berechnung einer Normalform (NF) einer ganzzahligen Matrix A mit einem ganzzahligen Vielfachen α der Gitterdeterminanten:

$$NF(A) = NF(A, \alpha \cdot I_n) = NF(A \bmod \alpha, \alpha \cdot I_n)$$

Wir können also die Normalform einer Matrix A berechnen, indem wir die Normalform in $\mathbb{Z}/\alpha\mathbb{Z}$ berechnen und anschließend die Diagonalmatrix $\alpha \cdot I_n$ zum Beispiel durch Gauß–Elimination entfernen. Denjenigen Leser, der sich für diese Thematik näher interessiert, verweisen wir auf [21],[7] und [9].

Die Größe der Zwischenresultate ist bei diesem Schema durch das Vielfache der Gitterdeterminanten α beschränkt. Sowohl in der Methode der *modularen Arithmetik I* als auch bei der hier vorgestellten Methode wird, um der *entry explosion* entgegenzuwirken, die Berechnung eines Objektes auf die Berechnung adäquater Objekte in Körpern ($\mathbb{F}_{m_i}$) bzw. Ringen ($\mathbb{Z}/\alpha\mathbb{Z}$) verlagert. Um die Verwandtschaft dieser beiden Methoden hervorzuheben, haben wir diese zweite Technik als *modulare Arithmetik II* bezeichnet.

Bei der Betrachtung der *modularen Arithmetik I* haben wir festgestellt, daß die verwendeten Primzahlen für die Laufzeit der Funktionen von außerordentlicher Bedeutung sind. Übertragen auf den Ansatz der *modularen Arithmetik II* bedeutet dies, daß die Laufzeit der Algorithmen direkt von der Größe des Vielfachen der Gitterdeterminanten abhängt. Um die Laufzeit der Implementierung zu verbessern, müssen wir unser Bestreben darauf ausrichten, ein möglichst kleines Vielfaches der Gitterdeterminanten zu finden. Ist die Bitlänge dieses Vielfachen kleiner als die Hälfte der Bitlänge des größtmöglichen ganzzahligen positiven *double*–Eintrages, würde sogar (in Anlehnung an die *modulare Arithmetik I*) die Einführung einer *long*–Variante Geschwindigkeitszuwachs versprechen.

10.1 Berechnung der Gitterdeterminanten

Die Größe des Vielfachen der Gitterdeterminanten ist für die Effizienz eines Algorithmus, der sich der Methode der *modularen Arithmetik II* zur Bekämpfung der *entry explosion* bedient, von außerordentlicher Bedeutung. Doch wie berechnet man nun solch ein Vielfaches?

Ein erster Ansatz zur Berechnung eines Vielfachen der Gitterdeterminanten einer Matrix A, der auch in [21] Verwendung findet, besteht darin, eine maximale reguläre Untermatrix von A zu suchen, deren Rang dem Rang von A entspricht, und davon die Determinante zu berechnen. Die praktischen Resultate dieser Methode sind jedoch nicht überzeugend.

Deshalb wird dieser Ansatz wie folgt zu einer zweiten Methode erweitert:

Anstatt nur eine maximale reguläre Untermatrix zu suchen, suchen wir mehrere und berechnen jeweils die Determinante. Als Vielfaches der Gitterdeterminanten wählen wir dann den größten gemeinsamen Teiler der berechneten Determinanten.

Diese Variante hat den Vorteil, daß die so erhaltenen Vielfachen wesentlich kleiner als die vorherigen sind. Sie hat aber den Nachteil, daß jede Berechnung einer regulären Untermatrix und der jeweiligen Determinanten teuer im Verhältnis zur Gesamtberechnung ist. Deshalb stellen sich uns nun folgende Fragen:

1. Wieviele maximale reguläre Untermatrizen garantieren ein optimales Ergebnis?

2. Wieviel Zeit darf man in die Berechnung des Vielfachen der Gitterdeterminanten investieren, damit entsprechende Zeitvorteile in der folgenden Berechnung wirksam werden?

Die erste Fragestellung führt auf direktem Wege zu folgendem Problem:

Gibt es zu einer Menge M positiver ganzer Zahlen mit $len \leq \#M$ eine echte Teilmenge N mit $\#N \leq len$, so daß $\mathrm{ggT}(x \in M) = \mathrm{ggT}(x \in N)$?
Diese Problemstellung bezeichnet man auch als das **MINIMUM–GCD–SET–Problem**.

Nach einem Resultat aus [19] bzw. [20] gilt folgendes:

10.1. Satz
Das MINIMUM GCD SET *Problem ist NP-vollständig.*

Beweis: Siehe [20]. ■

Somit streben wir nicht danach, eine Lösung zu finden, sondern nach Heuristiken, die eine für unsere Belange gute Lösung liefern.

Mit dieser Zielsetzung haben wir verschiedene Versuche mit einer unterschiedlichen Anzahl an zu wählenden Untermatrizen durchgeführt, die nach unterschiedlichen Konstruktionsschemata berechnet wurden. Dabei haben wir folgendes Ergebnis erhalten:

Methode 1: Für nichtquadratische, zufällig erzeugte Matrizen mit Einträgen betragsmäßig kleiner als $2^{31}-1$ ist der „Trade of" zwischen der Anzahl der Untermatrizen und der Größe des Vielfachen der Gitterdeterminanten optimal, wenn zunächst zwei reguläre Untermatrizen, die möglichst wenige Einträge gemeinsam haben, gesucht werden, dann die Determinanten dieser erhaltenen Teilmatrizen berechnet werden und zuletzt der ggT über den Determinanten gebildet wird.

Methode 2: Für quadratische Matrizen ist die Wahl nur einer regulären Untermatrix mit maximalem Rang optimal, weil in vielen Fällen die Determinante der quadratischen Matrix der Gitterdeterminanten entspricht, d.h. daß die quadratische Matrix regulär ist.

In der Klasse *bigint_matrix* dienen folgende Funktionen zur Berechnung eines Vielfachen der Gitterdeterminanten:

Prototypen:

1. Die Funktionen

   - `void A.latticedet(bigint &LDET) const`
   - `void A.latticedet2(bigint &LDET) const`
   - `void A.latticedet3(bigint &LDET) const`

   speichern ein Vielfaches der Gitterdeterminanten des von den Spalten der Matrix A erzeugten Gitters in der Variablen $LDET$.

2. Die Funktionen

   - `bigint A.latticedet() const`
   - `bigint latticedet(const bigint_matrix &A)`
   - `bigint A.latticedet2() const`
   - `bigint latticedet2(const bigint_matrix &A)`

- `bigint A.latticedet3() const`
- `bigint latticedet3(const bigint_matrix &A)`

geben ein Vielfaches der Gitterdeterminanten des von den Spalten der Matrix A erzeugten Gitters zurück.

Implementierung:

In der Beschreibung der Implementierung beziehen wir uns auf die oben angeführte Unterteilung in zwei Methoden.

- Die Funktion *A.latticedet(LDET)* berechnet die Gitterdeterminante nach der *Methode 2*. Die Funktion *A.latticedet2(LDET)* verwendet dazu die *Methode 1*. Die Berechnung einer regulären Untermatrix in beiden Funktionen bedient sich der Funktionen *lininr* und *lininc*.
  Bei der Funktion *A.latticedet3(LDET)* verwenden wir ebenfalls die *Methode 1*, setzen aber voraus, daß die Matrix A vollen Zeilenrang hat. Unter dieser Voraussetzung können wir Laufzeit gegenüber der Funktion *A.latticedet2(LDET)* einsparen, weil die Benutzung der Funktion *lininr* entfällt.
- Die Funktionen der zweiten Gruppe bilden, wie bereits einige Funktionen im Kapitel *modulare Arithmetik I*, lediglich Benutzerschnittstellen zu den Funktionen der ersten Gruppe.

10.2 Modulare Berechnung der Smith–Normalform

Gehen wir nun zunächst darauf ein, was man unter der Smith–Normalform einer ganzzahligen Matrix A versteht.

10.2. Definition
Eine ganzzahlige $(m \times n)$– Matrix A ist in **Smith–Normalform (SNF)**, *wenn sie den Bedingungen*

$$\begin{array}{rll} a_{i,j} = 0 & \textit{für} & i \neq j \wedge 1 \leq i \leq m \wedge 1 \leq j \leq n \\ a_{i,i} \geq 0 & \textit{für} & 1 \leq i \leq m \\ a_{i-1,i-1} | a_{i,i} & \textit{für} & 2 \leq i \leq m \end{array}$$

genügt.

Sei nun A eine ganzzahlige Matrix mit vollem Zeilenrang. Desweiteren sei *LDET* ein ganzzahliges Vielfaches der Gitterdeterminanten von A.

In der Klasse *bigint_matrix* wurden folgende Funktionen, die sich der *modularen Arithmetik II* bedienen und die die Matrix A in die Smith–Normalform überführen, implementiert:

Prototypen:

- `void A.snfmod_dkt()`
- `void A.snfmod_dkt(const bigint &LDET)`
- `void snfmod_dkt(bigint_matrix &A)`
- `void snfmod_dkt(bigint_matrix &A, const bigint &LDET)`

und

- `void A.snfmod_cohen()`
- `void A.snfmod_cohen(const bigint &LDET)`
- `void snfmod_cohen(bigint_matrix &A)`
- `void snfmod_cohen(bigint_matrix &A, const bigint &LDET)`

Implementierung:

Wie die Namensgebung der Funktionsgruppen bereits verrät wurden zwei verschiedene Algorithmen zur Berechnung der Smith–Normalform verwendet.

1. Der erste basiert auf einem Algorithmus von P. D. Domich, R. Kannan und L. E. Trotter zur Berechnung der Hermite–Normalform. Die in diesem Algorithmus [8], [7] enthaltene Idee wurde von J. L. Hafner und K. S. McCurley in [10] für die Berechnung der Smith–Normalform dargestellt. Implementiert haben wir dies in den folgenden Funktionen:

   - Die Funktionen *A.snfmod_dkt(LDET)* und *snfmod_dkt(A,LDET)* transformieren die Matrix A in Smith–Normalform. Dabei benutzen sie das Vielfache $LDET$ der Gitterdeterminanten des von den Spalten der Matrix A erzeugten Gitters zur Kontrolle der Zwischenergebnisse. Beide Funktionen brechen mit der Fehlermeldung

     snfmod_dkt :: number of rows != rank

     ab, wenn die Matrix A nicht den vollen Zeilenrang besitzt.
   - Die Funktionen *A.snfmod_dkt()* und *snfmod_dkt(A)* berechnen ein Vielfaches der Gitterdeterminanten $LDET$ des von den Spalten der Matrix A erzeugten Gitters mittels der Funktion *latticedet2* und rufen anschließend die Funktion *A.snfmod_dkt(LDET)* auf.

2. Der zweite Algorithmus stellt eine Variante des ersten dar. In [5] wird beschrieben, wie man die Abfolge der Schritte zur Berechnung der *Hermite–Normalform*, die in [8] beschrieben ist, verändern kann, um dadurch vielleicht eine bessere Effizienz zu erreichen. Diese Idee haben wir in Anlehnung an [10] übertragen. Die Korrektheit dieser Modifikation ergibt sich direkt aus der Korrektheit der Modifikation in [5]. Die genaue Beschreibung der einzelnen Funktionen ist analog zur obigen.

Laufzeitvergleich:

Die System PARI, MATHEMATICA und MAPLE stellen unseres Wissens keine modularen Funktionen zur Berechnung der Smith–Normalform bereit. Deshalb geben wir an dieser Stelle einen LiDIA–internen Laufzeitvergleich der Funktionen *snfmod_dkt* und *snfmod_cohen* an. Dabei berechnen wir die Gitterdeterminante $LDET$ vor und geben sie den Funktionen beim Aufruf mit.

Dimension	*snfmod_dkt*	*snfmod_cohen*
	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:61	00:00:43
10×20	00:00:68	00:00:40
20×20	00:08:52	00:08:69
20×40	00:05:94	00:03:98
30×30	00:45:13	01:08:06
30×60	00:19:31	00:14:71
40×40	02:14:18	04:37:53
40×80	01:05:65	00:46:97
50×50	06:33:72	16:35:86
50×100	01:24:77	01:30:05

Tabelle 10.2.1: Laufzeitvergleich – modulare Berechnung der Smith–Normalform

10.3 Modulare Berechnung der Hermite–Normalform

Analog zum Abschnitt *Modulare Berechnung der Smith–Normalform* geben wir zunächst eine Definition der Hermite–Normalform einer ganzzahligen Matrix an.

10.3. Definition
Eine ganzzahlige $(m \times n)$*–Matrix* A *mit* $rang(A) = m$ *ist in* **Hermite–Normalform (HNF)**, *wenn* A *eine obere Dreiecksmatrix ist und den Bedingungen*

$$\begin{array}{rl} a_{i,n-m+i} \geq 0 & \textit{für} \quad 1 \leq i \leq m \\ 0 \leq a_{i,j} < a_{i,n-m+i} & \textit{für} \quad j \in \{n-m+i, \ldots, m\} \end{array}$$

genügt.

In Analogie zur Smith–Normalform transfomieren die folgenden Funktionen eine ganzzahlige Matrix A in Hermite–Normalform:

Prototypen:

- `void A.hnfmod_dkt()`
- `void A.hnfmod_dkt(const bigint &LDET)`
- `void hnfmod_dkt(bigint_matrix &A)`
- `void hnfmod_dkt(bigint_matrix &A, const bigint &LDET)`

und

- `void A.hnfmod_cohen()`
- `void A.hnfmod_cohen(const bigint &LDET)`
- `void hnfmod_cohen(bigint_matrix &LDET)`
- `void hnfmod_cohen(bigint_matrix &A, const bigint &LDET)`

Implementierung:

In Analogie zur Berechnung der Smith–Normalform wurden zur Berechnung der Hermite–Normalform ebenfalls zwei Algorithmentypen implementiert.

1. Der erste Algorithmus ist eine Implementierung des Algorithmus von Domich, Kannan und Trotter. Er wird detailliert in [8] und [10] beschrieben.

   - Die Funktionen *A.hnfmod_dkt(LDET)* und *hnfmod_dkt(A,LDET)* transformieren die Matrix A in Hermite–Normalform. Dabei benutzen sie das Vielfache $LDET$ der Gitterdeterminanten des von den Spalten der Matrix A erzeugten Gitters zur Reduktion der Zwischenergebnisse. Beide Funktionen brechen mit der Fehlermeldung

     hnfmod_dkt :: number of rows != rank

ab, wenn die Matrix A nicht den vollen Zeilenrang besitzt.

- Die Funktionen *A.hnfmod_dkt()* und *hnfmod_dkt(A)* berechnen ein Vielfaches der Gitterdeterminanten $LDET$ des von den Spalten der Matrix A erzeugten Gitters und rufen anschließend die Funktion *A.hnfmod_dkt(LDET)* auf.

2. Der zweite Algorithmus basiert auf einer Modifikation dieser Berechnungsabfolge. Detailliert wird dieser Algorithmus in [5] beschrieben. Die Detaillbeschreibung der einzelnen Funktionen ist analog zu oben.

Laufzeitvergleich:

In den Laufzeiten der folgenden Tabelle sind die Zeiten zur Berechnung eines Vielfachen der Gitterdeterminanten nicht enthalten. Die Gitterdeterminante $LDET$ wird den Funktionen beim Aufruf mitgegeben. Als Funktion haben wir *hnf_dkt(LDET)* ausgewählt.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:66	00:00:37	–	–
20×20	00:11:11	00:09:27	–	–
30×30	00:74:13	02:24:60	–	–
40×40	05:59:57	12:10:20	–	–
50×50	17:47:09	39:37:57	–	–

Tabelle 10.5.1: Laufzeitvergleich – modulare Berechnung der Hermite–Normalform

In den Systemen MATHEMATICA und MAPLE konnten wir keine Funktionen finden, die Berechnungen dieser Art ausführen.

Desweiteren möchten wir an dieser Stelle zwei weitere Funktionen angeben. Sie berechnen eine Transformationsmatrix T zu einer ganzzahligen Matrix A, so daß $A \cdot T =$*HNF(A)* auf der Basis der modularen Hermite–Normalform–Berechnung.

Prototypen:

Die Funktionen

- `void A.hnfmod_mueller(bigint_matrix & T)`
- `void hnfmod_mueller(bigint_matrix & A, const bigint_matrix & T)`

formen T in eine unimodulare Transformationsmatrix um, mit $A \cdot T =$*HNF(A)*.

Implementierung:

Für diese Funktionen wurde der Algorithmus aus der Diplomarbeit *Effiziente Algorithmen für Probleme der linearen Algebra über* $\mathbb{Z}$ von Achim Müller [21] (Seiten 146–148) verwendet. Dort findet der Leser eine genaue Beschreibung, auf die wir an dieser Stelle verzichten möchten.

Laufzeitvergleich:

Auf einen Laufzeitvergleich verzichten wir an dieser Stelle ebenfalls und verweisen auf einen Laufzeitvergleich gegen Ende des folgenden Kapitels.

Kapitel 11

Nichtmodulare Algorithmen

Der letzte Ansatz, den wir hier anführen möchten, unterscheidet sich grundlegend von den beiden vorherigen. Während Ansätze, die sich modularer Techniken bedienen (*modulare Arithmetik I* und *II*), von Beginn an eine obere Schranke für die maximale Größe der Zwischenergebnisse angeben und bei Überschreitung dieser Schranke eine entsprechende Reduktion (z.B. durch Berechnungen innerhalb endlicher Körper) ausführen, wird bei dieser Vorgehensweise versucht, das Anwachsen der Zwischenergebnisse in einem Berechnungsschritt möglichst klein zu halten. Im Unterschied zu den Techniken der modularen Arithmetik läßt sich hier kein Schema angeben, das uns die Eindämmung der *entry explosion* garantiert. Vielmehr besteht dieser Ansatz aus einer Ansammlung von Ideen, die sich positiv auf die Größe der Zwischenergebnisse und damit auf die Effizienz auswirken.

11.1 Ausgangspunkt

Im folgenden werden wir am Beispiel der Berechnung der Smith–Normalform und der Hermite–Normalform einige dieser Ideen vorstellen. Wir möchten betonen, daß sie sich problemlos auf andere Berechnungen übertragen lassen. Im Vergleich der Effizienz mit der der modularen Techniken macht sich deren Einsatz aber lediglich bei Algorithmen der *modularen Arithmetik II* bezahlt. Aus diesem Grund benutzen wir zur Darstellung der Ideen die Berechnung der Smith– und der Hermite–Normalform.

11.1.1 Ein Algorithmus zur Berechnung der Smith–Normalform

Ausgangspunkt unserer Betrachtung ist für die Smith–Normalform–Berechnung eine Variante des Algorithmus aus dem Buch von *Hartley und Hawkes* [11]. Diese Variante wird in verschiedenen Publikationen, z.B. [15] und [13], beschrieben.

Das Ziel im folgenden ist es, eine Matrix $A \in \mathbb{Z}^{m \times n}$ mit $rank(A) = m$ in SNF zu bringen. Die Variante des Algorithmus, die wir an dieser Stelle benutzen, wandert entlang der Diagonalen abwärts und bringt iterativ die restlichen Elemente der jeweils zugehörigen Spalte und Zeile auf Null. Wir bezeichnen zur einfacheren Beschreibung den Index des aktuellen Diagonalelementes als Arbeitsindex, die Position als Arbeitsposition und die zugehörige Spalte und Zeile als Arbeitsspalte bzw. Arbeitszeile. Desweiteren bezeichnen wir den größten gemeinsamen Teiler einer Menge von Zahlen als *mgcd*, in Anlehnung an die englische Bezeichnung *multiple greatest common divisor*.

Wir beschreiben nun die Programmabfolge in der ersten Iteration, d.h. den Übergang:

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix} \longrightarrow \left(\begin{array}{c|ccc} x_{0,0} & 0 & \cdots & 0 \\ \hline 0 & x_{1,1} & \cdots & x_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & x_{m-1,1} & \cdots & x_{m-1,n-1} \end{array}\right)$$

wobei $x_{0,0}$ ein Teiler aller Elemente der Restmatrix

$$A_1 = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n-1} \\ \vdots & \ddots & \vdots \\ x_{m-1,1} & \cdots & x_{m-1,n-1} \end{pmatrix}$$

ist. Die Berechnung verläuft dabei wie folgt:

Phase 1: Wähle das Pivotelement *PIVOT*!
In dieser Phase suchen wir in dem aktuellen Pivotbereich nach einem Element ungleich Null (in [15] zusätzlich betragsmäßig minimal). Der Pivotbereich beinhaltet dabei alle Elemente rechts unterhalb der Arbeitsposition und die Elemente der Arbeitszeile und –spalte.

Phase 2: Bringe *PIVOT* zur aktuellen Arbeitsposition (0,0)!
Durch entsprechenden Spalten– bzw. Zeilentausch (swap_columns, swap_rows) bringen wir das Pivotelement auf die Position (0,0).

Phase 3: mgcd– und Eliminationsphase

(a) **Berechne den mgcd der Arbeitszeile und eine entsprechende Transformationsmatrix T_1!**

In dieser Phase berechnen wir eine Transformationsmatrix T_1, so daß

$$A \cdot T_1 = \begin{pmatrix} x_{0,0} & 0 & \cdots & 0 \\ & \cdots & \cdots & * \\ \vdots & \vdots & \vdots & \vdots \\ & \cdots & \cdots & * \end{pmatrix}$$

mit $x_{0,0}$ =ggT(Elemente der Arbeitszeile). Dazu benutzen wir den Algorithmus von Blankinship, den wir im Abschnitt *mgcd–Berechnung* näher beleuchten werden. Anschließend berechnen wir $A \leftarrow A \cdot T_1$.

(b) **Berechne den mgcd der Arbeitsspalte und eine entsprechende Transformationsmatrix T_2!**
In dieser Phase berechnen wir unter Benutzung des Algorithmus von Blankinship die Transformationsmatrix T_2, so daß

$$T_2 \cdot A = \begin{pmatrix} x_{0,0} & * & \cdots & * \\ 0 & * & \cdots & * \\ \vdots & \vdots & \vdots & \vdots \\ 0 & * & \cdots & * \end{pmatrix}$$

mit $x_{0,0}$ =ggT(Elemente der Arbeitsspalte). Anschließend berechnen wir $A \leftarrow T_2 \cdot A$.

Phase 4: Formtest
Hier überprüfen wir, ob alle Elemente der Arbeitszeile und Arbeitsspalte, außer dem aktuellen Diagonalelement, gleich Null sind. Ist dies nicht der Fall, gehen wir zurück zur Phase 3. (Wir können durch die Multiplikation mit T_2 Elemente ungleich Null erhalten.)

Phase 5: Teilbarkeitstest
In dieser Phase überprüfen wir, ob das aktuelle Diagonalelement Teiler aller Elemente rechts unterhalb der Arbeitsposition ist. Ist dies nicht der Fall, addieren wir die Zeile, in der wir ein Element gefunden haben, welches nicht von dem Arbeitselement geteilt wird, zur Arbeitszeile und fahren mit Phase 3 fort. Teilt das aktuelle Diagonalelement alle Elemente rechts unterhalb seiner Position, gehen wir eine Iteration weiter, d.h. wir gehen in der Diagonalen ein Element abwärts und beginnen rekursiv für die Untermatrix A_1 mit der Phase 1.

11.1.2 Ein Algorithmus zur Berechnung der Hermite–Normalform

Für die Hermite–Normalform–Berechnung benutzen wir als Ausgangsalgorithmus eine erweiterte Variante der Gauß–Elimination.

Jetzt sei unser Ziel, eine Matrix $A \in \mathbb{Z}^{m \times n}$ mit $rank(A) = m$ in HNF zu bringen. Die Variante der Gauß–Elimination, die wir hier verwenden, bewegt sich entlang der Diagonalen aufwärts. Dabei bringt der Algorithmus iterativ die Elemente unterhalb der Diagonalen zeilenweise auf Null.

Betrachten wir nun den Übergang:

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix} \longrightarrow \left(\begin{array}{ccc|c} & & & x_{0,n-1} \\ & A_1 & & \vdots \\ & & & x_{m-2,n-1} \\ \hline 0 & \cdots & 0 & x_{m-1,n-1} \end{array} \right)$$

mit:

$$A_1 = \begin{pmatrix} x_{0,0} & \cdots & x_{0,n-2} \\ \vdots & \ddots & \vdots \\ x_{m-1,0} & \cdots & x_{m-2,n-2} \end{pmatrix}.$$

Die Berechnung verläuft hierbei wie folgt:

Phase 1: Wähle Pivotelement ***PIVOT*****!**
In dieser Phase suchen wir in dem aktuellen Pivotbereich nach einem Element ungleich Null. Der Pivotbereich beinhaltet dabei alle Elemente der aktuellen Arbeitszeile links von der aktuellen Arbeitsposition.

Phase 2: Bringe ***PIVOT*** **zur aktuellen Arbeitsposition (m-1,n-1)!**
Durch einen entsprechenden Spaltentausch bringen wir das Pivotelement auf die Position (m-1,n-1).

Phase 3: mgcd– und Eliminationsphase
In dieser Phase berechnen wir eine Transformationsmatrix T, so daß

$$A \cdot T = \begin{pmatrix} * & \cdots & \cdots & * \\ \vdots & \vdots & \vdots & \vdots \\ & \cdots & \cdots & * \\ 0 & \cdots & 0 & x_{m-1,n-1} \end{pmatrix}$$

mit $x_{m-1,n-1}$ =ggT(Elemente der Arbeitszeile). Dazu benutzen wir den Algorithmus von Blankinship, den wir im Abschnitt *mgcd–Berechnung* näher beleuchten werden. Anschließend bewegen wir uns auf der Diagonalen ein Element aufwärts und beginnen rekursiv für die Untermatrix A_1 wieder mit Phase 1.

Ab der Bearbeitung der Untermatrix A_1 erweitert sich der vorgestellt Algorithmus um die folgende Phase

Phase 4: Reduktionsphase
In dieser Phase werden die Elemente rechts von der aktuellen Arbeitsposition modulo des Arbeitselementes reduziert. Dazu addieren (subtrahieren) wir geeignete Vielfache der Arbeitsspalte zu (von) der zu reduzierenden Spalte.

Die hier vorgestellte Arbeitsweise entspricht im wesentlichen der aus [16]. Dort wird auch die Korrektheit dieser Vorgehensweise erläutert.

Bei näherer Betrachtung der beiden oben angegebenen Algorithmen gibt es zwei Ansatzpunkte für eine Kontrolle des Anwachsens der Zwischenergebnisse und damit für eine Effizienzsteigerung.

1. die Phase der Pivotbestimmung
2. die Phase der mgcd–Berechnung

Alle anderen Phasen hängen vollkommen deterministisch von diesen Phasen ab und sind aus diesem Grund als „Kontrollphasen“ ungeeignet.

Nun wollen wir zeigen, daß durch geschickte Veränderungen in diesen beiden Phasen die Effizienz des Algorithmus wesentlich ansteigt. Zu diesem Zweck stellen wir in den nächsten beiden Abschnitten einige Modifikationen vor und belegen deren Effizienz durch tabellarische Laufzeitvergleiche gegen Ende dieses Kapitels.

Wir zeigen zunächst, daß der Einfluß des Algorithmus zur Berechnung des größten gemeinsamen Teilers einer Menge ganzen Zahlen (*mgcd = multiple greatest common divisor*) auf die Größe der entstehenden Zwischenergebnisse und somit auf die Laufzeit enorm ist.

Aus diesem Grund werden wir zunächst die Pivotierung außer acht lassen und uns der mgcd–Berechnung zuwenden. Dabei nehmen wir der Einfachheit halber an, daß wir als Pivotelement das erste Element ungleich Null wählen.

11.2 mgcd–Berechnung

Wie bereits oben erwähnt wurde, bietet sich eine geeignete mgcd–Berechnung als Kontrollmechanismus für die Größe der entstehenden Zwischenergebnisse an. Dazu benötigen wir allerdings eine genauere Charakterisierung einer *geeigneten* mgcd–Berechnung.

Betrachten wir zunächst die Aufgabe, für die Elemente der letzten Zeile einer ganzzahligen $(m \times n)$–Matrix A den größten gemeinsamen Teiler zu suchen, diesen durch unimodulare Transformationen auf die Position $(m-1, n-1)$ zu bringen und alle übrigen Elemente in der Berechnungszeile ebenfalls durch unimodulare Transformationen zu entfernen. Dies entspricht der mgcd– und Eliminationsphase in den obigen Algorithmen. Erkenntnisse, die wir für diese Aufgabenstellung erhalten, lassen sich direkt auf die Transformation der anderen Zeilen bzw. auf die Transformation von Spalten übertragen. In Matrix–Schreibweise ist unser Ziel also:

Berechne eine unimodulare Matrix T mit:

$$A \cdot T = \begin{pmatrix} x_{0,0} & \cdots & \cdots & x_{0,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m-2,0} & \cdots & \cdots & x_{m-2,n-1} \\ 0 & \cdots & 0 & x_{m-1,n-1} \end{pmatrix}$$

wobei $x_{m-1,n-1} = \mathrm{ggT}(a_{m-1,0}, \ldots, a_{m-1,n-1})$ ist!

1. Idee: Eine erste Idee dazu ist, zunächst den ggT einer Anzahl von Elementen inklusive seiner Darstellung als Linearkombination zu berechnen und dabei die Faktoren innerhalb der Darstellung zu minimieren. Mit anderen Worten heißt dies bezogen auf die obige Aufgabe:

Suche einen ganzzahligen Vektor $x = (x_0, \ldots, x_{n-1})$ mit

$$(a_{m-1,0}, \ldots, a_{m-1,n-1}) \cdot x \quad = \quad \mathrm{ggT}(a_{m-1,0}, \ldots, a_{m-1,n-1})$$

mit

$$\prod_{i=0}^{n-1} x_i \quad \textit{minimal!}$$

Dieses Problem wurde intensiv in [19] untersucht und hat sich als sehr schwierig herausgestellt, was der folgende Satz belegt.

11.1. Satz
Für einen gegebenen Vektor von n positiven ganzen Zahlen $a = [a_1, \ldots, a_n]$ ist die Aufgabe, den kürzesten Vektor x bezüglich der L_0–Metrik oder der L_∞–Norm zu finden, so daß gilt: $a \cdot x = \sum_{i=1}^{n}(a_i \cdot x_i) = \mathrm{ggT}(a_1, \ldots, a_n)$, NP-hart.

Beweis: Siehe [20] ■

Nehmen wir einmal an, wir hätten die mgcd-Berechnung mit Faktoren annehmbarer Größe abgeschlossen, z.B. mit einem der Algorithmen von Bodewig, von Bradley oder von Hu. Außerdem hätten wir eine obere Schranke S für die Größe der Einträge im Vektor x. Um die letzte Zeile der Matrix A nun auf die oben erwähnte gewünschte Gestalt zu bringen, müssen wir die Matrix A von rechts her mit folgender Transformationsmatrix multiplizieren:

$$T_1 = \begin{pmatrix} g - x_0 \cdot a_{m-1,0} & -x_0 \cdot a_{m-1,1} & \cdots & -x_0 \cdot a_{m-1,n-2} & x_0 \\ -x_1 \cdot a_{m-1,0} & g - x_1 \cdot a_{m-1,1} & \cdots & -x_1 \cdot a_{m-1,n-2} & x_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -x_{n-2} \cdot a_{m-1,0} & -x_{n-2} \cdot a_{m-1,1} & \cdots & g - x_{n-2} \cdot a_{m-1,n-2} & x_{n-2} \\ -x_{n-1} \cdot a_{m-1,0} & -x_{n-1} \cdot a_{m-1,1} & \cdots & -x_{n-1} \cdot a_{m-1,n-2} & x_{n-1} \end{pmatrix}$$

Hierbei bezeichnen wir mit g den ggT der letzten Zeile. Durch einfaches Nachrechnen kann man verifizieren, daß diese unimodulare Transformationsmatrix das gewünschte leistet.

Um nun zum Beispiel die HNF der Matrix A zu berechnen, müssen wir $k = \min(m, n)$ solche Matrizen $T_1, \ldots, T_k$ berechnen, von rechts an die Ausgangsmatrix A heranmultiplizieren und anschließend eine Reduktion der Nichtdiagonalelemente modulo dem jeweiligen Diagonalelement durchführen. Dementsprechend erhalten wir als obere Schranke für die Größe der entstehenden Zwischenresultate vor dem Prozeß der Normalisierung eine Zahl der Größenordnung $O(|A|^{2^k})$ [14].

Dies führt für die Komplexität zu einer exponentiellen oberen Schranke. Die Laufzeiten von implementierten Algorithmen zur HNF– bzw. SNF–Berechnung, die sich dieses Ansatzes bedienen, sind denkbar schlecht. Das hatten wir entsprechend der vorhergehenden Betrachtungen erwartet.

Als Beispiele für Algorithmen zur HNF– bzw. SNF–Berechnung, die sich dieser Idee bedienen, seien die Algorithmen von Bodewig, von Bradley, Barnette und Pace angeführt. Hauptsächlich unterscheiden sich diese durch die Art und Weise, wie sie versuchen, den Vektor x mit möglichst kleinen Faktoren zu finden, und dadurch, ob sie die mgcd–Berechnung implizit in der Normalformberechnung oder explizit als Funktionsaufruf durchführen.

2. Idee: George Havas und Bohdan S. Majewski haben 1994 in verschiedenen Publikationen ein alternative Herangehensweise an obige Problemstellung dargestellt. Anstatt den größten gemeinsamen Teiler einer Menge von Elementen inklusive seiner Darstellung zu berechnen, um anschließend daraus die Transformations-

matrix T zu konstruieren, berechnen sie diese direkt in der Art und Weise, daß

$$A \cdot T = \begin{pmatrix} x_{0,0} & \cdots & \cdots & x_{0,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m-2,0} & \cdots & \cdots & x_{m-2,n-1} \\ 0 & \cdots & 0 & x_{m-1,n-1} \end{pmatrix}$$

gilt. Dabei liegt ihr Bestreben nicht darin, die Elemente der mgcd-Spalte der Transformationsmatrix klein zu halten, wie es in der Idee 1 der Fall war, sondern darin, die Einträge in jeder Spalte der Transformationsmatrix T zu minimieren.

Um das zu erreichen, modifizieren Havas und Majewski eine Variante des Algorithmus von Blankinship [2], [14]. Ausgangspunkt ist dabei folgender Algorithmus:

Variante des Algorithmus von Blankinship

EINGABE: $v = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$
AUSGABE: unimodulare Matrix $T \in \mathbb{Z}^{n\times n}$ mit $v \cdot T = (0, \ldots, 0, \mathrm{ggT}(v))$.

matrix T = $\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}$;
$j = index\_abs\_min(v)$; $\quad (|v_j| = \min(|v_0|, \ldots, |v_{n-1}|))$
swap(v_j, v_{n-1}); $\quad (v_j \longleftrightarrow v_{n-1})$
T.swap_columns($n-1,\ j$);
while $(\exists l : v_l \neq 0 \wedge\ l \neq n-1)$ **do**
  $pos\_div\_rem(q, r, v_l, v_{n-1})$; $\quad (v_l = q \cdot v_{n-1} + r \wedge r \geq 0)$
  $v_l = v_l - q \cdot v_{n-1}$; $\quad (v_l = r)$
  $t_{*,l} = t_{*,l} - q \cdot t_{*,n-1}$;
  swap(v_l, v_{n-1}); $\quad (v_l \longleftrightarrow v_{n-1})$
  T.swap_columns($n-1,\ l$);
od
return (T;)

Bemerkung:

- Im Schritt 6 dieses Algorithmus wird die Division mit Rest ausgeführt. Dabei liefert sie hier den kleinsten positiven Rest.
- Betrachtet man den Algorithmus genauer, so wird in diesem Algorithmus eine Folge von ggT–Berechnungen mittels des Euklidischen Algorithmus durchgeführt.

Dieser Algorithmus liefert unter dem Gesichtspunkt der Minimierung der Einträge in der Transformationsmatrix schlechte Resultate.

Aus diesem Grund schlagen Havas und Majewski folgende Änderungen in diesem Algorithmus, die sie in ihrer Publikation [12] als *best–remainder*–Strategie bezeichnen, vor:

1. Zunächst wird die verwendete Division mit Rest dahin gehend abgeändert, daß sie anstatt des kleinsten positiven Rests den betragsmäßig kleinsten Rest zurückliefert.
2. Desweiteren reduzieren sie mit dem aktuellen Pivotelement alle Elemente des Vektors und nicht nur ein Element des Vektors. Dies hat den Vorteil, daß nach einem Reduktionsdurchlauf alle Elemente des Vektor betragsmäßig kleiner als das aktuelle Pivotelement sind. Dies bringt Geschwindigkeitsgewinne, weil wir weniger Reduktionsdurchläufe benötigen.

Der Algorithmus hat dann folgendes Aussehen:

Modifikation von Havas und Majewski

EINGABE: $v = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$
AUSGABE: unimodulare Matrix $T \in \mathbb{Z}^{n \times n}$ mit $v \cdot T = (0, \ldots, 0, \mathrm{ggT}(v))$.

(1) matrix T =
$$\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \ddots & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix};$$
(2) $j = index\_abs\_min(v);$ $\quad(|v_j| = \min(|v_0|, \ldots, |v_{n-1}|))$
(3) $\mathrm{swap}(v_j, v_{n-1});$ $\quad(v_j \longleftrightarrow v_{n-1})$
(4) T.swap_columns($n-1$, j);

```
while (∃l : v_l ≠ 0 ∧ l ≠ n − 1) do
  for (i = 0; i < n; i++) do
    pos_div_rem(q, r, v_i, v_{n−1});            (v_i = q · v_{n−1} + r)
    v_i = v_i − q · v_{n−1};
    t_{*,i} = t_{*,i} − q · t_{*,n−1};
    j = index_abs_min(v);                        (|v_j| = min(|v_0|, ..., |v_{n−1}|))
    swap(v_j, v_{n−1});                          (v_j ⟷ v_{n−1})
    T.swap_columns(n − 1, j);
  od
od
 return (T;)
```

Auf die theoretische Seite der Modifikationen wollen wir an dieser Stelle nicht näher eingehen, sondern verweisen auf [20], [12] und [13].

Auf der praktischen Seite haben diese relativ einfachen Modifikationen des Algorithmus von Blankinship großen Einfluß auf die Laufzeit und auf die Größe der Einträge in der Transformationsmatrix. Dies kann man aus der folgenden Tabelle ersehen, in welcher die maximalen Zwischenergebnisse bei der SNF–Berechnung ausgewählter Matrizen aufgeführt sind. Dabei betrachten wir den zu Beginn des Teil *Ein Algorithmus zur Berechnung der Smith–Normalform* vorgestellten Algorithmus zur SNF–Berechnung einerseits mit der Variante des Algorithmus von Blankinship als mgcd–Berechnung und andererseits mit der Modifikation von Havas und Majewski.

Als Matrizen haben wir einmal die Matrix R1 aus [15] und zum anderen eine zufällig generierte (20×20)–Matrix R2 mit Einträgen betragsmäßig kleiner als 10^8 gewählt. Die Matrix R1 haben wir im Anhang aufgeführt.

Matrix	Blankinship	Havas und Majewski
R1	$\approx 10^{971}$	26453
R2	$\approx 10^{4203}$	$\approx 10^{62}$

Tabelle 11.2.1: Zwischenergebnisse – Smith–Normalform–Berechnung

Zusammenfassend wird deutlich, daß in Algorithmen, die auf der Berechnung eines größten gemeinsamen Teilers einer Menge von ganzen Zahlen beruhen, der Einfluß der gewählten mgcd–Berechnung recht groß ist. Deshalb kommen wir zu dem Schluß, daß Forschungen auf dem Gebiet der ggT–Berechnung von großer Bedeutung für eine Anzahl von Algorithmen sein werden.

11.3 Pivotierung

Widmen wir uns nun dem zweiten oben angesprochenen Ansatzpunkt zur Effizienzsteigerung und zur Kontrolle des Anwachsens der Zwischenergebnisse. Die Auswahl eines Pivotelementes wird durch den Bereich, in dem wir nach dem Pivotelement suchen können, und durch ein Kriterium, das uns sagt, ob ein Element der Matrix als Pivot geeignet ist, beinflußt.

11.3.1 Pivotbereich

Bei der Auswahl des Suchbereiches für die Pivotierung müssen wir folgende zwei Aspekte in Betracht ziehen:

- Je mehr Elemente wir zur Auswahl haben, desto größer ist die Wahrscheinlichkeit, ein Element mit guten Pivoteigenschaften zu finden.
- Je mehr Elemente wir zur Auswahl in Betracht ziehen, desto größer ist der Zeitbedarf für die Pivotierung.

Den „Trade of" muß man in Abhängigkeit von den speziell vorliegenden Problemen untersuchen. Bei den praktischen Tests, die wir durchgeführt haben, hat sich für uns aus dem Zeitaspekt keine Notwendigkeit einer Beschränkung des Pivotbereiches ergeben. Deshalb wird bei allen Funktionen, die die Pivotierung verwenden, der maximale Pivotbereich durchsucht.

Für die HNF–Berechnung umfaßt der Pivotbereich die Elemente links von der aktuellen Arbeitsposition in der gleichen Zeile. Für die SNF–Berechnung umfaßt der Pivotbereich die rechte untere Teilmatrix, einschließlich der aktuellen Arbeitszeile und Arbeitsspalte.

Es stellt sich die Frage, was gute Pivoteigenschaften sind bzw. was das Kriterium ist, das uns ein Element als Pivotelement attraktiv macht.

11.3.2 Pivotkriterium

Betrachtet man die Pivotwahl aus dem Blickwinkel der Laufzeit, ist die „1" von besonderer Bedeutung. Gelingt es uns, als Pivot eine „1" zu finden, ist die mgcd–Berechnung sowohl im Algorithmus zur SNF–, als auch im Algorithmus zur HNF–Berechnung besonders einfach, und die Phase mgcd–Berechnung und Elimination kann extrem schnell ausgeführt werden. Dementsprechend lautet unsere erste Pivotstrategie:

$PIVOT_1$	Wähle als Pivotelement das betragsmäßig kleinste Element ungleich Null !

Ungeklärt bei dieser Strategie bleibt, was wir tun, wenn mehrere Elemente der Matrix den gleichen Betrag haben und dieser Wert das Minimum aller Beträge darstellt.

Betrachten wir nun das Problem der Pivotwahl aus dem Blickwinkel der Bekämpfung der *entry explosion*. In der mgcd–Berechnung ist, wie man sich leicht an den eben vorgestellten Algorithmen klarmachen kann, implizit eine Eliminationphase enthalten.

In dieser Eliminationsphase wird ein der Darstellung des mgcd entsprechendes Vielfaches der Arbeitsspalte zu den anderen Spalten addiert, um die Elemente in der Arbeitszeile zu Null zu machen. Für uns bedeutet dies, daß wir demjenigen Element den Vorzug geben sollten, das zu dieser Addition am wenigsten beiträgt, d.h. dessen Spaltennorm gemäß einer vorgegebenen Norm die kleinste unter den beteiligten Elementen ist.

Für die SNF–Berechnung erweitert sich dieses Schema. Während wir bei der HNF–Berechnung lediglich mit der Elimination von Einträgen innerhalb der Arbeitszeile zu tun haben, müssen wir bei der SNF–Berechnung in gleichem Maße Elemente der Arbeitsspalte eliminieren. Aus diesem Grund fließt in das Pivotkriterium bei der SNF–Berechnung zusätzlich eine Zeilennorm ein.
Als Spalten– und Zeilennormen verwenden wir die L_0 Metrik bzw. die L_k Norm ($k > 0$). Für $k = 0$ zählt diese Metrik die Einträge ungleich Null der betrachteten Zeile bzw. Spalte. Für $k = 1$ werden die Beträge der Elemente der jeweiligen Zeile bzw. Spalte aufaddiert und für $k = 2$ haben wir die euklidische Norm.

Bei der SNF–Pivotierung stehen wir nun vor der Frage, wie wir Zeilennorm und Spaltennorm miteinander verbinden können, um dadurch die Anzahl der möglichen Pivotelemente zu verkleinern. Zu diesem Zweck bieten sich der Einfachheit halber folgende Varianten für ein Element $a_{i,j} \in \mathbb{Z}^{m,n}$ und die L_k–Norm (Metrik) an:

1. $L_k(a_{*,j}) + L_k(a_{i,*}) = |a_{i,j}|_k^R + |a_{i,j}|_k^C$ (Addition)
2. $L_k(a_{*,j}) \cdot L_k(a_{i,*}) = |a_{i,j}|_k^R \cdot |a_{i,j}|_k^C$ (Multiplikation)

Als Pivotkriterien erhalten wir nach dieser Betrachtung:

HNF:	Wähle als Pivot dasjenige Element, dessen Spaltennorm minimal ist !
SNF:	Wähle als Pivot dasjenige Element, für das das Produkt bzw. die Summe der jeweiligen Spalten– und Zeilennorm minimal ist !

Wir haben lediglich der Vollständigkeit halber eine Formulierung für die HNF–Berechnung angegeben. Man erkennt jedoch, daß dieses Kriterium im Kriterium für die SNF–Berechnung enthalten ist. Der Pivotbereich der HNF beschränkt sich lediglich auf die Elemente links von der aktuellen Arbeitsposition in der gleichen Zeile. Somit ist die Zeilennorm für alle betrachteten Elemente identisch.

Wir haben somit für die HNF– und für die SNF–Berechnung zwei Kriterien, nach denen wir das Pivotelement auswählen können. Bei beiden wissen wir nicht, was bei dem Auftreten mehrerer Elemente, die die Pivoteigenschaft erfüllen, geschehen soll. Was liegt also näher, als die verschiedenen Pivoteigenschaften zu einer zu kombinieren. Wir erhalten dadurch folgende Kombinationskriterien:

$PIVOT_2$	Als Pivotelement wählen wir das Element aus dem Pivotbereich, das betragsmäßig den kleinsten Betrag besitzt. Falls mehrere solche Elemente vorhanden sind, wählen wir dasjenige, für das das Produkt bzw. die Summe der jeweiligen Spalten– und Zeilennorm minimalen ist.
$PIVOT_3$	Als Pivotelement wählen wir das Element aus dem Pivotbereich, für das das Produkt bzw. die Summe der jeweiligen Spalten– und Zeilennorm minimal ist. Falls mehrere solche Elemente vorhanden sind, wählen wir das betragsmäßig kleinste aus.

Bei realen Testläufen zur Hermite–Normalform–Berechnung mit dichtbesetzten Matrizen hat sich herausgestellt, daß trotz dieser unterschiedlichen Pivotstrategien ($PIVOT_1$, $PIVOT_2$, $PIVOT_3$) in der Regel dasselbe Element der Matrix als Pivotelement ausgewählt wird. Auf den ersten Blick ist dies erstaunlich, auf den zweiten Blick aber dadurch erklärbar, daß ein betragsmäßig minimales Element zur Spaltennorm und damit zur Kombination der Spalten– und Zeilennorm am wenigsten beiträgt. (Beachte: Die Zeilennorm ist für alle Elemente des Pivotbereiches gleich!)

Diese Erkenntnis hat zur Folge, daß sich Pivotierungen mittels der zuletzt vorgestellten Kombinationskriterien nur bei der SNF–Berechnung auszahlt. Bei der HNF–Berechnung ist die Suche nach dem betragsmäßig kleinsten Element im aktuellen Pivotbereich effizienter und führt nach unserer obigen Erkenntnis zu den gleichen Resultaten. Dies erklärt die Funktionsvielfalt im Abschnitt *Smith–Normalform* im Vergleich zum Abschnitt *Hermite–Normalform*.

Havas und Majewski haben in [12] andere Pivotierungsmöglichkeiten vorgestellt. Bei unseren Testläufen haben sich aber keine nennenswerten Veränderungen bzw. Verbesserungen ergeben. Aus diesem Grund haben wir auf diese Konstrukte bei der Zusammenstellung der Klasse *bigint_matrix* verzichtet.

Zum Abschluß stellen wir im Rest dieses Kapitels Funktionen zur SNF– und HNF–Berechnung vor, die zumindest eine der hier angeführten Ideen beinhalten.

11.4 Hermite–Normalform

Betrachten wir zunächst die in der Klasse *bigint_matrix* enthaltenen nichtmodularen Algorithmen zur Berechnung der Hermite–Normalform.

Zielsetzung der linearen Algebra:

Berechne die Hermite–Normalform der Matrix A bzw. berechne die Hermite–Normalform und eine Transformationsmatrix T, so daß $A \cdot T =$HNF(A)!

Prototypen:

Die Prototypen unterteilen sich in zwei Gruppen:

- Die Funktionen
  - `void A.hnf_simple()`
  - `void hnf_simple(bigint_matrix &A)`
  - `void A.hnf_havas()`
  - `void hnf_havas(bigint_matrix &A)`

  wandeln die Matrix A in Hermite–Normalform um.

- Die folgenden Funktionen berechnen zusätzlich eine unimodulare Transformationsmatrix T, so daß HNF(A)= $A \cdot T$.
  - `void A.hnf_simple(bigint_matrix &T)`
  - `void hnf_simple(bigint_matrix &A, bigint_matrix &T)`
  - `void A.hnf_havas(bigint_matrix &T)`
  - `void hnf_havas(bigint_matrix &A, bigint_matrix &T)`

Implementierung:

- Betrachten wir zuerst den oberen Block von Funktionen.
  Sie wandeln lediglich die Eingabematrix in Hermite–Normalform um. Die Funktionen mit dem Funktionsnamen *hnf_simple* benutzen zur Berechnung der HNF den Algorithmus, den wir im Abschnitt „*Ein Algorithmus zur Berechnung der Hermite–Normalform*“ als Ausgangsbasis vorgestellt haben. Die Funktionen mit dem Namen *hnf_havas* beinhalten alle im Zusammenhang mit der HNF–Berechnung vorgestellten Verbesserungen. Die Berechnung des größten gemeinsamen Teilers nach der Modifikation von Havas und Majewski haben wir aus Laufzeitaspekten in dieser Funktion fest verankert.

  Die Berechnung der Hermite–Normalform bricht in beiden Funktionstypen zu dem Zeitpunkt ab, an dem der Algorithmus feststellt, daß die Matrix nicht den vollen Zeilenrang besitzt.

- Die Funktionen der zweiten Gruppe arbeiten analog zu denen der ersten Gruppe. Allerdings werden alle Operationen, die auf der Eingabematrix A ausgeführt werden, auch auf einer Einheitsmatrix ausgeführt. Dazu initialisieren wir T mit der Einheitsmatrix. Zunächst überprüfen wir, ob die Matrix T eine quadratische Matrix der Dimension $A.columns$ ist. Ist dies nicht der Fall, wird sie mittels der Funktionen *set_no_of_rows* bzw. *set_no_of_columns* in eine solche verwandelt. Anschließend wird die Matrix T so mit Einsen und Nullen belegt, daß eine Einheitsmatrix entsteht. Damit haben wir die benötigte Einheitsmatrix und können wie oben angegeben fortfahren. Auf diese Art erhalten wir auf einem einfachen Weg die Transformationsmatrix.

Laufzeitvergleich:

Wir geben an dieser Stelle einen Laufzeitvergleich zwischen allen in LiDIA enthaltenen nichtmodularen Funktionen und den nichtmodularen Funktionen der Systeme PARI und MAPLE zur HNF–Berechnung an. MATHEMATICA besitzt unseres Wissens nach keine Funktion zur Hermite–Normalform–Berechnung.

Dimension	hnf_simple	hnf_havas	PARI	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:41	00:00:22	00:04:28	00:03:08
10×30	00:02:95	00:00:90	05:14:75	00:11:28
10×50	00:07:06	00:01:49	10:17:10	00:35:73
10×70	00:12:90	00:01:96	13:35:65	00:41:63
10×90	00:20:35	00:02:57	03:53:84	00:48:05
20×20	00:05:13	00:02:99	–	55:19:25
20×40	00:22:03	00:07:65	–	–
20×60	00:47:78	00:13:69	–	–
20×80	01:19:11	00:16:31	–	–
20×100	02:18:90	00:20:79	–	–
30×30	00:31:91	00:18:52	–	–
30×50	02:09:60	00:39:96	–	–
30×70	04:39:56	00:58:48	–	–
30×90	08:00:58	01:22:48	–	–

Tabelle 11.4.1: Laufzeitvergleich – Berechnung der Hermite–Normalform

Das Symbol „–“ in obiger Tabelle bedeutet, daß die Berechnung länger als eine Stunde benötigt, oder der Speicherbedarf 16 MB überschritten hat.

Desweiteren betrachten wir die Laufzeiten der Funktionen der Klasse *bigint_matrix*, die uns die Transformationsmatrix zur HNF–Berechnung liefern.

Dimension	hnfmod_mueller (min:sec:hsec)	hnf_simple (min:sec:hsec)	hnf_havas (min:sec:hsec)
10×10	00:01:17	00:00:60	00:00:51
10×30	00:12:28	00:07:12	00:03:77
10×50	00:46:69	00:22:43	00:09:01
10×70	02:02:70	00:50:99	00:15:66
10×90	04:08:60	01:28:77	00:23:02
20×20	00:17:87	00:11:21	00:09:85
20×40	01:13:67	01:05:10	00:36:51
20×60	03:11:29	03:19:99	01:22:90
20×80	06:27:80	06:50:94	02:21:51
20×100	11:45:46	12:29:21	04:36:31
30×30	01:48:79	01:35:80	01:19:11
30×50	05:23:74	07:04:58	03:57:65
30×70	10:06:37	20:04:95	07:48:88
30×90	17:15:40	41:19:01	14:36:08

Tabelle 11.4.1: Laufzeitvergleich – Berechnung der Hermite–Normalform

Bemerkung:

In beiden vorangehenden Tabellen erkennt man deutlich die Laufzeitverbesserung der Havas–Majewski–Modifikation im Verhältnis zum Ausgangsalgorithmus. Desweiteren wird deutlich, daß zur Berechnung der Transformtionsmatrix die Routine *hnf_havas* am besten ist.

11.5 Smith–Normalform

Zielsetzung der linearen Algebra:

Berechne die Smith–Normalform der Matrix A bzw. berechne die Smith–Normalform und die zugehörigen Transformationsmatrizen L und R, so daß $L \cdot A \cdot R = SNF(A)$!

Prototypen:

- Die Funktionen
  - `void A.snf_hartley()`
  - `void snf_hartley(bigint_matrix &A)`
  - `void A.snf_simple()`
  - `void snf_simple(bigint_matrix &A)`
  - `void A.snf_havas()`
  - `void snf_havas(bigint_matrix &A)`

- `void A.snf_mult(long art = 1)`
- `void snf_mult(bigint_matrix &A, long art = 1)`
- `void A.snf_add(long art = 1)`
- `void snf_add(bigint_matrix &A, long art = 1)`
- `void A.snf_new(long art = 1)`
- `void snf_new(bigint_matrix &A, long art = 1)`

transformieren die Matrix A auf Smith–Normalform.

- Die folgenden Funktionen wandeln die Matrix A in Smith–Normalform um und speichern in den Variablen L und R vom Typ *bigint_matrix* die zugehörigen Transformationsmatrizen, so daß $L \cdot A \cdot R =$SNF(A).

  - `void A.snf_hartley(bigint_matrix &L, bigint_matrix &R)`
  - ```
    void snf_hartley( bigint_matrix &A, bigint_matrix &L,
                      bigint_matrix &R )
    ```
  - `void A.snf_simple(bigint_matrix &L, bigint_matrix &R)`
  - ```
    void snf_simple( bigint_matrix &A, bigint_matrix &L,
                     bigint_matrix &R )
    ```
  - `void A.snf_havas(bigint_matrix &L, bigint_matrix &R)`
  - ```
    void snf_havas( bigint_matrix &A, bigint_matrix &L,
                    bigint_matrix &R )
    ```
  - ```
    void A.snf_mult( bigint_matrix &L, bigint_matrix &R,
                     long art = 1 )
    ```
  - ```
    void snf_mult( bigint_matrix &A, bigint_matrix &L,
                   bigint_matrix &R, long art = 1 )
    ```
  - ```
    void A.snf_add( bigint_matrix &L, bigint_matrix &R,
                    long art = 1 )
    ```
  - ```
    void snf_add( bigint_matrix &A, bigint_matrix &L,
                  bigint_matrix &R, long art = 1 )
    ```
  - ```
    void A.snf_new( bigint_matrix &L, bigint_matrix &R,
                    long art = 1 )
    ```
  - ```
    void snf_new( bigint_matrix &A, bigint_matrix &L,
                  bigint_matrix &R, long art = 1 )
    ```
    ```

Implementierung:

- Um auf eine einfache Art darzustellen, welche Algorithmen hinter den Funktionen der ersten Gruppe stehen, geben wir zuerst den Basisalgorithmus an, der allen Funktionen dieser Gruppe zugrundeliegt. Anschließend stellen wir in einer Tabelle lediglich Abweichungen vom Basisalgorithmus zusammen.

  Der Basisalgorithmus, der allen obigen Funktionen zugrundeliegt, ist der Algorithmus aus dem Abschnitt „*Ein Algorithmus zur Berechnung der Smith–Normalform*". Anhand der folgenden Tabelle verdeutlichen wir die jeweiligen, zu einer Funktion gehörenden Änderungen. Dabei verwenden wir die folgenden, abkürzenden Schreibweisen:

  - Die mgcd–Berechnung mit einer Variante des Algorithmus von Blankinship bezeichnen wir mit *Blank*.
  - Die mgcd–Berechnung mit der Modifikation von Havas und Majewski bezeichnen wir mit *Havas*.

Funktionsname	mgcd–Algorithmus	Pivotierung
snf_hartley	Blank, implizit	$PIVOT_1$
snf_simple	Havas, explizit	$PIVOT_1$
snf_havas	Havas, implizit	$PIVOT_1$
snf_mult	Havas, implizit	$PIVOT_2$, Multiplikation
snf_add	Havas, implizit	$PIVOT_2$, Addition
snf_new	Havas, implizit	$PIVOT_3$, Multiplikation

Tabelle 11.5.1: Zusammenstellung der SNF–Funktionen

  Die Bezeichnungen *implizit* bzw. *explizit* in der Spalte mgcd–Algorithmus geben an, ob der jeweilige Algorithmus fest in der betreffenden Funktion implementiert wurde, oder explizit als Funktion vorhanden ist.

  Die Angaben *Addition* bzw. *Multiplikation* in der Spalte Pivotierung bezeichnen die arithmetische Operation, die in dem angegebenen Pivotkriterium verwendet wurde, um Spalten– und Zeilennorm zu verknüpfen.

  Die Funktionen *snf_add*, *snf_mult* und *snf_new* besitzen, wie man an den Prototypen leicht erkennen kann, zusätzlich den Parameter *art*. Dieser Parameter gibt an, daß die L_{art}–Norm zur Berechnung der Zeilen– bzw. Spaltennorm innerhalb der jeweilig benutzten Pivotierung verwendet werden soll.

- Die Funktionen der zweiten, oben angegebenen Gruppe beinhalten den gleichen Algorithmus wie die namensgleichen Funktionen der ersten Gruppe. Der Unterschied besteht darin, daß als Nebenprodukt des jeweiligen Algorithmus hier die Transformationsmatrizen L und R entstehen.

Dazu wird zunächst überprüft, ob die Matrizen L und R die richtigen Dimensionen besitzen, d.h. $L.rows = L.columns = A.rows$ und $R.rows = R.columns = A.columns$. Ist dies nicht der Fall, werden sie auf die jeweilige Dimension mittels der Funktionen *set_no_of_rows* oder *set_no_of_columns* gebracht. Anschließend werden sie mit Nullen und Einsen belegt, so daß jeweils eine Einheitsmatrix entsteht. Nun wird der eigentliche SNF–Algorithmus (gemäß obiger Tabelle) ausgeführt. Dabei werden alle Zeilenoperationen ebenfalls auf der Matrix L und alle Spaltenoperationen auf der Matrix R ausgeführt. Nach der Terminierung des Algorithmus beinhalten R und L die gewünschten Transformationsmatrizen.

Laufzeitvergleich:

In den beiden folgenden Tabellen stellen wir die Funktionen der Klasse *bigint_matrix* zur Smith–Normalform–Berechnung denen der Systeme MAPLE und PARI gegenüber. Dabei klammern wir die Funktion *snf_hartley* wegen zu langer Laufzeiten aus. In dem System MATHEMATICA haben wir keine vergleichbare Funktion zur SNF–Berechnung gefunden.

Dimension	snf_havas	snf_mult	snf_add	snf_simple
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:33	00:00:37	00:00:35	00:00:56
10×40	00:01:24	00:01:59	00:01:57	00:05:07
10×70	00:02:43	00:02:81	00:02:87	00:15:08
10×100	00:03:21	00:03:85	00:03:81	00:25:71
20×20	00:02:97	00:03:36	00:03:31	00:05:64
20×50	00:09:98	00:11:08	00:11:06	00:34:62
20×80	00:16:13	00:20:17	00:20:21	01:19:61
30×30	00:14:25	00:15:27	00:16:00	00:29:14
30×60	00:40:16	00:45:64	00:45:27	02:23:93
30×90	01:08:72	01:16:86	01:16:53	05:21:93
40×40	00:45:13	00:49:06	00:50:50	01:40:66
40×70	02:06:29	02:18:10	02:18:76	07:51:81
40×100	03:33:14	03:41:99	03:48:94	17:56:53
50×50	02:06:68	02:15:55	02:14:82	05:06:68
50×80	05:49:90	06:18:93	06:20:17	24:02:88
60×60	05:39:07	05:54:72	06:00:10	14:17:45

Tabelle 11.5.2: Laufzeitvergleich – Berechnung der Smith–Normalform

Dimension	snf_new	PARI	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:41	00:00:12	00:02:75
10×40	00:01:91	–	00:05:20
10×70	00:03:15	–	00:14:48
10×100	00:04:44	–	00:21:20
20×20	00:04:02	00:03:87	00:51:70
20×50	00:13:70	–	00:74:13
20×80	00:23:29	–	01:57:85
30×30	00:17:96	25:44:43	06:35:55
30×60	00:55:48	–	04:47:03
30×90	01:32:15	–	08:09:53
40×40	00:59:59	–	25:43:00
40×70	02:49:09	–	–
40×100	04:41:97	–	–
50×50	02:44:02	–	–
50×80	07:28:68	–	–
60×60	07:04:40	–	–

Tabelle 11.5.2: Laufzeitvergleich – Berechnung der Smith–Normalform

PARI unterstützt mit seiner Funktion lediglich die Smith–Normalform–Berechnung für quadratische Matrizen. Sonstige mit „–“ gekennzeichnete Berechnungen wurden entweder zu dem Zeitpunkt abgebrochen, als der Speicherbedarf mehr als 16 MB betrug, oder die Laufzeit eine Stunde überschritten hat.

Kapitel 12

Weitere Funktionen der linearen Algebra über $\mathbb{Z}$

In diesem Kapitel stellen wir Funktionen der linearen Algebra über $\mathbb{Z}$ vor, die sich entweder durch Kombinationen der in den vorherigen Kapiteln erläuterten Funktionen berechnen lassen oder in keines der bisher vorgegebenen Schemata passen.

Alle Algorithmen, die den folgenden Funktionen zugrundeliegen, stammen aus der Diplomarbeit „*Effiziente Algorithmen für Probleme der linearen Algebra über* $\mathbb{Z}$" von Achim Müller und wurden nur unwesentlich verändert. Deshalb werden wir die Implementierung der jeweils betrachteten Funktionen nur andeuten oder uns mit einem Verweis auf die entsprechende Stelle in der Arbeit von Achim Müller und einer Aufzählung der verwendeten Funktionen begnügen.

12.1 Diagonalmatrix, transponierte Matrix

Für viele Berechnungen der linearen Algebra über $\mathbb{Z}$ benötigt man die Einheitsmatrix, oder allgemein eine Matrix A, bei der die Einträge mit gleichem Spalten– und Zeilenindex mit a und alle anderen mit b belegt sind. Ebenso ist es oft von Interesse, eine Matrix A zu transponieren. Deshalb existieren in der Klasse *bigint_matrix* die folgenden Funktionen:

Prototypen:

- Die Funktionen
  - `void A.diag(const bigint & a, const bigint & b)`
  - ```
    void diag( bigint_matrix & A, const bigint & a,
               const bigint & b )
    ```
    ```

belegen die Einträge mit gleichem Spalten– und Zeilenindex der Matrix A mit a und die übrigen Einträge mit b.

- Die Funktionen
  - `bigint_matrix A.trans() const`
  - `bigint_matrix trans(const bigint_matrix & A)`

  geben A^T zurück.

Implementierung:

- Die Funktionen *A.diag(a,b)* und *diag(A,a,b)* durchlaufen zeilenweise das Wertearray der Matrix A. Dabei schreiben sie a an alle Positionen (i,i) mit $0 \leq i < \min(A.rows, A.columns)$ und b an alle anderen Positionen.
- Die Funktionen *A.trans()* und *trans(A)* erzeugen zuerst eine Variable TMP vom Typ *bigint_matrix* mit $A.columns$ Zeilen und $A.rows$ Spalten. Dann werden die Werte $A.value[i][j]$ nach $TMP.value[j][i]$ für $0 \leq i < A.rows$ und $0 \leq j < A.columns$ kopiert. Zuletzt wird die Matrix TMP als Funktionswert zurückgegeben.

Laufzeitvergleich:

In der folgenden Tabelle betrachten wir die Laufzeit der Transponiertenbildung. Die Laufzeit der Diagonalfunktion erscheint uns von geringerer Bedeutung zu sein und wird deshalb nicht weiter berücksichtigt.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
50×50	00:00:09	00:00:02	00:00:03	00:01:06
50×100	00:00:21	00:00:02	00:00:06	00:01:95
100×100	00:00:35	00:00:05	00:00:16	00:03:70
100×200	00:01:03	00:00:09	00:00:32	00:35:13
200×200	00:01:40	00:00:18	00:00:68	–
200×300	00:03:47	00:00:24	00:00:98	–
300×300	00:03:18	00:00:46	00:01:45	–

Tabelle 12.1.1: Laufzeitvergleich – Berechnung einer transponierten Matrix

12.2 Kernberechung

Eine der Hauptaufgaben der linearen Algebra ist die Bestimmung einer Basis des Kerns einer linearen Abbildung ϕ mit Abbildungsmatrix A. Folgende Funktionen berechnen eine solche Basis.

Prototypen:

- `void B.kernel(const bigint_matrix &A)`
  berechnet eine Basis des Kerns der zu A gehörenden linearen Abbildung und speichert die Basisvektoren als Spalten in der Matrix B.

- `bigint_matrix kernel(const bigint_matrix &A)`
  berechnet eine Basis des Kerns der zu A gehörenden linearen Abbildung, speichert die Basisvektoren als Spalten in einer Matrix und gibt diese zurück.

Implementierung:

- Der Algorithmus, der der Funktion *B.kernel(A)* zugrunde liegt, wird detailliert in [21] (Seite 149) beschrieben. Deshalb werden wir an dieser Stelle nur kurz auf die Implementierung eingehen.
  Zuerst berechnen wir den Rang r und die Indizes $i_1, \cdots, i_r$ der linear unabhängigen Zeilen der Matrix A mittels der Funktion *lininr*. Dann erzeugen wir eine temporäre $(r \times A.columns)$–Matrix TMP, die die Werte der Zeilen $i_1, \cdots, i_r$ erhält. Von dieser Matrix berechnen wir die Hermite–Normalform inklusive der Transformationsmatrix. Hierbei benutzen wir die Matrix B zur Speicherung der Transformationsmatrix. Mittels der Funktion *set_no_of_columns* transformieren wir die Matrix B in eine $(A.columns \times (A-columns - r))$–Matrix und beenden die Funktion.

- Die Funktion *kernel(A)* ist lediglich eine Benutzerschnittstelle zu der eben beschriebenen Funktion.

Laufzeitvergleich:

Aufgrund dessen, daß die Systeme PARI, MATHEMATICA und MAPLE den Kern über $\mathbb{Q}$ und nicht über $\mathbb{Z}$ berechnen ist ein Laufzeitvergleich mit diesen Systemen nicht sinnvoll. Wir geben deshalb lediglich die Laufzeiten der *bigint_matrix*–Funktionen an.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:04	–	–	–
10×20	00:01:05	–	–	–
20×20	00:00:46	–	–	–
20×40	00:18:51	–	–	–
30×30	00:02:06	–	–	–
30×60	02:29:62	–	–	–
40×40	00:06:10	–	–	–
40×80	12:27:67	–	–	–
50×50	00:14:60	–	–	–

Tabelle 12.2.1: Laufzeitvergleich – Berechnung des Kerns

12.3 Bildberechnung

Eine weitere wichtige Aufgabe ist die Berechnung einer Basis des Bildes einer linearen Abbildung ϕ mit Abbildungsmatrix A. Zur Lösung dieser Aufgabe enthät die Klasse *bigint_matrix* folgende Funktionen.

Prototypen:

- `void B.image(const bigint_matrix &A)`
  berechnet eine Basis des Bildes der zu A gehörenden linearen Abbildung und speichert die Basisvektoren als Spalten in der Matrix B.

- `bigint_matrix image(const bigint_matrix &A)`
  berechnet eine Basis des Bildes der zu A gehörenden linearen Abbildung, speichert die Basisvektoren als Spalten in einer temporären Matrix und gibt diese als Funktionswert zurück.

Implementierung:

- Zur Berechnung des Bildes einer linearen Abbildung mit Abbildungsmatrix A wurde der Algorithmus aus [21] (Seite 151) in der Funktion *B.image(B)* implementiert. Ziel ist es, die Matrix A auf Spalten–Stufen–Form zu bringen und führende Nullspalten zu verwerfen. Die wesentlichen Bausteine dabei sind *lininr* und eine Funktion zur Berechnung der Hermite–Normalform inklusive der Transformationsmatrix. Auf eine weitere Beschreibung verzichten wir an dieser Stelle, weil sie entweder zu umfangreich oder zu unverständlich ausfallen würde. Wir verweisen auf die oben erwähnte Stelle in der Diplomarbeit von Achim Müller.

- Die Funktion *image(A)* ist eine Benutzerschnittstelle zu der oben angeführten Funktion.

Laufzeitvergleich:

Wie bereits bei der Kernberechnung angesprochen wurde, berechnen die Systeme PARI, MATHEMATICA und MAPLE auch das Bild einer linearen Abbildung über $\mathbb{Q}$. Aus diesem Grund wäre eine Laufzeitvergleich unfair. Deshalb geben wir an dieser Stelle lediglich die Laufzeiten der *bigint_matrix*–Funktionen an.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:12	–	–	–
10×20	00:00:29	–	–	–
20×20	00:01:53	–	–	–
20×40	00:03:70	–	–	–
30×30	00:08:79	–	–	–
30×60	00:23:09	–	–	–
40×40	00:37:20	–	–	–
40×80	01:36:71	–	–	–
50×50	02:02:71	–	–	–

Tabelle 12.3.1: Laufzeitvergleich – Berechnung des Bildes

12.4 Urbildberechnung und das Lösen linearer Gleichungssysteme

Die Berechnung einer Lösung eines linearen Gleichungssystems $A \cdot x = b$ bzw. die Berechnung des Urbildes von b der linearen Abbildung ϕ mit Abbildungmatrix A ist wohl die bekannteste Zielsetzung der linearen Algebra. Die Klasse *bigint_matrix* bietet dazu folgende Funktionen an.

Prototypen:

- Die Funktionen
  - `void B.invimage(const bigint_matrix &A, const bigint *v)`
  - `void B.solve(const bigint_matrix &A, const bigint *v)`

  berechnen eine spezielle Lösung des linearen Gleichungssystems $A \cdot x = b$ und eine Basis des Kerns der zu A gehörenden linearen Abbildung und speichern die spezielle Lösung und die Basis des Kerns in der Matrix B. Dann folgt:

1. $b_{*,0}, \ldots, b_{*,columns-2}$ = Basis des Kerns der zu A gehörenden linearen Abbildung.
2. $b_{*,B.columns-1}$= spezielle Lösung des Gleichungssystems $A \cdot x = b$

Findet der Algorithmus keine spezielle Lösung, besteht die Lösungsmatrix B aus genau einer mit Null belegten Spalte.

- Die Funktionen
  - `bigint_matrix invimage( const bigint_matrix &A,`
    `const bigint *v )`
  - `bigint_matrix solve(const bigint_matrix &A, const bigint *v)`

  berechnen eine spezielle Lösung des linearen Gleichungssystems $A \cdot x = b$ und eine Basis des Kerns der zu A gehörenden linearen Abbildung und speichern die spezielle Lösung und die Basis des Kerns in einer temporären Matrix. Die Wertebelegung der temporären Matrix entspricht der oben angegebenen Belegung der Matrix B. Diese temporäre Matrix geben die Funktionen dann als Funktionswert zurück.

Implementierung:

Hinter den Funktionen der ersten Gruppe verbirgt sich der eigentliche Algorithmus. Die Funktionen der zweiten Gruppe bilden lediglich eine Benutzerschnittstelle zu den Funktionen der ersten Gruppe. Implementiert wurde die Arbeitsanweisung aus [21] (Seite 157).

Laufzeitvergleich:

PARI, MATHEMATICA und MAPLE geben Funktionen an, die eine Lösung über $\mathbb{Q}$ und nicht über $\mathbb{Z}$ berechnen. Deshalb werden wir, wie in den beiden vorangehenden Abschnitten auf einen Laufzeitvergleich verzichten und lediglich die Laufzeiten unserer Funktionen angeben.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:79	–	–	–
10×20	00:02:29	–	–	–
20×20	00:12:05	–	–	–
20×40	00:41:29	–	–	–
30×30	01:23:25	–	–	–
30×60	05:05:92	–	–	–
40×40	06:15:43	–	–	–
50×50	22:48:48	–	–	–

Tabelle 12.4.1: Laufzeitvergleich – Berechnung des Urbildes

12.5 Reguläres Urbild

Gegeben sei eine reguläre ganzzahlige $(n \times n)$–Matrix A und eine Folge von rechten Seiten $b(0), \ldots, b(m-1)$. Gesucht sind eine Folge von Vektoren $x(0), \ldots, x(m-1) \in \mathbb{Z}^n$ und eine Folge von ganzen Zahlen $g_0, \ldots, g_{m-1}$, so daß

$$\begin{array}{rcl} A \cdot x(0) & = & g_0 \cdot b(0) \\ A \cdot x(1) & = & g_1 \cdot b(1) \\ & & \\ \vdots & \vdots & \vdots \\ A \cdot x(m-1) & = & g_{m-1} \cdot b(m-1) \end{array}$$

Die Klasse *bigint_matrix* bietet zu dieser Aufgabenstellung die folgenden Funktionen an.

Prototypen:

Sei A die oben angegebene $(n \times n)$ Matrix und $B = (b(0), \ldots, b(m-1))$ die ganzzahlige $(n \times m)$–Matrix der rechten Seiten.

- Dann speichert die Funktion

  - `void X.reginvimage( const bigint_matrix & A,`
    `const bigint_matrix & B )`

  die Lösung des obigen Problems in der Matrix X, wobei

$$X = \begin{pmatrix} x(0)_0 & x(1)_0 & \cdots & x(m-1)_0 \\ x(0)_1 & x(1)_1 & \cdots & x(m-1)_1 \\ \vdots & \vdots & \vdots & \vdots \\ x(0)_{n-1} & x(1)_{n-1} & \cdots & x(m-1)_{n-1} \\ g_0 & g_1 & \cdots & g_{m-1} \end{pmatrix}$$

- Dagegen berechnet die Funktion

  - `bigint_matrix reginvimage( const bigint_matrix & A,`
    `const bigint_matrix & B )`

  eine Lösung obiger Art und gibt diese als Funktionswert zurück.

Implementierung:

- In der Funktion *X.reginvimage(A,B)* wurde der Algorithmus aus [21] (Seite 113) verwendet. Basisbausteine dieser Funktion sind *det*, *adj* und eine Routine zur Berechnung des größten gemeinsamen Teilers einer Menge von ganzen Zahlen.
- Die Funktion *reginvimage(A,B)* bildet, wie bereits oben angedeutet wurde, lediglich eine Benutzerschnittstelle zur Funktion *X.reginvimage(A,B)*.

Laufzeitvergleich:

Die Systeme PARI, MATHEMATICA und MAPLE bieten unseres Wissens keine Funktionen mit vergleichbarer Funktionalität an. Deshalb geben wir an dieser Stelle lediglich die Laufzeiten der LiDIA–Funktionen an. Die Laufzeiten beziehen sich auf die Ausführung der Funktion *X.reginvimage(A,B)*, wobei A und B quadratische Matrizen angegebener Dimensionen sind.

Dimension	LiDIA	PARI	MATHEMATICA	MAPLE
	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)	(min:sec:hsec)
10×10	00:00:85	–	–	–
20×20	00:09:07	–	–	–
30×30	00:39:31	–	–	–
40×40	01:57:68	–	–	–
50×50	04:43:70	–	–	–
60×60	09:38:65	–	–	–
70×70	19:11:43	–	–	–

Tabelle 12.5.1: Laufzeitvergleich – Berechnung des regulären Urbildes

12.6 Minimum, Maximum u.a.

Der Benutzer dieser Klasse wird sich bei verschiedenen Aufgabenstellungen dafür interessieren, wie groß der größte Eintrag, wie groß das betragsmäßig größte Element u.s.w. in einer Matrix ist. Wir stellen für solche Fragestellungen folgende Funktionen zur Verfügung.

Prototypen:

- **Maximum:**
  - Die Funktion

  - `void A.max(bigint & MAX) const`

  speichert den Wert des größten Eintrages der Matrix A in der Variablen MAX.

- Die Funkionen
  - `bigint A.max() const`
  - `bigint max(const bigint_matrix & A)`

  geben den Wert des größten Eintrages der Matrix A als Funktionswert zurück.

- **absolutes Maximum:**
  - Die Funktion
    - `void A.max_abs(bigint & MAX) const`

    speichert den Wert des betragsmäßig größten Eintrages der Matrix A in der Variablen MAX.
  - Die Funkionen
    - `bigint A.max_abs() const`
    - `bigint max_abs(const bigint_matrix & A)`

    geben den Wert des betragsmäßig größten Eintrages der Matrix A als Funktionswert zurück.

- **Minimum:**
  - Die Funktion
    - `void A.min(bigint & MIN) const`

    speichert den Wert des kleinsten Eintrages der Matrix A in der Variablen MIN.
  - Die Funkionen
    - `bigint A.min() const`
    - `bigint min(const bigint_matrix & A)`

    geben den Wert des kleinsten Eintrages der Matrix A als Funktionswert zurück.

- **absolutes Minimum:**
  - Die Funktion
    - `void A.min_abs(bigint & MIN) const`

    speichert den Wert des betragsmäßig kleinsten Eintrages der Matrix A in der Variablen MIN.
  - Die Funkionen
    - `bigint A.min_abs() const`

 * `bigint min_abs(const bigint_matrix & A)`

 geben den Wert des betragsmäßig kleinsten Eintrages der Matrix A als Funktionswert zurück.

- **k–te Potenz der L_k–Norm einer Zeile:**
  - Die Funktion
    * `void A.row_norm(bigint & NORM, long i, long k) const`

    berechnet $NORM = \sum_{l=0}^{A.columns-1} |a_{i,l}|^k$.
  - Die Funkionen
    * `bigint A.row_norm(long i, long k) const`
    * `bigint row_norm( const bigint_matrix & A, long i, long k )`

    geben den Wert $\sum_{l=0}^{A.columns-1} |a_{i,l}|^k$ als Funktionswert zurück.

- **k–te Potenz der L_k–Norm einer Spalte:**
  - Die Funktion
    * `void A.column_norm(bigint & NORM, long i, long k) const`

    berechnet $NORM = \sum_{l=0}^{A.rows-1} |a_{l,i}|^k$.
  - Die Funkionen
    * `bigint A.column_norm(long i, long k) const`
    * `bigint column_norm( const bigint_matrix & A, long i, long k )`

    geben den Wert $\sum_{l=0}^{A.rows-1} |a_{l,i}|^k$ als Funktionswert zurück.

Implementierung:

- Zu den Funktionengruppen *Maximum*, *absolutes Maximum*, *Minimum* und *absolutes Minimum* ist nicht viel zu sagen. Die jeweiligen Funktionen durchlaufen das Wertearray der Matrix A und suchen den gewünschten Wert.

- Die Funktionen der Gruppe *Potenz der Zeilennorm* überprüfen zuerst, ob für den Parameter i die Bedingung $0 \leq i < A.rows$ gilt. Ist diese Bedingung verletzt, wird die Funktion mit der Fehlermeldung

  row_norm :: Error in parameter !

  beendet. Ist die Bedingung erfüllt, wird die oben erwähnte Berechnung durchgeführt.

- Die Funktionen der Gruppe *Potenz der Spaltennorm* überprüft zuerst, ob der Parameter i der Bedingung $0 \leq i < A.columns$ genügt. Ist diese Bedingung verletzt, wird die Funktion mit der Fehlermeldung

  column_norm :: Error in parameter !

  beendet. Ist die Bedingung erfüllt, wird die oben erwähnte Berechnung durchgeführt.

Laufzeitvergleich:

Wie bereits im vorangehenden Abschnitt fehlen auch zu diesen Funktionen entsprechende Äquivalente in den Systemen PARI, MATHEMATICA und MAPLE. In der folgenden Tabelle geben wir die Laufzeiten der Funktionen *max* und *row_norm* aus dem LiDIA–Paket an. Hierbei betrachten wir die euklidische Zeilennorm.

Dimension	*max*	*row_norm*
	(min:sec:hsec)	(min:sec:hsec)
100×100	00:00:05	00:00:01
200×200	00:00:16	00:00:01
300×300	00:00:37	00:00:02
400×400	00:00:66	00:00:03
500×500	00:01:01	00:00:04
600×600	00:01:48	00:00:04
700×700	00:02:84	00:00:05
800×800	00:05:38	00:00:05
900×900	00:09:24	00:00:06

Tabelle 12.6.1: Laufzeitvergleich – Maximum / Potenz der Zeilennorm

12.7 Zufallsmatrizen

In der vorliegenden Arbeit werden zu allen Laufzeitmessungen Zufallsmatrizen mit fester Vorgabe für die Größe der Einträge verwendet. Um auf eine einfache Art und Weise solche Matrizen zu erzeugen, wurden die folgenden Funktionen implementiert.

Prototypen:

- `void A.randomize(const bigint & S)`
  belegt die Einträge der Matrix A mit Zufallswerten im Bereich von 0 bis S.

- `bigint_matrix randomize(long i, long j, const bigint & S)`
  erzeugt eine $(i \times j)$–Matrix mit Zufallseinträgen im Bereich von 0 bis S.

Implementierung:

- Die Funktion *A.randomize(S)* durchläuft das Wertearray der Matrix A zeilenweise und überschreibt den aktuellen Wert durch einen zufällig erzeugten Wert im Bereich von 0 bis S. Als (Pseudo–)Zufallszahlengenerator wird *randomize* aus der Klasse *bigint* verwendet.

- Die Funktion *randomize(i,j,S)* erzeugt eine $(i \times j)$–Matrix TMP ruft anschließend *TMP.randomize(S)* auf und gibt die Matrix TMP als Funktionswert zurück.

Eine Laufzeitbetrachtung erscheint uns an dieser Stelle überflüssig.

Teil V

Eingabe, Ausgabe und Schnittstellen zu anderen Computer–Algebra–Systemen

Kapitel 13

Datenformate

Nachdem wir uns bisher mit Basisfunktionen, Funktionen der linearen Algebra, arithmetischen Operationen und vielem anderen auseinander gesetzt haben, stellen wir in diesem Kapitel die Funktionen zur Ein– und Ausgabe einer Variablen vom Typ *bigint_matrix* vor.

Entscheidend für den Aufbau dieser Funktionen ist dabei das einer Matrix zugrundeliegende Datenformat, in dem die Variable abgelegt bzw. gelesen wird. Desweiteren ist die Anzahl der unterstützten Formate ausschlaggebend für den Datenaustausch mit anderen Computer–Algebra–Systemen, was der Philosophie des LiDIA–Systems entspringt und was die Akzeptanz bei potentiellen Nutzern steigert.

Im folgenden stellen wir die Formate vor, die von der Matrixklasse *bigint_matrix* unterstützt werden. Als Beispiel nehmen wir an, daß die Matrix

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

in dem jeweiligen Format dargestellt werden soll.

- Matrix A im PARI–Format (GP–Format):

$$[a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}; a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}; a_{2,0}, a_{2,1}, a_{2,2}, a_{2,3}]$$

- Matrix A im MATHEMATICA–Format:

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

- Matrix A im MAPLE–Format:

$$
\begin{aligned}
array(1..4, 1..4, [(1,1) = a_{0,0}, (1,2) = a_{0,1}, (1,3) = a_{0,2}, (1,4) = a_{0,3},\\
(2,1) = a_{1,0}, (2,2) = a_{1,1}, (2,3) = a_{1,2}, (2,4) = a_{1,3},\\
(3,1) = a_{2,0}, (3,2) = a_{2,1}, (3,3) = a_{2,2}, (3,4) = a_{2,3}]);
\end{aligned}
$$

- Matrix A im LiDIA–Format:

$$
\begin{array}{r}
3\\
4\\
a_{0,0}\ a_{0,1}\ a_{0,2}\ a_{0,3}\\
a_{1,0}\ a_{1,1}\ a_{1,2}\ a_{1,3}\\
a_{2,0}\ a_{2,1}\ a_{2,2}\ a_{2,3}
\end{array}
$$

Es fällt auf, daß jedes der hier betrachteten Computer–Algebra–Systeme ein eigenes Datenformat für Matrizen verwendet. Der Nutzer mehrerer Systeme ist somit gezwungen, entweder von Hand oder durch ein Hilfsprogramm eine Formatkonvertierung vorzunehmen, wenn er eine Matrix von einem System in ein anderes übertragen will.

In LiDIA soll dieser Aufwand minimiert werden. Deshalb ist die Eingabe der Klasse *bigint_matrix* so konzipiert, daß das vorliegende Format automatisch erkannt wird, die Daten konvertiert und richtig eingelesen werden. Falls dem Nutzer das Ausgangsformat bekannt ist, kann er sich schnellerer Funktionen bedienen, die auf das jeweilige Datenformat zugeschnitten sind.

Zum besseren Verständnis dieser Konzeption unterscheiden wir im folgenden:

1. Ein- und Ausgabefunktionen ohne Datenformatbeschränkung und
2. Ein- und Ausgabefunktionen mit Datenformatbeschränkung.

Bevor wir in den nächsten beiden Abschnitten die Funktionen zur Ein– und Ausgabe detailliert vorstellen, gehen wir zunächst auf eine Frage ein, die sich dem Leser wohl aufdrängt. Wenn der Aufwand der Datenkonvertierung in LiDIA minimiert werden soll, weshalb gibt es dann ein LiDIA–Format als Datenformat einer Matrix?
Um diese Frage beantworten zu können, müssen wir zunächst aufzeigen, wie sich die anderen Formate auf Einlesefunktionen auswirken. Um eine Matrix korrekt einlesen zu können, müssen wir zunächst die Dimensionen der Matrix herausfinden. Dies ist beim *MATHEMATICA*– und *PARI*–Format mit einem Scannen des Eingabestreams verbunden und führt somit zu vermeidbaren Laufzeiteinbußen. Im *MAPLE*–Format ist die Dimensionserkennung, wie wir später noch sehen werden, einfach, aber das Einlesen der Matrixeinträge birgt einen großen Overhead, den wir vermeiden wollen.

Wir haben uns also aus Laufzeitaspekten für ein LiDIA–Format mit dem oben angegebenen Aussehen entschieden.

Bemerkung:

Die Einführung eines LiDIA–Formats stellt im Gegensatz zu vielen anderen Computer–Algebra–Systemen keine Einschränkung dar. Durch die Vielzahl der unterstützten Schnittstellen bleibt es stets dem Nutzer überlassen, welches Format er für seine Matrizen verwendet.

13.1 Ein- und Ausgabefunktionen mit Datenformatbeschränkung

Aufgrund dessen, daß die Funktionen zur Eingabe bzw. Ausgabe ohne Datenformatbeschränkung auf denen mit Datenformatbeschränkung aufbauen, gehen wir auf letztere zuerst ein.

Die im folgenden vorgestellten Funktionen sind auf ein spezielles Datenformat zugeschnitten. Somit lassen sich die folgenden Prototypen vier Bereichen zuordnen.

Prototypen:

1. LiDIA–**Format:**
   - `void A.write_to_stream(ostream &OUT)`
     schreibt eine Matrix A im LiDIA–Format auf den Stream *OUT*.
   - `void A.read_from_stream(istream &IN)`
     liest eine Matrix, die im LiDIA–Format vorliegt, vom Stream *IN* in eine Variable A vom Typ *bigint_matrix* ein.
2. **PARI–Format:**
   - `void A.write_to_gp(ostream &OUT)`
     schreibt eine Matrix A im PARI–Format auf den Stream *OUT*.
   - `void A.read_from_gp(istream &IN)`
     liest eine Matrix, die im PARI–Format vorliegt, vom Stream *IN* in eine Variable A vom Typ *bigint_matrix* ein.
3. **MATHEMATICA–Format:**
   - `void A.write_to_mathematica(ostream &OUT)`
     schreibt eine Matrix A im MATHEMATICA–Format auf den Stream *OUT*.
   - `void A.read_from_mathematica(istream &IN)`
     liest eine Matrix, die im MATHEMATICA–Format vorliegt, vom Stream *IN* in eine Variable A vom Typ *bigint_matrix* ein.

4. **MAPLE–Format:**

   - `void A.write_to_maple(ostream &OUT)`
     schreibt eine Matrix A im MAPLE–Format auf den Stream OUT.
   - `void A.read_from_maple(istream &IN)`
     liest eine Matrix, die im MAPLE–Format vorliegt, vom Stream *IN* in eine Variable A vom Typ *bigint_matrix* ein.

Implementierung:

Im folgenden beschreiben wir lediglich die Implementierung der Einlesefunktionen. Die Implementierung der Streamausgabefunktionen ergibt sich kanonisch aus den zugrundeliegenden Formaten, die wir zu Beginn dieses Kapitels vorgestellt haben. Außerdem setzen wir voraus, daß die einzulesende Matrix im jeweiligen Format korrekt auf dem Eingabestream steht. Die Fehlerbehandlung bei falschem Format werden wir etwas später betrachten.

- Die Funktion *A.read_from_stream(IN)* liest eine Matrix, die im LiDIA–Format auf dem Eingabestream *IN* steht, in die Variable A vom Typ *bigint_matrix* ein. Dazu werden zunächst zwei *long*–Werte vom Stream *IN* gelesen. Diese beiden Werte entsprechen der Anzahl der Zeilen und der Anzahl der Spalten der auf dem Stream liegenden Matrix. Die Matrix A wird nun mittels der Funktionen *set_no_of_rows* bzw. *set_no_of_columns* an die eingelesenen Dimensionen angepaßt. Anschließend werden $(A.rows \cdot A.columns)$ *bigint*–Werte vom Stream *IN* gelesen und zeilenweise in das Wertearray der Matrix A geschrieben.
  Die Korrektheit dieser Vorgehensweise wird durch die Formatcharakterisierung des LiDIA–Formats garantiert.

- Das Einlesen einer Matrix, die im PARI–Format auf einem Stream *IN* liegt, wird durch die Funktion *A.read_from_gp(IN)* realisiert.
  Zunächst muß die Funktion die Dimensionen der auf dem Stream stehenden Matrix ermitteln. Dazu durchsuchen wir den Eingabestream im Bereich von „[“ bis „]“ und ermitteln die Anzahl der Kommata und der Semikolons. Die Anzahl der dabei gefundenen Semikolons plus 1 entspricht der Anzahl der Zeilen und die Anzahl der Kommas zwischen der öffnenden Klammer „[“ und dem ersten Auftauchen eines Semikolons plus 1 entspricht der Anzahl der Spalten der auf dem Stream stehenden Matrix. Anschließend bringen wir die Matrix A mittels der Funktionen *set_no_of_rows* bzw. *set_no_of_columns* auf die notwendige Größe und lesen dann vom Beginn des Streams an, dem PARI–Format entsprechend, abwechselnd Charakters und *bigint*–Werte ein. Die *bigint*–Werte schreiben wir zeilenweise in das Wertearray der Matrix A. Gelesene Kommata und Semikolons verwerfen wir. Der Einlesevorgang endet mit dem Einlesen der schließenden Klammer „]“.
  Wie bereits im Falle des LiDIA–Formats, garantiert das zugrundeliegende PARI–Format die Korrektheit dieser Vorgehensweise.

- Im Bezug auf den Datenaustausch mit dem Computer–Algebra–System MATHEMATICA ist die Funktion *A.read_from_mathematica(IN)* für das Einlesen einer Matrix in MATHEMATICA–Format von einem Eingabestream *IN* in die Variable A von Typ *bigint_matrix* verantwortlich.
  Ebenso wie bei der Funktion *A.read_from_gp(IN)*, müssen wir bei dieser Funktion zunächst die Dimensionen der auf dem Stream *IN* stehenden Matrix ermitteln. Dabei hilft uns die Beobachtung, daß eine Matrix im MATHEMATICA–Format einen geklammerten Ausdruck mit maximaler Klammerungstiefe zwei bildet. Als Klammern verwendet MATHEMATICA „{“ und „}“. Wir durchlaufen den gesamten Eingabestream und zählen, wie oft die Klammerungstiefe zwei erreicht und wieder verlassen wird. Diese Anzahl entspricht dann der Anzahl der Zeilen der auf dem Stream *IN* liegenden Matrix. Die Anzahl der Spalten erhalten wir, indem wir die Anzahl der Kommata in einem Bereich mit Klammerungstiefe 2 ermitteln und 1 addieren. Nachdem wir so durch einen Suchlauf durch den gesamten Eingabestream die Dimensionen der Matrix ermittelt haben, passen wir die Dimensionen der Matrix A den ermittelten Dimensionen mit Hilfe der Funktionen *set_no_of_rows* und *set_no_of_columns* an. Anschließend lesen wir, dem MATHEMATICA–Format entsprechend, abwechselnd Charakters und *bigint*–Werte vom Stream *IN* , verwerfen die so gelesenen Klammern „{,}“ und Kommata und schreiben die gelesenen *bigint*–Werte zeilenweise in das Wertearray der Matrix A.
  Die hier beschriebene Vorgehensweise ist wegen der Formatcharakterisierung des MATHEMATICA–Formats korrekt.

- Als letztes Format betrachten wir das MAPLE–Format. Um eine Matrix, die im MAPLE–Format auf dem Stream *IN* vorliegt, einzulesen, verwenden wir die Funktion *read_from_maple(IN)*.
  Das Durchlaufen des Eingabestreams zwecks Dimensionsermittlung ist bei diesem Format nicht nötig. Wir beginnen am Anfang des Eingabestreams und suchen nach dem ersten Vorkommen des Musters „$< x > .. < y >$“ mit x und y als *long*–Werte. Damit erhalten wir als Anzahl der Zeilen $|y-x|$. Desweiteren merken wir uns den Wert y in der Variablen $startr$. Das zweite Vorkommen des obigen Musters bestimmt analog die Anzahl der Spalten und den Wert der Variablen $startc$. Nachdem wir nun die Dimensionen der auf dem Stream liegenden Matrix kennen, passen wir die Dimensionen der Matrix A mittels der Funktionen *set_no_of_rows* bzw. *set_no_of_columns* den ermittelten Dimensionen an. Nun suchen wir $(A.rows \cdot A.columns)$–mal nach dem Muster „$(< x >, < y >) = < w >$“ . Dabei sind x und y *long*–Werte und w ein *bigint*–Wert. Den Wert w schreiben wir an die Position $(|x - startr|, |y - startc|)$.
  Die Korrektheit dieser Vorgehensweise ergibt sich aus dem weiter oben dargelegten MAPLE–Format.

Befassen wir uns jetzt mit der Fehlerbehandlung in den obigen Funktionen. Allen oben angeführten Streamlesefunktionen liegt folgendes Fehlerbehandlungskonzept zugrunde:

1. Lese das nächste Symbol des Streams!
2. Überprüfe, ob dieses Symbol dem jeweiligen Ablageformat entspricht! Entspricht das Symbol nicht dem zugrundeliegenden Format, gib eine Fehlermeldung aus und beende die Funktionsausführung!

Bei Tests mit diesem Konzept hat sich herausgestellt, daß die Lesefunktionen durch die Formatüberprüfungen nach jedem Zeichen des Streams sehr langsam werden. Aus diesem Grund haben wir uns entschlossen, auf eine Formatüberprüfung an dieser Stelle zu verzichten.

13.2 Ein- und Ausgabefunktionen ohne Datenformatbeschränkung

In der Klasse *bigint_matrix* wurden die Operatoren << und >> zwecks Ein– und Ausgabe von Matrizen ohne Datenformatbeschränkung überladen.

Prototypen:

- `ostream & operator << (ostream &OUT, const bigint_matrix &A)`
- `istream & operator >> (istream &IN, bigint_matrix &A)`

Für den Eingabeoperator bedeutet dies, daß er in der Lage ist, unabhängig davon, in welchem der im vorigen Abschnitt dargestellten Formate die einzulesende Matrix vorliegt, diese korrekt einzulesen. Betrachtet man die oben angeführten Formate genauer, so wird deutlich, daß man durch das Einlesen des ersten Zeichens bereits entscheiden kann, welches Format vorliegt. In der Implementierung des Eingabeoperators tun wir dies und rufen anschließend die dem Format entsprechende Eingabefunktion mit Datenbeschränkung auf.
Für den Ausgabeoperator ist eine solche Charakterisierung etwas schwieriger. Er ist in der Lage in Abhänigkeit von einer klasseninternen, variablenabhängigen *long*–Variablen *print_mode*, die Matrix in jedem der unterstützten Formate auszugeben. Zusätzlich besitzt der Ausgabeoperator einen sogenannten „beautify"–Modus. Dieser dient dazu, die Darstellung einer Matrix auf dem Bildschirm etwas lesbarer zu machen. Dies erreichen wir, indem wir die Ausgabe kleinerer Einträge der Matrix durch das Hinzufügen von Leerzeichen auf die Dezimallänge des betragsmäßig größten Eintrages bringen.

Beispiel:

Ein Matrix, die im PARI–Format das Aussehen „[1,324,73;9846,9,12;943,4,3]“ besitzt, wird im „beautify“–Modus wie folgt ausgegeben:

```
(   1  324   73)
(9846    9   12)
( 943    4    3)
```

Welcher Wert der Variablen *print_mode* nun welche Ausgabe auslöst, kann man folgender Tabelle entnehmen.

Wert von MODE	Ausgabeformat
0	„beautify“
1	LiDIA
2	PARI
3	MATHEMATICA
4	MAPLE

Tabelle 13.2.1: Kodierung für Ausgabemodus

Den Werte der Variablen *print_mode* für eine Variable A des Typs *bigint_matrix* ändert man durch den Aufruf der folgenden Funktion.

- `void A.set_print_mode(long art)`

Hierbei gibt der Wert des Parameters *art* den neuen Ausgabemodus für die Variable A an. Ist der Wert des Parameters *art* kleiner als Null oder größer als 4, wird der Ausgabemodus automatisch auf „beautify“ gesetzt.

Teil VI

Anhang

Anhang A

Handler und Testprogramme im Zusammenhang mit LiDIA

A.1 Testprogramme

Bei der Entwicklung und Implementierung der Klasse *bigint_matrix* haben wir eine Vielzahl von Testprogrammen geschrieben. Dabei hat sich deutlich gezeigt, daß das richtige Austesten einer Klasse enorm schwierig ist. Wir wollen aber an dieser Stelle nicht auf Konzeptionen zum richtigen Testen oder auf Aspekte der Softwarequaltitätssicherung eingehen, sondern lediglich zwei Testprogramme vorstellen, die im Zusammenhang mit LiDIA die Portierung und Installation der Bibliothek testen sollen.

Die beiden Testprogramme sollen die einzelnen Funktionen der Klasse *bigint_matrix* auf mögliche Portierungsfehler hin testen.

1. Das erste Testprogramm, das zur Zeit in LiDIA den Namen *bigint_matrix_appl.c* trägt, geht folgendermaßen vor:

   (a) Zuerst werden drei Testmatrizen, die im LiDIA–Paket enthalten sind, von Platte gelesen.

   (b) Anschließend werden alle wichtigen Funktionen mit diesen Matrizen als Eingabe ausgeführt und die jeweiligen Ergebnisse in ein File namens *bigint_matrix_appl.out* geschrieben.

   Der Benutzer kann anschließend durch Vergleich der Dateien *bigint_matrix_appl.out* und *bigint_matrix_appl.dat*, die ebenfalls im LiDIA–Paket enthalten ist, überprüfen, ob alle Routinen ordnungsgemäß arbeiten.

2. Dem zweiten Testprogramm liegt eine gänzlich andere Vorgehensweise zugrunde. Nacheinander generiert es zufällige Matrizen mit Dimensionen zwischen 10

und 200. Dann führt das Programm verschiedene Berechnungen mathematischer Objekte aus und testet, ob für eine Eingabematrix A folgende Bedingungen gelten.

- Ist A eine quadratisch Matrix, dann testen wir:
  - Ist $\det(A) \cdot I_n$ gleich $A \cdot adj(A)$?
  - Ergibt die Matrix A eingesetzt in das charakteristische Polynom der Matrix A die Nullmatrix (Satz von Cayley–Hamilton [3])?
  - Ist die Gitterdeterminante ein Teiler der Determinante?
  - u.a.
- Ansonsten testen wir:
  - Liefern alle Routinen zur HNF– und SNF–Berechnung bei Eingabe einer Matrix A mit vollem Zeilenrang das gleiche Resultat?
  - Ergeben die berechneten Transformationsmatrizen multipliziert mit A eine Matrix in HNF bzw. SNF?
  - u.a.

Dieses Programm ist in der aktuellen Version 1.0 von LiDIA noch nicht enthalten. Wahrscheinlich wird es in einer der folgenden Versionen zum Paket gehören.

A.2 Handler

Um eine einheitliche Ausgabe von Fehlermeldungen, Warnmeldungen, Meldungen der Speicherverwaltung und Debug–Meldungen zu erreichen, schreibt die Konzeption von LiDIA die Verwendung der folgenden Handler vor:

- error_handler
- warning_handler
- memory_handler
- debug_handler

Alle Fehlermeldungen und Warnmeldungen der Klasse *bigint_matrix* erfolgen über diese Handler.

Anhang B

Beipielmatrix

Anhang C

Manual Page aus LiDIA

NAME

bigint_matrix .... Linear algebra over $\mathbb{Z}$

ABSTRACT

bigint_matrix is a class for doing linear algebra over the ring of rational integers. This class may either be used for representing a modul, which is generated by the columns of the matrix

$$A = (a_{i,j})_{\substack{0 \le i < m \\ 0 \le j < n}} \in \mathbb{Z}^{m \times n},$$

or for representing an homomorphism by its matrix. In the first case, the class supports for example computing bases, hermite normal form, smith normal form etc., in the second case it may be used for computing the determinant, the characteristic polynomial, the image, the kernel etc. of the homomorphism.

DESCRIPTION

A variable of type **bigint_matrix** consists of a 2-dimensional array **bigint **value** and two long integers (**rows, columns**) which store the number of rows and columns of the matrix, respectively. In the following descriptions we use $A.rows$ to label the number of rows and $A.columns$ to label the number of columns of matrix A.

Constructors/Destructor

If $a \le 0$ or $b \le 0$ or $v = NULL$ in one of the following constructors the `error_handler` is invoked. The behaviour of some of the following constructors is undefined, if the array v has less than a rows or less than b columns.

`bigint_matrix()`

constructs a 1×1 matrix initialized with zero.

`bigint_matrix(long a)`

constructs an $a \times 1$ matrix initialized with zero.

`bigint_matrix(long a, long b)`

constructs an $a \times b$ matrix initialized with zero.

`bigint_matrix(long a, long b, long ** v)`

constructs an $a \times b$ matrix initialized with the values of the 2-dimensional array v.

`bigint_matrix(long a, long b, const bigint ** v)`

constructs an $a \times b$ matrix initialized with the values of the 2-dimensional array v.

`bigint_matrix(const bigint_matrix & A)`

constructs a copy of matrix A.

`~bigint_matrix()`

ASSIGNMENTS

Let A be of type **bigint_matrix**.

The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void A.assign(const bigint_matrix & B)`

`void assign(bigint_matrix & A, const bigint_matrix & B)`

sets the dimension of matrix A to the dimension of matrix B and copies each entry of B to the corresponding position in A.

ACCESS FUNCTIONS

Let A be of type **bigint_matrix**. Note that we begin the numeration of columns and rows with zero. Therefore the `error_handler` is invoked if i is negative in one of the following functions.

`long A.get_no_of_rows() const`

returns the number of rows of matrix A.

`void A.set_no_of_rows(long i)`

changes the number of rows of matrix A into i. If $i < A.rows$ then the rows $i, \ldots, (A.rows - 1)$ are discarded. If $i > A.rows$ then this function adds $A.rows - i$ new rows initialized with zero.

`bigint * A.operator[ ](long i) const`

`bigint * A.row(long i) const`
allocates memory and returns a copy of row i. If $i \geq A.rows$ then the `error_handler` is invoked.

`void A.sto_row(const bigint *v, long j, long i)`
stores j entries of array v in row i of matrix A starting at column zero. If $j > A.columns$ or $i > A.rows$ the `error_handler` is invoked.

`long A.get_no_of_columns() const`
returns the number of columns of matrix A.

`void A.set_no_of_columns(long i)`
changes the number of columns of matrix A into i. If $i < A.rows$ then the columns $i, \ldots, (A.columns - 1)$ are discarded. If $i > A.rows$ then this function adds $A.rows - i$ new columns initialized with zero.

`bigint * A.operator( )(long i) const`

`bigint * A.column(long i) const`
allocates memory and returns a copy of column i. If $i \geq A.columns$ then the `error_handler` is invoked.

`void A.sto_column(const bigint *v, long j, long i)`
stores j entries of array v in column i of matrix A starting at row zero. If $j > A.rows$ or $i > A.columns$ the `error_handler` is invoked.

`void A.sto(long i, long j, const bigint & x)`
stores x as $a_{i,j}$. If $i \geq A.rows$ or $j \geq A.columns$ the `error_handler` is invoked.

`bigint A.member(long i, long j) const`
returns the value of $a_{i,j}$. If $i \geq A.rows$ or $j \geq A.columns$ the `error_handler` is invoked.

ARITHMETICAL OPERATIONS

bigint_matrix supports the following operators:

(unary) `-`
(binary) `bigint_matrix` *op* `bigint_matrix` with $op \in \{$`+`, `-`, `*`$\}$
(binary) `bigint_matrix` *op* `bigint_matrix` with $op \in \{$`+=`, `-=`, `*=`$\}$
(binary) `bigint_matrix` *op* `bigint` with $op \in \{$`+`, `-`, `*`, `%`$\}$
(binary with assignment) `bigint_matrix` *op* `bigint` with $op \in \{$`+=`, `-=`, `*=`, `%=`$\}$
(binary) `bigint_matrix * (bigint *)`

Here the operators operating on two matrices and the unary minus implement the usual operations known from linear algebra. If the dimensions of the operands do not fulfill the usual restrictions known from linear algebra, the `error_handler` is invoked. The operator

`bigint_matrix * (bigint *)`

realizes the matrix–vector–multiplication.

The operators *op* which take a usual **bigint** as argument do perform the operation *op* componentwise on each entry of the matrix, e.g. `bigint_matrix % bigint` reduces each entry modulo the given **bigint**.

To avoid copying, all operators also exist as functions and of course the restrictions for the dimensions are still valid:

```
void add(bigint_matrix & C, const bigint_matrix & A,
      const bigint_matrix & B)
```

$C = A + B$.

```
void add(bigint_matrix & A, const bigint_matrix & B,
      const bigint & e)
```

Let $r = B.rows$, $c = B.columns$.

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0} & \dots & a_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0}+e & \dots & b_{0,c-1}+e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}+e & \dots & b_{r-1,c-1}+e \end{pmatrix}.$$

```
void subtract(bigint_matrix & C, const bigint_matrix & A,
         const bigint_matrix & B)
```

$C = A - B$.

```
void subtract(bigint_matrix & A, const bigint_matrix & B,
         const bigint & e)
```

Let $r = B.rows$, $c = B.columns$.

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0} & \dots & a_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0}-e & \dots & b_{0,c-1}-e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}-e & \dots & b_{r-1,c-1}-e \end{pmatrix}.$$

```
void multiply(bigint_matrix & C, const bigint_matrix & A,
         const bigint_matrix & B)
```

$C = A * B$.

```
void multiply(bigint_matrix & A, const bigint_matrix & B,
         const bigint & e)
```

$A = e * B$

`void multiply(bigint * v, const bigint_matrix & A,`
`const bigint * w)`

assigns the result of the muliplication $A * w$ to v (matrix – vector – multiplication). If not enough memory has been allocated for vector v the `error_handler` is invoked.

`void remainder(bigint_matrix & A,`
`const bigint_matrix & B, const bigint & e)`

Let $r = B.rows$, $c = B.columns$.

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0} & \dots & a_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0} mod\ e & \dots & b_{0,c'-1} mod\ e \\ \vdots & \ddots & \vdots \\ b_{r'-1,0} mod\ e & \dots & b_{r'-1,c'-1} mod\ e \end{pmatrix}.$$

The entries of A are non-negative values less than $|e|$.

`void negate(bigint_matrix & B, const bigint_matrix & A)`

$B = -A$.

COMPARISONS

The binary operators `==` and `!=` are overloaded and can be used for comparison by components. Let A be an instance of type **bigint_matrix**.

`int A.equal(const bigint_matrix & B) const`

`int equal(const bigint_matrix & A,`
`const bigint_matrix & B)`

returns 1 if A and B have the same dimensions and A and B are identical, 0 otherwise.

`int A.unequal(const bigint_matrix & B) const`

returns 1 if A and B are different, 0 otherwise.

`int unequal(const bigint_matrix & A,`
`const bigint_matrix & B)`

returns 0 if A and B have the same dimensions and A and B are identical, 1 otherwise.

`int A.is_column_zero(long i) const`

returns 1 if all entries of column i of matrix A are zero, 0 otherwise.

`int is_column_zero(const bigint_matrix & A, long i)`

returns 1 if all entries of column i of matrix A are zero, 0 otherwise.

`int A.is_row_zero(long i) const`

returns 1 if all entries of row i of matrix A are zero, 0 otherwise.

```
int is_row_zero(const bigint_matrix & A, long i)
```

returns 1 if all entries of row i of matrix A are zero, 0 otherwise.

Divide and Compose Functions

Let A be of type **bigint_matrix**. The following divide and compose functions allow area overlap of the submatrices.

If the given conditions in the descriptions of the following functions are not fulfilled, the `error_handler` is invoked.

```
void A.divide(bigint_matrix & B, bigint_matrix & C,
              bigint_matrix & D, bigint_matrix & E)
```

partitions matrix A into the submatrices B, C, D and E, with

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows, D.rows, E.rows \leq A.rows \\ 0 \leq B.columns, C.columns, D.columns, E.columns \leq A.columns. \end{gathered}$$

```
void A.divide_h(bigint_matrix & B, bigint_matrix & C)
```

partitions matrix A into the submatrices B and C, with $A = \begin{pmatrix} B & C \end{pmatrix}$, where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows \leq A.rows \\ 0 \leq B.columns, C.columns \leq A.columns. \end{gathered}$$

```
void A.divide_v(bigint_matrix & B, bigint_matrix & C)
```

partitions matrix A into the submatrices B and C, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$, where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows \leq A.rows \\ 0 \leq B.columns, C.columns \leq A.columns. \end{gathered}$$

```
void A.compose(const bigint_matrix & B, const
               bigint_matrix & C, const bigint_matrix &
               D, const bigint_matrix & E)
```

composes the matrices B, C, D and E to the matrix A, with $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$ where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows, D.rows, E.rows \leq A.rows \\ 0 \leq B.columns, C.columns, D.columns, E.columns \leq A.columns. \end{gathered}$$

```
void A.compose_h(const bigint_matrix & B,
                 const bigint_matrix & C)
```

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B & C \end{pmatrix}$ where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows \leq A.rows \\ 0 \leq B.columns, C.columns \leq A.columns. \end{gathered}$$

```
void A.compose_v(const bigint_matrix & B,
                 const bigint_matrix & C)
```

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$ where the following conditions must be fulfilled:

$$\begin{gathered} 0 \leq B.rows, C.rows \leq A.rows \\ 0 \leq B.columns, C.columns \leq A.columns. \end{gathered}$$

Swap Functions

```
void swap(bigint_matrix & A, bigint_matrix & B)
```

swap A and B.

```
void A.swap_columns(long i, long j)
```

swap the entries of the columns i and j in A. If not $0 \leq i, j < A.columns$ the `error_handler` is invoked.

```
void A.swap_rows(long i, long j)
```

swap the entries of the rows i and j in A. If not $0 \leq i, j < A.rows$ the `error_handler` is invoked.

Linear Algebra

If the dimensions of the arguments don't fulfill the usual restrictions known from linear algebra, the `error_handler` is invoked.

```
void A.adj(const bigint_matrix & B)
```

```
bigint_matrix adj(const bigint_matrix & B)
```

stores the adjoint matrix of B to A.

```
bigint * A.charpoly() const
```

```
bigint * charpoly(const bigint_matrix & A)
```

returns an array v of coefficients of the characteristic polynomial of A, where v[i] contains the coefficient of x^i (see [21]).

```
bigint A.det() const
```

`bigint det(const bigint_matrix & A)`

returns the determinant of matrix A.

`void A.det(x) const`

$x = det(A)$.

`void A.diag(const bigint & a, const bigint & b)`

```
void diag(bigint_matrix & A, const bigint & a,
          const bigint & b)
```

stores a at all positions (i, i) with $0 \leq i < min(A.rows, A.columns)$ and b at all other positions.

`void A.hnf()`

`void hnf(bigint_matrix & A)`

transforms A in hermite normal form. Since the efficiency of the specific HNF reduction algorithms always depends on the matrix, we implemented several reduction strategies. Out of these, `hnf` chooses the algorithm, which is the fastest on big, dense matrices. Depending on your application, you might want to choose some other algorithm out of the list of algorithms described in the next section.

`void A.hnf(bigint_matrix & T)`

`void hnf(bigint_matrix & A, bigint_matrix & T)`

transforms A in hermite normal form and sets T to the corresponding transformation matrix. That means: $A * T = HNF(A)$. Again, this is just an interface for choosing the generally fastest algorithm.

`void A.image(const bigint_matrix & B)`

stores a basis of the image of the homomorphism defined by B to A. That means, the columns of matrix A are a generating system of this image.

`bigint_matrix image(const bigint_matrix & B)`

returns a basis of the image of the homomorphism defined by B.

```
void A.invimage(const bigint_matrix & B,
                const bigint * v)
```

stores the solutions of the linear equation $B * x = v$ to matrix A. The first column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B.

`bigint_matrix invimage(const bigint_matrix & B,`
`                        const bigint *v)`

returns a **bigint_matrix** A with the following properties: The first column of matrix A is a solution x of the system $B * x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B.

`void A.kernel(const bigint_matrix & B)`

`bigint_matrix kernel(const bigint_matrix & B)`

stores a basis of the kernel of the homomorphism with matrix B to A (see [21]), i.e., the columns of the matrix A form a generating system of the kernel.

`bigint A.latticedet() const`

`bigint latticedet(const bigint_matrix & A)`

returns a positive multiple of the determinant of the lattice generated by the columns of A (see [9]).

`void A.latticedet(bigint & x) const`

$x =$ `latticedet`(A).

`bigint A.latticedet2() const`

`bigint latticedet2(const bigint_matrix & A)`

returns a positive multiple of the lattice determinant of A (see [9]), which gernerally is much smaller than the result of `A.latticedet()`. Note that this function takes more running time than `A.latticedet()`.

`void A.latticedet2(bigint & x) const`

$x =$ `latticedet2`(A).

`long * A.lininc() const`

`long * lininc(const bigint_matrix & A)`

returns an array of type long with the rank of A (at position zero) followed by the indices of the linearly independent columns of A.

`long * A.lininr() const`

`long * lininr(const bigint_matrix & A)`

returns an array of type long with the rank of A (at position zero) followed by the indices of the linearly independent rows of A (see [21]).

`long A.rank() const`

`long rank(const bigint_matrix & A)`

returns the rank of matrix A (see [21]).

`bigint_matrix A.trans() const`

`bigint_matrix trans(const bigint_matrix & A)`

returns A^T.

`void A.reginvimage(const bigint_matrix & B,`
`                   const bigint_matrix & C)`

calculates a matrix X and **bigint** coefficients $g_0, \ldots, g_{m-1}$, where $m = B.columns$, such that

$$B * X = C * \begin{pmatrix} g_1 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & g_{m-1} \end{pmatrix}$$

and sets the matrix

$$A = \begin{pmatrix} X \\ g_0 \cdots g_{m-1} \end{pmatrix}$$

if B is regular. Otherwise the `error_handler` is invoked.

`bigint_matrix reginvimage(const bigint_matrix & B,`
`                          const bigint_matrix & C)`

calculates a matrix X and **bigint** coefficients $g_0, \ldots, g_{m-1}$, where $m = B.columns$, such that

$$B * X = C * \begin{pmatrix} g_1 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & g_{m-1} \end{pmatrix}$$

and returns a matrix

$$A = \begin{pmatrix} X \\ g_0 \cdots g_{m-1} \end{pmatrix}$$

if B is regular. Otherwise the `error_handler` is invoked.

`void A.solve(const bigint_matrix & B, const bigint * v)`

stores the solutions of the linear equation $B * x = v$ to matrix A. The first column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B.

`bigint_matrix solve(const bigint_matrix & B,`
`const bigint *v)`

returns a **bigint_matrix** A with the following properties: The first column of matrix A is a solution x of the system $B * x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B.

`void A.snf()`

`void snf(bigint_matrix & A)`

transforms A to Smith Normal Form (Gauss Jordan elimination). As was the case with the `hnf`, this is an interface for choosing the generally fastest implemented reduction algorithm.

`void A.snf(bigint_matrix & B, bigint_matrix & C)`

`void snf(bigint_matrix & A, bigint_matrix & B,`
`bigint_matrix & C)`

transforms A to Smith Normal Form and B, C to the corresponding transformation matrices with $\texttt{snf}(A) = B * A * C$. Again, this is just an interface for choosing the generally fastest algorithm.

Hermite Normal Form

The following functions transform the given matrix in Hermite Normal Form, that means an upper triangular matrix with:

$$a_{i,A.columns-A.rows+i} > a_{i,j} \geq 0 \quad \forall j > (A.columns - A.rows + i)$$

In these functions we assume that for the given matrices the rank is equal to the number of rows. If this condition is not fulfilled the functions using modular methods for the computation (i.e., the functions, whose names start with `hnfmod`) invoke the `error_handler`, whereas the other functions terminate at the position, where this is detected, returning a partially HNF-reduced matrix.

`void A.hnf_simple()`

`void hnf_simple(bigint_matrix & A)`

transforms A to Hermite Normal Form using Gauß–Elimination.

`void A.hnf_simple(bigint_matrix & T)`

`void hnf_simple(bigint_matrix & A, bigint_matrix & T)`

transforms A to Hermite Normal Form and sets T to the corresponding transformation matrix (i.e., $HNF(A) = A * T$) using a variant of Gauß–Elimination.

`void A.hnf_havas()`

`void hnf_havas(bigint_matrix & A)`

transforms A to Hermite Normal Form using result of [12] [13] and [15].

`void A.hnf_havas(bigint_matrix & B)`

`void hnf_havas(bigint_matrix & A, bigint_matrix & T)`

transforms A to Hermite Normal Form and sets T to the corresponding transformation matrix (i.e., $HNF(A) = A * T$) using results of [12] [13] and [15].

`void A.hnfmod_dkt()`

`void hnfmod_dkt(bigint_matrix & A)`

transforms A to Hermite Normal Form using the algorithms of [7].

`void A.hnfmod_dkt(const bigint & x)`

`void hnfmod_dkt(bigint_matrix & A, const bigint & x)`

transforms A to Hermite Normal Form using the algorithm of [7] assuming that x is a multiple of the lattice determinant.

`void A.hnfmod_cohen()`

transforms A to Hermite Normal Form using the algorithm of [5].

`void A.hnfmod_cohen(const bigint & x)`

transforms A to Hermite Normal Form using the algorithm of [5] assuming that x is a multiple of the lattice determinant.

`void A.hnfmod_mueller(bigint_matrix & T)`

`void hnfmod_mueller(bigint_matrix & A, bigint_matrix & T)`

transforms A to Hermite Normal Form and sets T to the corresponding transformation matrix with $HNF(A) = A * T$ (see [21]).

Smith Normal Form

The following functions tranform the given matrix into Smith Normal Form, that means into a diagonal matrix $A = (a_{i,j})_{0 \leq i \leq n, 0 \leq j \leq n}$ such that for all $0 \leq i < n$:

$$a_{i,i} \geq 0$$
$$a_{n,n} \geq 0$$
$$a_{i,i} | a_{i+1,i+1}$$

`void A.snf_simple()`

`void snf_simple(bigint_matrix & A)`

transforms A to Smith Normal Form using a variant of Gauß–Jordan elimination.

`void A.snf_simple(bigint_matrix & C, bigint_matrix & B)`

```
void snf_simple(bigint_matrix & A, bigint_matrix & C,
                bigint_matrix & B)
```

transforms A to Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e., $SNF(A) = C * A * B$) using a variant of Gauß–Jordan elimination.

`void A.snf_hartley()`

`void snf_hartley(bigint_matrix & A)`

transforms A to Smith Normal Form using a variant of the algorithm of Hartley and Hawkes.

`void A.snf_hartley(bigint_matrix & C, bigint_matrix & B)`

```
void snf_hartley(bigint_matrix & A, bigint_matrix & C,
                 bigint_matrix & B)
```

transforms A to Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e., $SNF(A) = C * A * B$) using a variant of the algorithm of Hartley and Hawkes.

`void A.snf_havas()`

`void snf_havas(bigint_matrix & A)`

transforms A to Smith Normal Form using results of [12] [13] and [15].

`void A.snf_havas(bigint_matrix & C, bigint_matrix & B)`

```
void snf_havas(bigint_matrix & A, bigint_matrix & C,
               bigint_matrix & B)
```

transforms A to Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e., $SNF(A) = C * A * B$) using results of [12] [13] [15].

The following functions are based on results of [12], [13] and [15]. The extensions `add`, `mult`, `new` specify how column norms and row norms are combined for choosing the pivot strategy and the paramter k is used to select the L_k–norm which is used as described in these papers (default: k = 1). If $k < 0$ the behaviour of the functions is undefined.

Member functions without matrix parameters and functions with one matrix parameter transform the given matrix in Smith Normal Form. All other functions compute also the corresponding transformation matrices B and C with $SNF(A) = C * A * B$:

`void A.snf_mult(long k)`

`void snf_mult(bigint_matrix & A)`

```
void A.snf_mult(bigint_matrix & C, bigint_matrix & B,
                long k)
```

```
void snf_mult(bigint_matrix & A, bigint_matrix & C,
              bigint_matrix & B, long k)
```

`void A.snf_add(long k)`

`void snf_add(bigint_matrix & A, long k)`

```
void A.snf_add(bigint_matrix & C, bigint_matrix & B,
               long k)
```

```
void snf_add(bigint_matrix & A, bigint_matrix & C,
             bigint_matrix & B, long k)
```

`void A.snf_new(long k)`

`void snf_new(bigint_matrix & A, long k)`

```
void A.snf_new(bigint_matrix & C, bigint_matrix & B,
               long k)
```

```
void snf_new(bigint_matrix & A, bigint_matrix & C,
             bigint_matrix & B, long k)
```

The following functions use modular algorithms, which are based on [5] and [7].

`void A.snfmod_dkt()`

`void snfmod_dkt(bigint_matrix & A)`

transforms A to Smith Normal Form [7].

`void A.snfmod_dkt(const bigint & x)`

`void snfmod_dkt(bigint_matrix & A, const bigint & x)`

transforms A to Smith Normal Form assuming that x is a multiple of the lattice determinant [7].

`void A.snfmod_cohen()`

`void snfmod_cohen(bigint_matrix & A)`

transforms A to Smith Normal Form [5].

`void A.snfmod_cohen(const bigint & x)`

`void snfmod_cohen(bigint_matrix & A, const bigint & x)`
transforms A to Smith Normal Form assuming that x is a multiple of the lattice determinant [5].

NORMS AND BOUNDS

`bigint A.max() const`

`bigint max(const bigint_matrix & A)`
returns the maximum of all components of matrix A.

`void A.max(bigint & x) const`
$x = max(A)$.

`bigint A.max_abs() const`

`bigint max_abs(const bigint_matrix & A)`
returns the maximum of the absolute values of all components of matrix A.

`void A.max_abs(bigint & x) const`
$x =$`max_abs`(A)

`bigint A.min() const`

`bigint min(const bigint_matrix & A)`
returns the minimum of all components of A.

`void A.min(bigint & x) const`
$x = min(A)$

`bigint A.min_abs() const`

`bigint min_abs(const bigint_matrix & A)`
returns the minimum of the absolute values of all components of matrix A.

`void A.min_abs(bigint & x) const`
$x =$`min_abs`(A)

`bigint A.hadamard() const`

`bigint hadamard(const bigit_matrix & A)`
returns the hadamard bound of A.

`void A.hadamard(bigint & x) const`
$x =$`hadamard`(A)

`void A.row_norm(bigint & x, long i, long j)`

sets $x = \sum_{l=0}^{A.columns-1} |a_{i,l}^{j}|$. If not $0 \leq i < A.rows$ the `error_handler` is invoked.

`bigint A.row_norm(long i, long j)`

`bigint row_norm(const bigint_matrix & A, long i, long j)`

returns $\sum_{l=0}^{A.columns-1} |a_{i,l}^{j}|$. If not $0 \leq i < A.rows$ the `error_handler` is invoked.

`void A.column_norm(bigint & x, long i, long j)`

sets $x = \sum_{l=0}^{A.rows-1} |a_{l,i}^{j}|$. If not $0 \leq i < A.columns$ the `error_handler` is invoked.

`bigint A.column_norm(long i, long j)`

```
bigint column_norm(const bigint_matrix & A, long i,
                   long j)
```

returns $\sum_{l=0}^{A.rows-1} |a_{l,i}^{j}|$. If not $0 \leq i < A.columns$ the `error_handler` is invoked.

RANDOMIZE

`void A.randomize(bigint & b)`

fills matrix A with random values between 0 and b.

`bigint_matrix randomize(long i, long j, const bigint & b)`

returns a matrix with i rows and j columns and random values between 0 and b.

Input/Output

The **istream** operator >> and the **ostream** operator << are overloaded. Usually the input and output of the program are of the following ASCII format:

$$r \sqcup c \sqcup a_{0,0} \sqcup a_{0,1} \sqcup \ldots \sqcup a_{0,c-1} \sqcup a_{1,0} \sqcup \ldots \sqcup a_{1,c-1} \sqcup \ldots \sqcup a_{r-1,0} \sqcup \ldots \sqcup a_{r-1,c-1},$$

where $r = A.rows$ and $c = A.columns$. Note that the **ostream** operator << is an exception, it outputs a **bigint_matrix** in the form

$$\begin{array}{cccccc} (& a_{0,0} & a_{0,1} & \ldots & a_{0,c-1} &) \\ (& a_{1,0} & \ldots & \ldots & a_{1,c-1} &) \\ (& \vdots & \vdots & \vdots & \vdots &) \\ (& a_{r-1,0} & \ldots & \ldots & a_{r-1,c-1} &) \end{array}$$

`void A.write_to_stream(const ostream & out) const`

writes matrix A to **ostream** *out*.

`void A.read_from_stream(const istream & in)`

reads matrix A from **istream** *in*.

Interfaces

`void A.write_to_gp(ostream & out) const`

writes matrix A to fstream *out* in GP format.

`void A.read_from_gp(istream & in)`

reads GP - matrix A from file.

`void A.write_to_mathematica(ostream & out) const`

writes matrix A to fstream *out* in MATHEMATICA format.

`void A.read_from_mathematica(istream & in)`

reads MATHEMATICA - matrix A from file.

SEE ALSO

bigint.h

EXAMPLES

```
#include <LiDIA/bigint_matrix.h>
#include <LiDIA/timer.h>

main()
{
  long j,up;
```

```
  bigint G,DET;
  timer x;

  cout << "\nAnzahl der Stellen: " << flush;
  cin >> up;
  power(G,(bigint)10,up);

  ofstream dz("MessDet");
  x.set_print_mode(HMS_MODE);
  for (j=1;j<=20;j++)
    {
      cout << "\nDimension: " << 10*j << "\n" << flush;
      dz << "\nDimension: " << 10*j << "\n" << flush;
      bigint_matrix A = randomize(j*10,j*10,G);
      A.det(DET);
      x.stop_timer();

      cout << "det = " << DET << "\n" << flush;
      dz << DET << "\n" << flush;
      cout << "\n" <<" Time: " << flush;
      x.print();
    }
  dz.close();
}
```

For further examples please refer to

`LiDIA/src/simple_classes/bigint_matrix_appl.c`.

AUTHOR

Stefan Neis, Patrick Theobald

Tabellenverzeichnis

Literaturverzeichnis

[1] Biehl, I., Buchmann, J., Papanikolaou, T.: LiDIA– *A library for computational number theory.* submitted ISAAC 95.

[2] Blankinship, W. A.: *A new version of the Euclidian algorithm.* Amer. Math. Mon., 1963

[3] Bronstein, I. N., Semendjajew, K. A.: *Taschenbuch der Mathematik.* Frankfurt/Main: Verlag Harri Deutsch Thun, 1989

[4] Buchmann, J.: *Einführung in die Computeralgebra.* Skript zur Vorlesung WS 91/92 an der Universität des Saarlandes

[5] Cohen, H.: *A Course in Computational Algebraic Number Theory.* Graduate text, Springer-Verlag, 1993

[6] Dentzer, R: *libI: Eine lange ganzzahlige Arithmetik.* Universität Heidelberg, 1991

[7] Domich, P. D.: *Residual Hermite Normal Form Computations.* ACM Transactions on Mathematical Software, Vol. 15, No. 3, Seite 275-286, 1989

[8] Domich, P.D., Kannan, R., Trotter, L.E.: *Hermite normal form computation using modular determinant arithmetic.* Mathematics of Operatuions Research, Vol. 12, No. 1, Seite 50-59, 1987

[9] Düllmann, S.: *Ein Algorithmus zur Bestimmung der Klassengruppe positiv definiter binärer quadratischer Formen.* Dissertation, Universität des Saarlandes, 1991

[10] Hafner, J. L., McCurley S.: *ASYMPTOTICALLY FAST TRIANGULARIZATION OF MATRICES OVER RINGS*, SIAM J. COMPUT., Vol. 20, No. 6, 1989,

[11] Hartley, B., Hawkes, T.O.: *Rings, modules and linear algebra.* London: Chapman and Hall, 1970

[12] Havas, G., Holt, D. F., Rees, S.: *Recognizing badly presented $\mathbb{Z}$-modules.* Linear Algebra and its Applications, Vol. 192, Seite 137-163, 1993

[13] Havas, G., Majewski, B. S.: *Integer matrix diagonalization.* Technical Report, No. 277. Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1994

[14] Havas, G., Majewski, B. S.: *Hermite normal form computation for integer matrices.* Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1994

[15] Havas, G., Sterling, L. S.: *Integer matrices and Abelian groups.* Lecture Notes in Computer Science, Vol. 72. Springer-Verlag, 1979

[16] Kannan, R., Bachem, A.: *POLYNOMIAL ALGORITHMS FOR COMPUTING THE SMITH AND HERMITE NORMAL FORMS OF AN INTEGER MATRIX.* SIAM J. COMPUT.,Vol. 8, No. 4, 1979

[17] Kernigham, W. Brian, Ritchie, M. Dennis: *Programmieren in C.* Carl Hanser Verlag, 1983

[18] Knuth, D. E.: *The Art of Computer Programming Volume 2.* Addison–Wesley, 1980

[19] Majewski, B. S., Havas, G.: *On the greatest common divisor of sets of integers.* Technical Report, No. 288. Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1994

[20] Majewski, B. S., Havas, G.: *The complexity of the greatest common divisor computations.* Lecture Notes in Computer Science, to appear

[21] Müller, A.: *Effiziente Algorithmen für Probleme der linearen Algebra über $\mathbb{Z}$.* Diplomarbeit, Universität des Saarlandes, 1994

[22] Sedgewick, R.: *Algorithmen in C.* Bonn, München: Addison–Wesley, 1992

Stichwortverzeichnis