Effiziente Algorithmen zur Berechnung von Elementarteilern ganzzahliger Matrizen - Implementation in GAP

von Andreas Hoppe

Diplomarbeit im Lehrstuhl D für Mathematik
an der RWTH Aachen

Diplom der Humboldt-Universität zu Berlin
im Institut für Reine Mathematik

gefördert von der Hans–Böckler–Stiftung

13. September 1994

Inhaltsverzeichnis

Thesen

1. In einer Klasse von zu einer (rechteckigen) ganzzahligen Matrix ähnlichen Matrizen gibt es eine ausgezeichnete Matrix in der SMITH–Normalform, die Diagonalelemente dieser Diagonalmatrix heißen Elementarteiler.

2. Die Ähnlichkeit von ganzzahligen Matrizen entspricht der Gleichheit von $\mathbb{Z}$–Modulen und der Isomorphie von ABELschen Gruppen.

3. Der GAUSS–Algorithmus ist das Standardverfahren zur Berechnung der Elementarteiler einer ganzzahligen Matrix. Es beruht auf der systematischen Umformung der Matrix.

4. Der GAUSS–Algorithmus leidet unter einer Explosion der Einträge der Zwischenmatrizen.

5. Der GAUSS–Algorithmus hat Wahlfreiheiten. Bei geschickter Auswahl kann die Koeffizientenexplosion entscheidend gebremst werden. (nichtmodulare Verfahren)

6. Es gibt ein Verfahren, daß die Elementarteilerberechnung in endlichen Ringen durchführt und dadurch die Koeffizientenexplosion vermeidet. (modulares Verfahren)

7. Man kann mit dem (diesem) modularen Verfahren nur die Elementarteiler der Matrix berechnen, nicht aber die Transformationsmnatrizen, die die Ausgangsmatrix in ihre SMITH–Normalform überführen.

8. Ich füge den verschiedenen nichtmodularen Verfahren eins hinzu: das Preview–Verfahren. Ich gehe damit an das Problem der Koeffizientenexplosion eher von der informatischen als von der mathematischen Seite heran.

9. Die Idee zum Preview–Verfahren ist nicht nur auf das Problem der nichtmodularen Elementarteilerverfahren beschränkt, sondern es lassen sich dessen Grundideen auf alle mathematische Probleme anwenden, für die ein einfacher Grundalgorithmus existiert, der Wahlentscheidungen mit exponentiell vielen Alternativen aufweist.

10. Ich hoffe, mit der Preview–Idee zu zeigen, daß es möglich ist, die Methoden der algorithmischen Mathematik mit rein informatischen Herangehensweisen zu bereichern.

11. Der Hauptteil meiner Arbeit war die Implementierung des modularen Verfahrens und des Preview–Algorithmus in GAP.

Einleitung

In meiner Diplomarbeit beschäftige ich mich mit neuen effizienten Algorithmen, die die Elementarteiler einer ganzzahligen Matrix berechnen. Meine Aufgabe war es, diese Algorithmen in ggf. veränderter Form im Computergruppentheoriesystem GAP [1] zu implementieren und möglicherweise neue Ideen zu verwirklichen.

Der GAUSS–Algorithmus ist das lange bekannte Standardverfahren, das mithilfe systematischer Umformungen der Matrix die Elementarteiler berechnet. Als die Elementarteiler noch mit der Hand ausgerechnet wurde, gab es an diesem einfachen Verfahren nichts auszusetzen. Aber als der Algorithmus auf dem Computer implementiert wurde, um auch solche Beispiele zu bearbeiten, bei denen niemand mehr auf die Idee kommen würde, zu Fuß zu rechnen, ergab sich ein Problem: Das Verfahren leidet unter der sogenannten Koeffizientenexplosion; die Einträge der Zwischenmatrizen werden immer größer und sprengen die Grenzen jedes Computers. Der GAUSS–Algorithmus hat in seinem Lauf eine Unmenge von Wahlfreiheiten, die nach heuristischen Gesichtspunkten ausgeführt werden können; das ist der erste wichtige Zugang zu dem Problem (nichtmodulare Verfahren). Bei geschickter Auswahl kann die Koeffizientenexplosion entscheidend gebremst werden. Der zweite Zugang (modulares Verfahren) aber löst das Problem auf eine viel direktere Weise — man rechnet in endlichen Ringen, so daß Koeffizientenexplosion von vornherein nicht stattfinden kann.

Im ersten Kapitel gebe ich die Definition der Elementarteiler und der SMITH–Normalform, und zeige die beiden elementaren Verfahren zu deren Berechnung. Außerdem versuche ich, den mathematischen Hintergrund zu beleuchten, warum man sich gerade in der Gruppentheorie für Elementarteiler interessiert.

Im zweiten Kapitel beschreibe ich dann den modularen Ansatz und gehe dabei auf Möglichkeiten der Verbesserung ein. Ich ging von der Formulierung des Algorithmus in [5] aus, implementierte das Verfahren in GAP völlig neu und versuchte, eigene Ideen zu einigen Detailfragen einzubringen, um die speziellen Vorzüge von GAP auszunutzen.

Das dritte Kapitel beschäftigt sich mit den nichtmodularen Verfahren (Varianten des GAUSS–Verfahrens). Ich erörtere die Frage, warum man sich angesichts der Überlegenheit des modularen Verfahrens überhaupt noch mit Verbesserungen des nichtmodularen Verfahrens auseinandersetzt. Ich gebe die bekannten Ansätze an, und füge noch einen neuen hinzu: den Preview–Algorithmus, dessen Strategien ich ausführlich beschreibe.

Mit meinem Preview–Verfahren gehe ich an das Problem der Koeffizientenexplosion eher von der informatischen als von der mathematischen Seite heran. Die Idee zum Preview–Verfahren ist nicht nur auf das Problem der nichtmodularen Elementarteilerverfahren beschränkt, sondern es lassen sich dessen Grundideen auf alle mathematische Probleme anwenden, für die ein einfacher Grundalgorithmus existiert, der Wahlentscheidungen mit exponentiell vielen Alternativen aufweist. Ich hoffe, mit der Preview–Idee zu zeigen, daß es möglich ist, die Methoden der algorithmischen Mathematik mit rein informatischen Herangehensweisen zu bereichern.

Das vierte Kapitel ist eine detaillierte Bedienungsanleitung meines Programms, das einerseits das modulare, anderseits zwei nichtmodulare Verfahren (Best–Remainder und Preview) umfaßt.

Im fünften Kapitel mache ich Zeit– und Koeffizientengrößenvergleiche für eine Reihe von Beispielen und vergleiche sie mit einer hier am Lehrstuhl vorhandenen (alten) FORTRAN–Implementation von George Havas. Ich zeige, wie einige gruppentheoretische Beispiele in GAP erzeugt werden können und wie mein Programm sie und einige andere bearbeitet.

Im sechsten Kapitel gebe ich den Quelltext des Programms und eine Einleitung zum Lesen und gehe auf ausgewählte Besonderheiten der Implementierung ein.

Das siebente Kapitel blickt kritisch auf die Ergebnisse meiner Arbeit zurück.

Das Thema dieser Arbeit verdanke ich Prof. Joachim Neubüser. Er war es, der mich in Aachen willkommen geheißen, mir das Thema gegeben und mich betreut hat. Mein Dank gilt weiterhin meinem Berliner Betreuer Dr. Hubert Grassmann, der mir besonders in der Abschlußphase hilfreich zur Seite gestanden hat. Ich danke Volkmar Felsch für die meisten gruppentheoretischen Beispiele und die Einführung in George Havas' Implementation. Ich danke Martin Schönert für seine Implementation des KANNAN–BACHEM–Verfahrens und der Best–Remainder–Strategy in GAP, und für die Hilfe beim Finden eines Bugs, der keiner war. Ich danke Werner Nickel und auch anderen vom Lehrstuhl D für hilfreiche Diskussionen. Ich danke Dr. George Havas und Dr. Ernst-Günter Gießmann für die Begutachtung meiner Arbeit.

Das Titelblatt gestaltete meine Freundin und angehende Grafikerin Rahel Melis.

Am Schluß verbleibt mir noch zu erklären, daß ich die Arbeit selbständig angefertigt und keine unerlaubten Hilfsmittel benutzt habe.

Kapitel 1

Die Elementarteiler

1.1 Definition

Ich betrachte eine Äquivalenzrelation auf der Menge aller ganzzahligen $m \times n$–Matrizen einer bestimmten Größe. Diese sei folgendermaßen definiert:

Man erhält von einer Matrix eine **ähnliche** Matrix, wenn man sie von links mit einer invertierbaren $m \times m$–Matrix oder von rechts mit einer invertierbaren $n \times n$–Matrix multipliziert.

In jeder Äquivalenzklasse ähnlicher Matrizen gibt es eine ausgezeichnete Matrix einer bestimmten Form, die SMITH–**Normalform**. Eine Matrix $M = (a_{ij})_{i=1,\ldots,m;j=1,\ldots,n}$ ist nach Definition in dieser Normalform, wenn nur Werte auf der Diagonalen ausgehend vom Matrixeintrag links oben (also die Elemente a_{ii}) ungleich Null sind, und wenn außerdem jedes Diagonalelement nichtnegativ ist und ein Diagonalelement einer bestimmten Zeile i alle Diagonalelemente der Zeilen mit höherer Zeilennummer $j > i$ teilt.

Die Matrix sehe also folgendermaßen aus:

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & & & & & \\ \vdots & & \ddots & \vdots & \vdots & & \\ 0 & & \cdots & a_{rr} & 0 & & \\ 0 & & \cdots & 0 & 0 & & \\ \vdots & & & & & \ddots & \\ 0 & & & & & & \end{pmatrix} \quad \begin{array}{l} r = \text{Rang(M)} \\ a_{ij} = 0, \text{ falls } i \neq j \\ a_{ii} > 0, \text{ falls } i \leq r \\ a_{ii} = 0, \text{ falls } i > r \\ a_{ii} | a_{(i+1)(i+1)} \end{array}$$

Die Elemente $a_{11}, \ldots, a_{rr}$ heißen **Elementarteiler** der Matrix.

Es sind zwei prinzipielle Möglichkeiten bekannt, sie zu berechnen. Die eine geht auf SMITH zurück, sie basiert auf der Berechnung von ggTs (größten gemeinsamen Teilern) von Minoren der Matrix einer bestimmten Größe.

Die andere, der GAUSS–**Algorithmus** , ist ein effektives Verfahren, das auf der zielgerichteten äquivalenten Umformung der Matrix beruht.

Der konstruktive Beweis der Existenz einer SMITHschen Normalform basiert auf dem GAUSS–Algorithmus . Der Beweis der Eindeutigkeit folgt danach aus den Überlegungen zu Minoren–ggTs.

1.2 GAUSS–Verfahren

1.2.1 elementare Operationen

Man nimmt sich eine Reihe invertierbarer Matrizen als **elementare Matrizen**, und zwar zweierlei Sorten: einmal $m \times m$–Matrizen für die Multiplikation von links, zum anderen $n \times n$–Matrizen für die Multiplikation von rechts. Da die elementaren Matrizen invertierbar sind, überführt die Multiplikation mit einer elementaren Matrix eine gegebene Matrix in eine ähnliche Matrix.

Andererseits stellt sich die Multiplikation mit einer elementaren $m \times m$–Matrix von links als einfache Operation mit den Zeilen der Matrix heraus; die Multiplikation mit einer elementaren $n \times n$–Matrix von rechts als einfache Operation mit den Spalten der Matrix.

Es gibt zu den zwei Sorten jeweils 3 Typen elementarer Matrizen. Die elementaren Matrizen für die Multiplikation von links haben die gleiche Form wie die für die Multiplikation von rechts und unterscheiden sich nur durch die Größe. Die entsprechende Matrixtransformation, die der Multiplikation der Matrix von rechts entspricht, ist in Klammern angegeben.

- Einsmatrix, außer daß das i–te Diagonalelement nicht 1 sondern -1 ist:

$$\begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & -1 & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix}$$

  Die Multiplikation mit dieser Matrix entspricht der Multiplikation der i–ten Zeile (bzw. i–ten Spalte) mit -1.

- Einsmatrix, außer daß die Diagonalelemente an den Postionen (i,i) und (j,j) =0 sind und dafür die Elemente an den Positionen (i,j) und (j,i) =1:

$$\begin{array}{cc} \begin{array}{ccccccc} & & i & & j & & \end{array} & \\ \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix} & \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \end{array}$$

  Die Multiplikation mit dieser Matrix entspricht der Vertauschung zweier Zeilen (oder Spalten) i und j.

- Einsmatrix, außer daß an irgendeiner Position (i,j) mit $i \neq j$ nicht 0 sondern $a \in \mathbb{Z}$ steht:

$$\begin{array}{cc} \begin{array}{ccccccc} & & i & & j & & \end{array} & \\ \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & 1 & & a & & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{pmatrix} & \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \end{array}$$

  Die Multiplikation mit dieser Matrix entspricht dem Addieren des a–fachen der j–ten Zeile zur i–ten (bzw. dem Addieren des a–fachen der i–ten Spalte zur j–ten).

Bei den jeweiligen Zeilenoperationen hat die Matrix die Größe $m \times m$, und die entsprechende Matrix wird von links multipliziert. Bei den jeweiligen Spaltenoperationen hat die Matrix die Größe $n \times n$, und die entsprechende Matrix wird von rechts multipliziert.

Man sieht leicht, daß diese Matrizen wirklich invertierbar sind. Man macht sich genauso einfach die Entsprechung der Zeilen– und Spaltenoperationen zu der Multiplikation mit den elementaren Matrizen klar.

Es ist zu beachten, daß im Unterschied zur linearen Algebra in den rationalen oder reellen Zahlen eine Multiplikation einer Zeile/Spalte mit einer beliebigen Zahl nicht erlaubt ist, weil diese Operation nicht umkehrbar ist, denn fast alle ganzen Zahlen haben kein inverses Element.

1.2.2 Der Algorithmus

Algorithmus 1 (nach [4])

1. Man wähle ein beliebiges von Null verschiedenes Element p der Matrix als Pivot.
2. Man vertausche Zeilen und Spalten, so daß der Pivot an der Position links oben steht.
3. Wenn der Pivot alle anderen Werte seiner Spalte teilt, gehe zum Schritt 5 über.
4. Man nehme nun ein Element w in der Pivotspalte, das nicht vom Pivot geteilt wird, und dividiere w durch p mit Rest[1] — der Quotient sei q. Man ziehe das q–fache der Pivotzeile von der Zeile von w ab. Man wähle jetzt die Position, an der vorher w stand, als Pivot und gehe zu 2.
5. Wenn der Pivot alle anderen Werte seiner Zeile teilt, gehe zum Schritt 7 über.
6. Man nehme nun ein Element w in der Pivotzeile, das nicht vom Pivot geteilt wird, und dividiere w durch p mit Rest — der Quotient sei q. Man ziehe das q–fache der Pivotspalte von der Spalte von w ab. Man wähle jetzt die Position, an der vorher w stand, als Pivot und gehe zu 2.
7. Man eliminiere jetzt jeden Eintrag in der Pivotzeile und der Pivotspalte durch Addieren eines geeigneten Vielfachen der Pivotspalte bzw. Pivotspalte.
8. Wenn jetzt ein Eintrag der Matrix nicht vom Wert des Pivots geteilt wird, dann addiere dessen Zeile zur Pivotzeile, und gehe zu 5.
9. Man betrachte jetzt die Teilmatrix, die durch Streichen der Pivotzeile und Pivotspalte entsteht. Wenn sie nicht aus lauter Nullen besteht, dann gehe damit zu 1.
10. Man betrachte wieder die gesamte Matrix — sie ist jetzt in Diagonalform, bei der die Teilbarkeit der Diagonalelemente bereits hergestellt ist.
11. Alle Zeilen mit negativen Diagonalelementen werden mit -1 multipliziert, die Matrix ist nun in der SMITHschen Normalform.

1.2.3 Korrektheit

Der **Beweis** der Korrektheit des Algorithmus ist folgendermaßen: Nachdem im Schritt 1 das Pivotelement gewählt wurde, wird in den folgenden Schritten immer nur ein neues Pivotelement gewählt, dessen absoluter Betrag kleiner ist als das ursprüngliche, da das neue Pivotelement immer als Rest einer Restdivision durch das alte hervorgeht.

Das Pivotelement kann nicht Null werden, da dazu die Restdivision aufgegangen sein müßte; aber dazu muß das Element w durch p teilbar gewesen sein; dann wäre aber gar keine Restdivision ausgeführt worden.

Da 1 und -1 alle ganzen Zahlen teilen, führt die sukzessive Verkleinerung des Pivotelementes dazu, daß es irgendwann alle Einträge der Matrix teilt.

[1]Für die Restdivision kann entweder die Variante, die den kleinsten positiven Rest ermittelt, oder die, die den Rest vom kleinsten Betrag ermittelt, genommen werden. Für die letztere Variante benötigt der Algorithmus weniger Schritte, die erstere Restdivision ist meist hardware–näher realisiert und darum schneller.

Damit wird nach endlich langer Zeit wieder zum Schritt 1 zurückgegangen. Da dem eine Verkleinerung der betrachteten Matrix vorausgeht, folgt daraus, daß der Algorithmus nach endlich langer Zeit abbricht.

Weiterhin hat der Algorithmus an der Matrix nur Operationen ausgeführt, die die Ähnlichkeit zur Ausgangsmatrix erhalten. Außerdem ist leicht zu sehen, daß die Matrix, nachdem der Algorithmus beendet ist, in der SMITH–Normalform ist. Daraus folgt nun jedoch auch die **Existenz der** SMITH–**Normalform**.

Satz 1 *Jede ganzzahlig invertierbare Matrix (d. h. eine Matrix, die Determinante 1 oder -1 hat, auch unimodulare Matrix) läßt sich als Produkt von elementaren Matrizen schreiben.*

Der **Beweis** erfolgt konstruktiv, indem man eine beliebige invertierbare Matrix schrittweise mithilfe der drei elementaren Operationen:

- Multiplikation einer Zeile/Spalte mit -1
- Vertauschung zweier Zeilen/Spalten
- Addition eines ganzzahligen Vielfachen einer Zeile/Spalte zu einer anderen Zeile/Spalte

zu einer Einsmatrix umformt.

Da eine invertierbare Matrix die Determinante 1 oder -1 hat, und die Multiplikation mit einer invertierbaren Matrix den absoluten Betrag der Determinante nicht ändert, ist die SMITH–Normalform einer invertierbaren Matrix die Einsmatrix gleicher Größe. Daraus folgt aber schon der Satz.

1.2.4 Komplexität

Der Algorithmus hat kubische Komplexität mit Zahlen, die sich nicht von vornherein polynomial beschränken lassen. Das zeigt sich in der naiven Implementation des GAUSS–Verfahrens sehr deutlich: die Einträge der Zwischenmatrizen auf dem Weg zur Normalform können auch dann sehr groß werden, wenn die Einträge der Ausgangsmatrix genauso wie die Einträge der Normalform klein sind. Das ist unter der Bezeichnung **entry explosion** oder **Koeffizentenexplosion** bekannt.

Unter bestimmten Voraussetzungen ist das Verfahren dennoch polynomial.

1.3 Minoren–ggT–Verfahren

Eine $k \times l$-**Teilmatrix** einer Matrix $(a_{ij})_{i=1,\ldots,m;j=1,\ldots,n}$ ist durch $1 \leq i_1 < \cdots < i_k \leq m$ und $1 \leq j_1 < \cdots < j_l \leq n$ als $(a_{i_t j_u})_{t=1,\ldots,k;u=1,\ldots,l}$ bestimmt. Eine k-**Teilmatrix** ist eine quadratische $k \times k$-Teilmatrix. Wenn ich im folgenden von Teilmatrizen reden werde, so meine ich stets quadratische Teilmatrizen. Eine **Rangteilmatrix** ist eine r–Teilmatrix, wobei r der Rang der Matrix ist. Ein d–**Minor** einer Matrix ist die Determinante einer Teilmatrix der Größe $d \times d$. Ein **Rangminor** ist ein r–Minor, wobei r der Rang der Matrix ist.

Satz 2 *Der größte gemeinsame Teiler (ggT) aller d–Minoren ($d \in \mathbb{Z}$) ist auf einer Äquivalenzklasse*[2] *von Matrizen konstant.*

Das läßt sich leicht zeigen, indem man beweist, daß sich der d–Minoren–ggT bei einer elementaren Operation nicht ändert.

Aus diesem Satz folgt, daß der d–Minoren–ggT einer beliebigen Matrix gleich dem d–Minoren–ggT der SMITHschen Normalform ist. Hier sind aber die d–Minoren–ggTs unmittelbar abzulesen:

- Wenn d größer als der Rang der Matrix ist, dann gibt es keinen von Null verschiedenen Minor, also ist der d–Minoren–ggT gleich 0.
- Wenn d kleiner gleich dem Rang ist, so sind die Teilmatrizen, die eine Determinante ungleich Null aufweisen, Diagonalmatrizen, deren Diagonalelemente von den Elementarteilern gebildet werden.

Darum sind die d–Minoren entweder Null oder ein Produkt von d Elementarteilern. Der ggT der d–Minoren ist dann das Produkt der d kleinsten Elementarteiler — somit gilt der Satz:

[2] bezüglich der in diesem Kapitel definierten Ähnlichkeitsrelation

Satz 3 *Wenn $e_1, \ldots, e_r$ die Elementarteiler einer ganzzahligen Matrix M vom Rang r sind, wobei $\forall i$: $e_i | e_{i+1}$ gelten soll, so ist der ggT aller d–Minoren gleich 0, falls $d > r$ und $\prod_{i=1}^{d} e_i$, falls $1 < d \leq r$.*

Daraus ergibt sich folgendes Verfahren zur Berechnung der Elementarteiler: Man rechnet für jedes $d = 1, \ldots,$ Rang(M) den d–Minoren–ggT aus und erhält die Elementarteiler durch Division: Sei D_d der d–Minoren–ggT einer Matrix M, dann ergibt sich für den i–ten Elementarteiler

$$e_i := \begin{cases} D_1 & \text{falls } i = 1 \\ D_i / D_{i-1} & \text{falls } 1 < i \leq \text{ Rang}(M) \end{cases}$$

Die **Korrektheit** des Verfahrens folgt aus dem vorhergehenden Satz 3. Da sich nun die Determinanten als einfache Ausdrücke in den Koeffizienten der Matrix schreiben lassen, sind die Elementarteiler eindeutig bestimmt; damit gilt auch die **Eindeutigkeit der** SMITH–**Normalform**.

1.3.1 Komplexität

Es gibt exponentiell viele Minoren, darum ist dieses Verfahren in jedem Fall von exponentieller Komplexität, hoffnungslos uneffektiv mit anderen Worten.

1.4 Bedeutung der Elementarteiler

1.4.1 Zerlegen von ABELschen Gruppen

Für meine Arbeit stand vor allem eine Anwendung im Vordergrund: das Zerlegen einer endlich erzeugten ABELschen Gruppe in ein direktes Produkt von zyklischen Gruppen.

Sei die Gruppe durch eine Menge von Relatoren in bestimmten Erzeugenden gegeben. Da die Gruppe kommutativ ist, kann ich die Erzeugenden in den Relatoren ordnen. Ich schreibe die Relatoren nun in Matrixform, indem ich jedem Erzeugenden eine Spalte und jedem Relator eine Zeile zuordne und die sich aus dem Ordnungsprozeß ergebenden Exponenten der einzelnen Erzeugenden als ganzzahligen Eintrag in die Matrix schreibe.

Sei also die Gruppe in den Erzeugern $E_1, \ldots, E_n$ durch die Relatoren $R_1, \ldots, R_m$ gegeben, dann kann ich die Erzeugenden des i–ten Relators $(i = 1, \ldots, m)$ ordnen, so daß ich R_i als $E_1^{a_{i1}} E_2^{a_{i2}} \cdots E_n^{a_{in}}$ schreiben kann. Meine Matrix ist jetzt $M = (a_{ij})_{i=1,\ldots,m; j=1,\ldots,n}$.

Es gilt folgender Satz:

Satz 4 ([4]) *Ähnliche Matrizen gehören isomorphen Gruppen an.*

Der **Beweis** läßt sich wieder mithilfe der elementaren Matrixoperation durchführen. Die elementaren Zeilenoperationen entsprechen Umformungen der Relatorenmenge, die elementaren Spaltenoperationen entsprechen Umdefinitionen der Erzeugenden, den sogenannten TIETZE–Transformationen.

Somit kann ich eine ausgezeichnete Gruppe in der Isomorphieklasse unserer betrachteten Gruppe als Entsprechung der SMITH–Normalform unter den Matrizen berechnen, indem ich die eben betrachtete Abbildung in der anderen Richtung betrachte — mir aus den Einträgen der Matrix die Relatoren dieser ausgezeichneten Gruppe herstelle. Wegen der einfachen Form der Matrix in Normalform läßt sich daraus folgendermaßen die Zerlegung der ABELschen Gruppe in zyklische Gruppen ablesen:

Wenn in einer Spalte ein Elementarteiler 1 steht, dann ist der dieser Spalte entsprechende Erzeuger das Einselement, und der Erzeuger braucht in der Zerlegung der Gruppe in zyklische Untergruppen nicht berücksichtigt zu werden.

Wenn in einer Spalte ein Elementarteiler $e \neq 1$ steht, dann hat der dieser Spalte entsprechende Erzeuger die Ordnung e und erzeugt eine zyklische Untergruppe der Ordnung e, die als direkter Faktor von der zu zerlegenden Gruppe abgespalten werden kann.

Wenn in einer Spalte alle Einträge gleich Null sind, dann ist die von dem dieser Spalte entsprechenden Erzeuger erzeugte Untergruppe die freie Gruppe eines Erzeugers und kann als direkter Faktor ebenfalls abgespalten werden.

1.4.2 typische Beispiele

Da meine wichtigen Beispiele gerade von gruppentheoretischen Problemen herrühren, ist natürlich die Frage interessant, ob sich die Matrizen, die sich z.B. aus einem Reidemeister–Schreier–Algorithmus ergeben, systematisch anders verhalten als zufällig verteilte[3] Matrizen.

Das Verständnis darüber warum das so ist, könnte helfen, diese speziellen Eigenschaften der Matrizen auszunutzen, um heuristische Wahlentscheidungen besser auszuführen.

Ich habe folgende Erfahrung gemacht: Bei größeren zufälligen Matrizen sind meistens alle Elementarteiler 1, wenn aber einmal ein nichttrivialer Elementarteiler auftritt, so war er relativ häufig eine große Zahl. Wohingegen die gruppentheoretischen Matrizen nur kleine Elementarteiler hatten, wenn nicht sogar alle Elementarteiler trivial waren.

Abgesehen davon stellt sich auch heraus, daß sich die gruppentheoretischen Matrizen auch bei etwa gleicher Matrixgröße, gleichgroßen Einträgen und gleicher Anzahl von Nullen angenehmer verhalten und auch bei einfachen Strategien nicht so schnell unter explodierenden Zwischenwerten leiden.

Da diese Observationen aber nur an einer geringen Zahl von Beispielen gemacht wurde, sind sie mit Vorsicht zu genießen, und nur als Vorschlag zu verstehen, welcher Art die Unterschiede bei praktisch häufig vorkommenden Beispielen sein könnten.

[3] insbesondere gleichverteilte, aber auch diskretisiert Gauss–verteilte

Kapitel 2

modularer Ansatz

Ein Standardinstrument der Computeralgebra mit ganzen Zahlen zur Vermeidung von explodierenden Zwischenergebnissen ist das Rechnen in Restklassenringen über den kanonischen Homomorphismus:

$$\phi_d : \mathbb{Z} \longrightarrow \mathbb{Z}_d$$

Auch zur Berechnung der Elementarteiler kann es angewendet werden, allerdings sind dazu einige Vorüberlegungen notwendig.

2.1 Das HAVASsche Verfahren

2.1.1 modulare SMITH–Normalform

Wir definieren analog zu Kapitel 1 eine Ähnlichkeitsrelation von $m \times n$–Matrizen über dem Ring $\mathbb{Z}_d$. Bei der analogen Definition einer SMITH–Normalform müssen wir aber beachten, daß es in $\mathbb{Z}_d$ mehr Einheiten gibt als 1 und -1. Da wir die Elementarteiler so definieren wollen, daß sie eindeutig bestimmt sind, haben wir im ganzzahligen Fall gefordert, daß die Elementarteiler alle positiv sein müssen — wir haben den positiven Wert als den **kanonischen** Wert genommen[1]. Da -1 die einzige nichttriviale Einheit in $\mathbb{Z}$ ist, genügt das, um die Eindeutigkeit zu sichern.

Wir betrachten die Abbildung $\psi_d : \mathbb{Z}_d \longrightarrow \mathbb{Z}$, der jedem Element in $\mathbb{Z}_d$ seinen kleinsten positiven Repräsentanten (bezüglich der Ordnung von $\mathbb{Z}$) in $\mathbb{Z}$ zuordnet. Wir bezeichnen in einer Klasse von Werten aus $\mathbb{Z}_d$, die auseinander durch die Multiplikation von Einheiten in $\mathbb{Z}_d$ hervorgehen, nun denjenigen Wert k als **kanonisch**, dessen Wert $\psi_d(k)$ minimal in $\mathbb{Z}$ ist. Wir werden von einem **kanonischen Wert** reden, falls dieser Wert bezüglich irgend einer Klasse von Werten aus $\mathbb{Z}_d$, die auseinander durch die Multiplikation von Einheiten in $\mathbb{Z}_d$ hervorgehen, der kanonische Wert ist. Der **kanonische Wert eines Elementes** ist der kanonische Wert der Klasse, der das Element angehört.

Wir definieren nun analog: Eine Matrix $M = (a_{ij})_{i=1,\ldots,m;j=1,\ldots,n}$ ist nach Definition in der d–**modularen** SMITH–**Normalform** , wenn nur Werte auf der Diagonalen ausgehend vom Matrixeintrag links oben (also die Elemente a_{ii}) bis zu einer bestimmten Zeile r ungleich Null sind, und wenn diese von Null verschieden Diagonalelemente kanonisch in $\mathbb{Z}_d$ sind und ein Diagonalelement einer bestimmten Zeile $i < r$ alle Diagonalelemente der Zeilen mit höherer Zeilennummer $j > i$ teilt. Die Matrix sieht also folgendermaßen aus:

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & & & & & \\ \vdots & & \ddots & \vdots & \vdots & & \\ 0 & & \cdots & a_{rr} & 0 & & \\ 0 & & \cdots & 0 & 0 & & \\ \vdots & & & & & \ddots & \\ 0 & & & & & & \end{pmatrix} \quad \begin{array}{l} r = \text{Rang(M)} \\ a_{ij} = 0,\ \text{falls } i \neq j \\ a_{ii} \neq 0 \text{ kanonisch },\ \text{falls } i \leq r \\ a_{ii} = 0,\ \text{falls } i > r \\ a_{ii} | a_{(i+1)(i+1)} \end{array}$$

[1]Ein Wert auf der Diagonalen kann leicht zu seinem Negativen gemacht werden; man multipliziere einfach -1 zu seiner Zeile.

Die Elemente auf der Diagonalen heißen d–**modulare Elementarteiler**.

2.1.2 Was nutzen modulare Elementarteiler?

Nun stellt sich die Frage, ob wir aus modularen Elementarteilern die ganzzahligen Elementarteiler gewinnen können. In vielen modularen Algorithmen rechnet man im Primkörper $\mathbb{Z}_p$. Doch was würden die modularen Elementarteiler in diesem Restklassenring sein? Da hier alle Zahlen Einheiten sind, würde es nur triviale Elementarteiler geben — und wir können die Frage nicht beantworten, ob es nichttriviale ganzzahlige Elementarteiler gibt.

Wir haben aber folgenden Satz:

Satz 5 ([4]) *Seien $e_1, \ldots, e_r$ die Elementarteiler einer ganzzahligen Matrix M, und d eine ganze positive Zahl, so daß e_r ein echter Teiler von d ist. Dann sind $e_1, \ldots, e_r$ auch die d–modularen Elementarteiler von M, in $\mathbb{Z}_d$ betrachtet.*

Beweis: Sei PMQ die SMITH–Normalform von M (wobei P und Q invertierbare Matrizen sind), seien P_d, M_d und Q_d die gleichen Matrizen im Restklassenring $\mathbb{Z}_d$ betrachtet, dann gilt:

$$PMQ \equiv_d P_d M_d Q_d,$$

weil bei der Matrixmultiplikation sich alle Positionen der Ergebnismatrix mit + und * in den Einträgen der Faktoren schreiben lassen. Weil aber alle Elementarteiler kleiner d sind, folgt daraus

$$PMQ = P_d M_d Q_d.$$

Weil sich auch die Determinante einer quadratischen Matrix mit + und * in den Koeffizenten der Matrix schreiben läßt, ist

$$\pm 1 = \operatorname{Det}(P) \equiv_d \operatorname{Det}_d(P_d)$$

(Det_d bezeichne die Determinante in $\mathbb{Z}_d$) und

$$\pm 1 = \operatorname{Det}(Q) \equiv_d \operatorname{Det}_d(Q_d),$$

damit sind P_d und Q_d in $\mathbb{Z}_d$ invertierbar.

Die Teilbarkeitsvoraussetzung brauchen wir nun, um zu sagen, daß die $e_1, \ldots, e_r$ kanonisch in $\mathbb{Z}_d$ sind, womit der Satz bewiesen ist.

2.1.3 Wie aber den Modul d finden?

Es scheint, wir hätten hier den Teufel mit dem Beelzebub ausgetrieben, denn woher sollen wir den größten Elementarteiler nehmen? Ironischerweise hilft uns hier das hoffnungslos uneffektive Minoren–ggT–Verfahren weiter. Sicherlich ist der größte Elementarteiler ein Teiler des Rangminoren–ggT. Und der ggT *einiger* Rangminoren ist ein Vielfaches des ggT *aller* Rangminoren.

Nun zeigt sich aber folgendes: Das Minoren–ggT–Verfahren ist nur dann so schrecklich uneffektiv, wenn wir die Elementarteiler *genau* aus ihm herausbekommen wollen. Aber wenn wir mit einem Vielfachen zufrieden sind ...

2.1.4 Wie aber die Determinante berechnen?

Doch nicht etwa mit einem ganzzahligen GAUSS–Verfahren ? Die Koeffizentenexplosion, die wir verhindern wollen, würde dann eben hier eintreten. Nein, man kann nämlich auch die Determinante mit einem anderen modularen Verfahren ausrechnen, denn, wie schon erwähnt, gilt für die modulare Determinante $\operatorname{Det}(R) \equiv_m \operatorname{Det}_m(R_m)$ für eine beliebige ganze Zahl m. Das ist besonders praktisch, wenn m eine Primzahl ist, denn dann rechnet man in einem Körper, und in einem Körper kann man den GAUSSschen Algorithmus anwenden.

Nun ist die ganzzahlige Determinante einer Matrix beschränkt durch die

Satz 6 (HADAMARD–**Schranke** , [9]) *Die Determinante einer $n \times n$–Matrix (a_{ij}) ist durch*

$$\prod_{i=1}^{n} \sqrt{\sum_{j=1}^{n} a_{ij}^2}$$

beschränkt.

Wenn also der Modul d größer als die doppelte HADAMARD–Schranke gewählt wurde, dann ist der Repräsentant mit dem kleinsten absoluten Betrag der d–modularen Determinante gleich der ganzzahligen Determinante.

2.1.5 Modulo einer solch riesigen Zahl?

Da kann man doch gleich in den ganzen Zahlen rechnen! Nein, denn man benutzt den **chinesischen Restklassensatz,** mit dem man die Ergebnisse der Determinantenberechnung modulo verschiedener Zahlen kombinieren kann:

Satz 7 (chinesischer Restklassensatz, [9]) *Seien $m_1, \ldots, m_k$ teilerfremde positive ganze Zahlen, $r_1, \ldots, r_k$ ganze Zahlen. Dann gibt es eine ganze Zahl r die das Kongruenzsystem*

$$\begin{gathered} r \equiv_{m_1} r_1 \\ r \equiv_{m_2} r_2 \\ \vdots \\ r \equiv_{m_k} r_k \end{gathered}$$

erfüllt. Diese Zahl r ist modulo dem kleinsten gemeinsamen Vielfachen (kgV) von $m_1, \ldots, m_k$ eindeutig bestimmt.

Wir nehmen jetzt also Primzahlen von einer angenehmen Größe, deren Produkt das Doppelte der HADAMARD–Schranke übersteigt. Wir berechnen die Determinante modulo dieser Primzahlen, wenden den Restklassensatz an und erhalten unsere ganzzahlige Determinante.

2.1.6 Woher den Rangminor nehmen?

Richtig, wir müssen zuerst den Rang wissen, bevor wir die Determinantenberechnung anfangen können. Ein Spezialfall der Beziehung über die modulare Determinante besagt:

Satz 8 *Sei p eine Primzahl, M eine ganzzahlige Matrix. Dann ist der Rang von M in $\mathbb{Z}$ nicht kleiner als der von M_p in $\mathbb{Z}_p$. Der modulare Rang ist gleich dem ganzzahligen, wenn p nicht ein Teiler aller Rangminoren ist.*

Der Rang ist also bestimmt, wenn er modulo sovieler Primzahlen bestimmt ist, daß deren Produkt die HADAMARD–Schranken aller Rangteilmatrizen[2] übersteigt. Und diese Schranken sind natürlich alle beschränkt durch die (verallgemeinerte) HADAMARD–Schranke der gesamten Matrix.

Fein raus sind wir, wenn der Rang modulo einer Primzahl schon der maximal mögliche ist (was durch Zeilen– oder Spaltenzahl gegeben ist), dann haben wir mit dem Satz sofort den ganzzahligen Rang und können zur Determinantenberechnung schreiten.

2.1.7 Warum denn nicht alles mit Primzahlen?

Diese Frage drängt sich den meisten auf, die das Verfahren zum ersten Mal sehen. Kann man nicht auch die Elementarteiler mit dem chinesischen Restklassensatz berechnen? Nein, in Primzahlkörpern sind z. B. immer alle Elementarteiler trivial, weil alle Reste Einheiten sind.

Doch die Ursache, daß man nicht irgendwelche Modulzahlen für die modulare Elementarteilerberechnung nehmen kann, liegt in folgender einfachen Tatsache: **Die Determinante kann man mit Operationen in den Koeffizenten der Matrix schreiben, die mit dem kanonischen Homomorphismus ϕ_d verträglich sind (nämlich + und *), die Elementarteiler jedoch nicht.**

[2] quadratische Teilmatrizen der Größe $r \times r$, wobei r der Rang der Matrix

Wenn wir den ganzahligen GAUSS–Algorithmus als Ausdruck in den Koeffizenten der Matrix schreiben wollen, so kommen wir nicht mit Addition und Multiplikation aus, denn innerhalb des Algorithmus kommt häufig der Test auf die Gleichheit vor. Wir brauchen also den KRONECKER–Operator:

$$\delta_{xy} := \begin{cases} 1 & \text{falls } x = y \\ 0 & \text{sonst} \end{cases}$$

Wie man sich leicht klarmacht, ist der KRONECKER–Operator mit dem kanonischen Homomorphismus ϕ_d nicht verträglich.

2.2 Modulare GAUSS–Algorithmus –Algorithmen

Jetzt habe wir uns also davon überzeugt, daß es möglich ist, einen geeigneten Modul zu finden, für den die modulare SMITH–Normalform gleich der ganzzahlige SMITH–Normalform ist. Wie können wir aber nun die modulare SMITH–Normalform überhaupt ausrechnen?

2.2.1 GAUSS–Algorithmus in $\mathbb{Z}_d$

Der GAUSS–Algorithmus ist in seiner ursprünglichen Form nur für ganze Zahlen definiert. Man kann jedoch in $\mathbb{Z}_d$ ein ähnliches Verfahren definieren, das die modularen Elementarteiler berechnet. Man benutzt ebenfalls elementare Operationen:

- Vertauschen zweier Zeilen/Spalten
- Multiplizieren einer Zeile/Spalte mit einer Einheit von $\mathbb{Z}_d$
- Addieren eines Vielfachen einer Zeile/Spalte zu einer anderen

Es ist leicht zu sehen, daß die Anwendung einer dieser Operationen eine Matrix in eine ähnliche Matrix überführt. Man kann zeigen, daß sich jede in $\mathbb{Z}_d$ invertierbare Matrix als Produkt der entsprechenden d–modularen Elementarmatrizen schreiben läßt.

Analog zum ganzzahligen GAUSS–Verfahren wählen wir einen von (der Restklasse) Null verschiedenen Pivot. Wenn ein Wert in der Pivotzeile oder –spalte vom Pivotwert geteilt wird, dann können wir ihn mit einer elementaren Operation eliminieren.

Wenn der Pivotwert aber einen Wert in der Pivotzeile– oder Spalte nicht teilt, dann können wir ihn durch die Anwendung einer elementaren Operation verkleinern.

Die Schwierigkeit besteht darin, daß Größenbeziehungen zwischen Restklassen in $\mathbb{Z}_d$ nicht von vornherein definiert sind. Zwei Möglichkeiten sind sinnvoll:

- Man vergleicht die Restklassen anhand des kleinsten positiven Repräsentanten.
- Man vergleicht die Restklassen anhand des kleinsten absoluten Betrages des positiven oder negativen Repräsentanten, wobei ein positiver Wert als kleiner angesehen wird als der negative mit gleichem absoluten Betrag.

Nun können wir eine Restdivision anhand dieser Vergleichsrelation definieren. Man macht sich klar, daß bei beiden Möglichkeiten die Restdivision nun wohldefiniert ist, d. h.

$$\forall a, b \in \mathbb{Z}_d, b \neq 0 : \exists r, q \in \mathbb{Z}_d : a = bq + r, \, r < b.$$

Der Beweis beruht nun darauf, daß das q zunächst mit einer Restdivision in $\mathbb{Z}$ bestimmt wird und danach die Gültigkeit der geforderten Eigenschaft in Z_d gezeigt wird.

Weil in beiden Varianten der Größenbeziehung zwischen Restklassen 1 das kleinste Element ungleich Null ist, entsteht nach einer endlich langen Folge von elementaren Operationen ein Element, das alle Einträge der Matrix teilt.

Es kann nun sein, daß dieser Wert noch nicht der kanonische Wert ist; er kann aber leicht durch eine entsprechende Zeilenmultiplikation zu diesem gemacht werden. Eine Verbesserung des Algorithmus kann erreicht werden, wenn nach der Wahl des Pivots dieser zunächst mit der Multipliklation der Pivotzeile mit der entsprechenden Einheit in $\mathbb{Z}_d$ zu seinem kanonischen Element reduziert wird.

2.2.2 Gauss–Algorithmus in $\mathbb{Z}_{p^n}$

Der modulare Gauss–Algorithmus in $\mathbb{Z}_{p^n}$ kann effektiver programmiert werden als der in $\mathbb{Z}_d$.

Man macht sich dabei zunutze, daß man hier eine Zeile mit einem Faktor, der zu p^n und damit zu p teilerfremd ist, multiplizieren darf, da diese Elemente im Ring $\mathbb{Z}_{p^n}$ Einheiten sind. Man beachte, daß wir in $\mathbb{Z}_d$ nicht so leicht nachprüfen konnten, ob ein Element eine Einheit ist.

Man geht folgendermaßen vor:

Algorithmus 2 (Gauss–Algorithmus modulo Primzahlpotenz)

1. Wenn die Matrix aus lauter Nullen besteht, dann ist man fertig. Sonst nehme man $m := p^n$ als aktuellen Modul, eine leere Liste L und $e := 0$ als aktuellen Potenzzähler.
2. Wenn die Matrix nur aus durch p teilbaren Elementen besteht, dann dividiere man die gesamte Matrix durch p, inkrementiere e, dividere m durch p.
3. Man wähle ein beliebiges nicht durch p teilbares Element der Matrix als Pivot und vertausche Zeilen und Spalten, so daß es an der Position links oben steht.
4. Man invertiere das Pivotelement in $\mathbb{Z}_{p^n}$ und multipliziere die Pivotzeile mit dem Inversen, so daß an der Pivotposition jetzt eine 1 steht.
5. Man eliminiere jetzt jeden Eintrag in der Pivotzeile und der Pivotspalte durch Addieren eines geeigneten Vielfachen der Pivotspalte bzw. Pivotspalte.
6. Man merke sich p^e als einen Elementarteiler in L. Man betrachte jetzt die Submatrix, die durch Streichen der Pivotzeile und Pivotspalte entsteht. Wenn sie nicht aus lauter Nullen besteht, dann gehe damit zu 2.
7. Man hat nun alle p^n–modularen Elementarteiler in der Liste L gesammelt.

Der Algorithmus gibt uns die p^n–modularen Elementarteiler. Das macht zunächst nur Sinn, wenn der ggT der Rangminoren zufällig eine Primzahlpotenz ist. Ich komme aber später darauf zurück, die p^n–modularen Elementarteiler zu nutzen, wenn der ggT der Rangminoren nicht eine Primzahlpotenz ist.

2.2.3 Gaussscher Algorithmus in $\mathbb{Z}_p$

Wie schon gesagt, genügt für die modulare Determinantenberechnung irgendein beliebiger Modul, und da sind Primzahlmoduln am besten geeignet, da in Primzahlkörpern alle Elemente außer Null Einheiten sind, wir also nach Herzenslust dividieren können.

Algorithmus 3 (Gaussscher Algorithmus in Primkörpern)

1. Wenn die Matrix aus lauter Nullen besteht, dann ist man fertig. Man setze eine Variable d auf 1.
2. Man wähle ein beliebiges von Null verschiedenes Element der Matrix als Pivot und vertausche Zeilen und Spalten, so daß er an der Position links oben steht. Bei jeder Vertauschung multipliziert man d mit -1.
3. Man multipliziere den Wert des Pivotelements zu d.
4. Man invertiere das Pivotelement in $\mathbb{Z}_p$ und multipliziere die Pivotzeile mit dem Inversen, so daß an der Pivotposition jetzt eine 1 steht.
5. Man eliminiere jetzt jeden Eintrag in der Pivotzeile und der Pivotspalte durch Addieren eines geeigneten Vielfachen der Pivotspalte bzw. Pivotspalte.
6. Man betrachte jetzt die Submatrix, die durch Streichen der Pivotzeile und Pivotspalte entsteht. Wenn sie nicht aus lauter Nullen besteht, dann gehe damit zu 2.
7. In d steht nun die p–modulare Determinante der Matrix.

2.3 Verfeinerungen

2.3.1 *einige* Rangminoren

Zunächst sind wir davon ausgegangen, irgendwelche verschiedenen Rangminoren herauszupicken, also extra für jede Rangteilmatrix die Determinante zu berechnen. In [4] wurde mit einigen Beispielen schnell klar gemacht, warum nicht etwa schon ein Minor genügt, er wäre viel zu groß. Doch schon der ggT von einigen wenigen Minoren war klein genug, um ihn als Modul für die modulare Elementarteilerberechnung zu akzeptieren.

Der Trick, daß wir trotzdem für einen Modul im wesentlichen nur eine Determinante berechnen, geht so: Wir bestimmen nicht völlig verschiedene Determinanten von Rangteilmatrizen, sondern von solchen, die bis auf die letzte Zeile identisch sind. So braucht man all jene Schritte des GAUSS–Verfahrens , die eine Diagonalform von den identischen Zeilen berechnen, nicht für jeden Rangminor zu wiederholen. Nur die Berechnungen für die letzte Zeile einer jeden Rangteilmatrix müssen neu gemacht werden. Man sieht aber leicht ein, daß das nur ein winziger Bruchteil der gesamten Diagonalisierung ist. Die verschiedenen Rangminoren entstehen so faktisch gleichzeitig.

Es stellt sich heraus, daß es im allgemeinen genug Rangteilmatrizen mit der geforderten Eigenschaft gibt, wenn es überhaupt mehrere Rangminoren gibt.

2.3.2 Verkleinerung des Elementarteiler–Moduls

Sei jetzt d die mithilfe der Determinanten errechnete Modul.

Man kann das GAUSS–Verfahren modulo d aufteilen, indem man d faktorisiert. Man berechnet die Elementarteiler modulo aller Primzahlpotenzen der Faktorisierung von d und bildet dann das elementweise Produkt dieser modularen Elementarteiler, um die d–modularen und damit die ganzzahligen Elementarteiler herauszubekommen. Die Grundlage dazu bildet folgender Satz:

Satz 9 *Sei p eine Primzahl, f eine natürliche Zahl, so daß p^f kein Teiler des größten Elementarteilers ist. Dann teilen die p^f–modularen Elementarteiler die ganzzhaligen Elementarteiler einer belibigen Matrix. Wenn also M die Matrix ist, M_{p^f} die Matrix in $\mathbb{Z}_{p^f}$, $d_1, \ldots, d_r$ die Elementarteiler von M und $\dot{d}_1, \ldots, \dot{d}_r$ die p^f–modularen Elementarteiler von M_{p^f}, dann gilt für $i = 1, \ldots, n : \dot{d}_i | d_i$ in $\mathbb{Z}$.*

Beweis: Sei PMQ die SMITH–Normalform von M, dann sind P und Q nicht nur in $\mathbb{Z}$ invertierbar, sondern auch die entsprechenden Matrizen P_{p^f} und Q_{p^f} in $\mathbb{Z}_{p^f}$. Damit ist $D =_{\text{Def}} P_{p^f} M_{p^f} Q_{p^f}$ eine zu M_{p^f} ähnliche Matrix. D ist nun eine Diagonalmatrix, in deren Hauptdiagonale $\bar{d}_1, \ldots, \bar{d}_r$ stehen; die ganzzahligen Elementarteiler $d_1, \ldots, d_r$ im Ring $\mathbb{Z}_{p^f}$ betrachtet. Die Werte $\bar{d}_1, \ldots, \bar{d}_r$ sind aber nun noch nicht notwendigerweise kanonisch in $\mathbb{Z}_{p^f}$.

Es ist leicht einzusehen, daß in $\mathbb{Z}_{p^f}$ gerade die p–Potenzen p^e für $e = 0, \ldots, f-1$ die kanonischen Werte sind, und daß der kanonische Wert eines gewissen Elementes a genau der p–Anteil in der Primzahlzerlegung von a, in $\mathbb{Z}$ betrachtet, ist, d. h. p^e, wenn $a = bp^e$, wobei b und p teilerfremd sein sollen. Daraus folgt aber, daß der bezüglich $\mathbb{Z}_{p^f}$ kanonische Wert eines Elements a ein Teiler von a, in $\mathbb{Z}$ betrachtet, ist.

Wir können nun in der Diagonalmatrix mit den Diagonalelementen $\bar{d}_1, \ldots, \bar{d}_r$ die Werte $\bar{d}_i$ sukzessive mithilfe elementarer Zeilenoperationen durch ihre kanonischen Werte ersetzen — das sind aber genau die p^f–modularen Elementarteiler.

Habe nun ein Elementarteiler d_i einen p–Anteil von p^e. Nach Voraussetzung ist e kleiner als f. Darum hat auch $\bar{d}_i$ einen p–Anteil von p^e. Da nun aber der p^f–modulare Elementarteiler $\dot{d}_i$ der p–Anteil p^e von $\bar{d}_i$ ist, ist der Satz bewiesen.

2.3.3 verbesserte HADAMARD–Schranken

Die HADAMARD–Schranke als obere Schranke für den absoluten Betrag der Determinante dieser Matrix haben wir zunächst mit

$$\prod_{i=1}^{n} \sqrt{\sum_{j=1}^{n} a_{ij}^2}$$

angegeben.

Wenn wir uns überlegen, daß eine transponierte Matrix ganz genau die selbe Determinante hat wie die ursprüngliche, dann ist die Determinante einer Matrix genauso gut durch

$$\prod_{i=1}^{n} \sqrt{\sum_{j=1}^{n} a_{ji}^2}$$

beschränkt, wobei man nicht sagen kann, welche von beiden kleiner ist; da sie aber beide gelten, kann man das Minimum von beiden nehmen, und hat u. U. schon einiges gewonnen.

Diese HADAMARD–Schranken galten für quadratische Matrizen von vollem Rang. Für die obere Schranke der absoluten Beträge aller Minoren einer rechteckigen $m \times n$–Matrix haben wir bisher die verallgemeinerte HADAMARD–Schranke

$$\prod_{i=1}^{m} \sqrt{\sum_{j=1}^{n} a_{ij}^2}$$

(wobei alle Nullzeilen vorher gestrichen werden) benutzt, die offensichtlich alle Minoren beschränkt, und dabei Unmengen verschenkt. Wenn wir uns die Formel einmal genauer ansehen, dann stellen wir fest, daß ein Produkt von euklidischen Zeilennormen gebildet wird.

Nehmen wir nun o. B. d. A. an, daß $m > n$. Dann brauchen wir eigentlich nur n–Minoren abzuschätzen, denn größere Teilmatrizen als $n \times n$ gibt es gar nicht. Also brauchen wir nur das Maximum der HADAMARD–Schranken aller möglichen n–Teilmatrizen abzuschätzen. Und das ist durch das größtmögliche Produkt von n Zeilennormen möglich. Wir sortieren also die Normen der Größe nach und nehmen die n größten und haben so schon eine viel kleinere Schranke gewonnen:

$$\sqrt{\prod \mathrm{Maximum}_n \left\{ \sum_{j=1}^{n} a_{ij}^2 \,\middle|\, 1 \le i \le m \right\}}$$

wobei $\mathrm{Maximum}_n$ für die Menge der größten n Zahlen der Menge stehen soll.

Nun können wir den Spieß auch wieder umdrehen und Spaltennormen betrachten. Und da gibt es folgendes Verfahren: Man sortiert die Einträge jeder Spalte der Größe ihres absoluten Betrages nach. Dann nehme man in jeder Spalte die n größten Werte und bildet die euklidische Spaltennorm davon. Dann nimmt man das Produkt der Spaltennormen und hat eine weitere verbesserte Schranke:

$$\prod_{j=1}^{n} \sqrt{\sum \mathrm{Maximum}_n \left\{ a_{ij}^2 \,\middle|\, 1 \le i \le m \right\}}$$

Aber zahlt sich das bei der ganzen Sortiererei im Endeffekt überhaupt aus? Dazu verdeutliche man sich, daß ein modularer GAUSS–Algorithmus bereits kubische Komplexität hat, eine Sortieroperation aber nur $\mathrm{O}(n \log n)$, insgesamt also $\mathrm{O}(n^2 \log n)$ (wenn man n und m als von der gleichen Größenordnung betrachtet). Und wirklich, in der Praxis ist diese beschriebene HADAMARD–Schranken–Berechnung viel schneller als ein modularer GAUSS–Algorithmus , und in vielen Fällen werden etliche davon eingespart.

Vielleicht macht es dann Sinn, noch einen Schritt weiter zu gehen, und die Hadamard–Schranke von jeder einzelnen n–Teilmatrix zu berechnen, und dann das Maximum zu nehmen? Da ist aber das Maß überschritten, denn es gibt $\binom{n}{m}$ solche Minoren, und das ist nun wirklich zu viel.

Wenn man nun den Rang r schon von vornherein weiß, kann man dieses Verfahren noch weiterführen, indem man nur die r maximalen Zeilen oder die r maximalen Spalten berücksichtigt.

2.3.4 Wurzel vom Produkt oder umgekehrt?

Bei der Berechnung der HADAMARD–Schranke tritt folgende Frage auf:

> Man hat die Wurzel von einem Produkt von Zahlen zu berechnen. Nach einer einfachen Rechenregel ist es gleich, zuerst die Zahlen zu multiplizieren, und danach die Wurzel aus dem Produkt zu ziehen oder zuerst die Wurzel aus jeder einzelnen Zahl zu ziehen, und danach das Produkt dieser Wurzeln zu berechnen. Bei welchen Zahlen ist das eine oder das andere günstiger.

Folgende Punkte sprechen dafür, die Wurzel aus dem Produkt ziehen:

- In GAP berechne ich anstatt der genauen Wurzel die kleinste ganze Zahl, die nicht kleiner als die reelle Wurzel ist. Damit ich also nie eine zu kleine HADAMARD–Schranke bekomme, muß ich bei jedem Wurzelziehen aufrunden. Damit meine ganzzahlige HADAMARD–Schranke nicht allzu sehr über der reellen HADAMARD–Schranke liegt, ist es günstig, so selten wie möglich die Wurzel zu ziehen.
- Das Wurzelziehen ist eine relativ teure Funktion, auch darum sollte man sie nicht zu oft ausführen.

Und dagegen spricht:

- Langzahlarithmetik dauert sehr lange, darum rechne man mit möglichst kleinen Zahlen. Die Zeit für eine Multiplikation steigt quadratisch mit der Zahlenlänge.

Daraus folgt, daß es umso besser ist, das Produkt aus den Wurzeln zu ziehen, je größer die Zahlen sind.

Ich habe die Laufzeit in GAP getestet und dabei (mit erheblichen Schwankungen) festgestellt, daß es erst ab einer Zahlengröße von 2^{1000} besser ist, daß Produkt der Wurzeln zu berechnen.

2.3.5 den Rang erraten

Wenn man immer artig für die Rangberechnung so viele Primzahlen genommen hat, wie die HADAMARD–Schranke vorschreibt, und bei jeder Primzahl immer der selbe Rang herauskommt, und das bei allen normalen Beispielen, dann fragt man sich zu Recht, ob das mit der HADAMARD–Schranke nicht ein wenig übertrieben ist.

Richtig, damit hier bei der Rangberechnung mit einer Primzahl etwas schiefläuft, muß diese Primzahl den Rangminoren–ggT, also wirklich jeden einzelnen Rangminor teilen. Bei Primzahlen der Größe 2^{14}, die wir benutzen[3], ist die Fehlerwahrscheinlichkeit sehr gering. Aber ganz sicher ist es eben nicht ...

Wenn man also mit einem kleinen Fehlerrisiko einverstanden ist, dann kann man den Rang modulo einer Primzahl als den ganzzahligen Rang ansehen und damit weiterrechnen.

2.3.6 Rang raten und prüfen

Wie schon gesagt, können wir fast sicher sein, daß der modulo einer Primzahl errechnete Rang r schon der ganzzahlige Rang ist. Wenn wir aber diese winzige Restunsicherheit nicht akzeptieren wollen, dann müssen wir auf die HADAMARD–Schranke zurückgreifen, und den Rang modulo sovieler Primzahlen prüfen, daß deren Produkt die HADAMARD–Schranke übersteigt.

Da wir nun sowieso nicht annehmen, daß der Rang größer als r ist, **dann genügt es, die HADAMARD–Schranke bezüglich aller $r+1$–Teilmatrizen zu nehmen, um zu beweisen, daß kein $r+1$–Minor ungleich Null ist.**

Wenn der Rang bezüglich keiner der Primzahlen, deren Menge der HADAMARD–Schranke entspricht, größer als der zuerst berechnete ist, so ist dieser Rang richtig und wir sind fertig. Wenn nicht, dann müssen wir eine neue Schranke berechnen, die auf dem höheren Rang beruht. Dazu ist es wichtig, die Ergebnisse bezüglich aller Primzahlen zu speichern, damit nicht alle Ergebnisse noch einmal berechnet werden müssen.

2.3.7 die Determinante erraten

Auch für die Determinante gibt es ein Verfahren, daß wir nicht mit unseren Primzahlen bis zur HADAMARD–Schranke gehen müssen, sondern schon vorher abbrechen können, wenn wir eine gewisse Restunsicherheit akzeptieren.

Beim bisherigen Verfahren haben wir einmal am Schluß mit dem chinesischen Restklassensatz die ganzzahlige Determinate ausgerechnet. Hier jedoch tun wir das nach jeder neu berechneten modularen Determinante. Wenn sich nach der Berechnung einer neuen modularen Determinante keine andere vorläufige ganzzahlige Determinante ergibt als bei der modularen Determinante davor, so kann man mit sehr hoher Wahrscheinlichkeit davon ausgehen, daß sie sich auch in Zukunft nicht mehr ändern wird, weil sie schon

[3] Das Produkt zweier Zahlen dieser Größenordnung paßt noch in ein Maschinenwort, darum sind die Berechnungen mit ihnen sehr schnell.

die korrekte ganzzahlige Determinante ist. Das kann aber natürlich auch Zufall sein, und es besteht die Möglichkeit, daß wir uns irren.

Man kann die Fehlerwahrscheinlichkeit folgendermaßen abschätzen: Ich betrachte hier zunächst nur den Fall, daß man eine einzige Determinante modular berechnet. Wenn man *einige* Determinanten gleichzeitig berechnet, und sie nur dann akzeptiert, wenn sich *alle* Determinanten bezüglich dieses Moduls nicht ändern, dann ist die Fehlerwahrscheinlichkeit natürlich noch viel kleiner als die Fehlerwahrscheinlichkeit für eine Determinante[4].

Sei also D die Determinante und $p_1, \dots, p_l$ Primzahlen, so daß $\prod_{i=1}^{l-1} p_i \leq 2|D| < \prod_{i=1}^{l} p_i$, d.h., daß man mit diesen Primzahlen die Determinante richtig bestimmen kann. Wir betrachten nun die Zerlegung von D als Zahlendarstellung mit $D = d_1 + p_1(d_2 + p_2(d_3 + \cdots + p_{l-1}d_l))$ wobei $-\frac{p_i}{2} < d_i < \frac{p_i}{2}$. Diese Zahlendarstellung ist eindeutig.

Wenn man die Determinante modulo $\prod_{i=1}^{k} p_i$ (mithilfe der Determinanten modulo $p_1, \dots, p_l$ und des chinesischen Restklassensatzes) berechnet, dann kommt man gerade auf die Zahl $d_1 + p_1(d_2 + p_2(d_3 + \cdots + p_{k-1}d_k))$. Damit jetzt das Verfahren an dieser Stelle k irregeleitet wird, die vorläufige ganzzahlige Determinante nach der Berechnung der k–ten modularen Determinante also gleich der vorläufigen ganzzahligen Determinante nach der Berechnung der $k-1$–ten modularen Determinante ist, obwohl sie noch nicht die richtige Determinante D ist, muß $d_k = 0$ sein.

Da die Determinanten nichts weiter mit den Primzahlen zu tun haben, kann man die d_i als unabhängig zu den p_i ansehen, unter dieser Voraussetzung gilt aber $d_k = 0$ mit der Wahrscheinlichkeit $\frac{1}{p_k}$.

Zwei Bemerkungen sind dazu noch zu machen:

- Die erste ist, daß diese eben errechnete Wahrscheinlichkeit entscheidend größer ist, als die Fehlerwahrscheinlichkeit der unsicheren Rangberechnung. Dort mußten *alle* Rangminoren durch die entsprechende Primzahl teilbar sein, um einen Fehler heraufzubeschwören, hier geht es um *einen* Minor.

- Die zweite Bemerkung ist zur Möglichkeit zu machen, einen Fehler zu erkennen. Bei der unsicheren Rangberechnung kann der Fehler bemerkt werden, wenn bei der modularen Elementarteilerberechnung noch ein weiterer Modul berücksichtigt wird, und sich dann bezüglich diesem ein anderer Rang herausstellt.

  Hingegen besteht bei der unsicheren Determinantenberechnung nicht die Möglichkeit, daß der Fehler *erkannt* wird, dafür ist es aber nicht unwahrscheinlich, daß er sich nicht weiter *auswirkt*. Nämlich dann, wenn das berechnete Rangminoren–ggT trotzdem ein Vielfaches des größten Elementarteilers ist.

2.3.8 Faktorisieren

Eine nicht zu unterschätzende Schwierigkeit ist das Faktorisieren des Rangminoren–ggTs. Ich benutze hier die schon in GAP implementierte Funktion; zuweilen ist ihr die Zahl zu groß, und sie bricht nach einer Weile die Berechnung ab.

Man kann natürlich nun einfach den modularen Elementarteileralgorithmus modulo dieser nicht zu faktorisierenden Zahl ausführen; das Faktorisieren war zunächst nur als mögliche Verbesserung gedacht. Doch daran wird man nicht allzu viel Freude haben, denn dieser große Modul hilft nun nicht mehr gegen die Koeffizentenexplosion. Die Rechnungen in großen Restklassenringen ist genauso langsam wie das Rechnen mit langen Zahlen; und es kann vorkommen, daß eine modulare Zwischenmatrix die Speicherkapazität des Computers erschöpft.

Wir können einen anderen Weg beschreiten: Da der Rangminoren–ggT gleich dem Produkt aller Elementarteiler ist, wird unser Modul umso kleiner, je mehr Rangminoren wir berechnen, um deren ggT zu bilden. Wir können also probieren, noch mehr Rangminoren nehmen, und deren ggT zu bilden.

In einer kritischen Situation sind wir, wenn es nicht mehr möglich ist, noch mehr Rangteilmatrizen unter der Voraussetzung zu finden, daß sie bis auf die letzte Zeile identisch sein sollen. Man kann sich natürlich auf den Standpunkt stellen, daß wir auf die gesamte Faktorisierung verzichten (damit auch auf die Sicherheit des Ergebnisses), und nur Primteiler bis zu einer bestimmten Größe berücksichtigen. Doch ich denke, wenn in einem solchen Fall das richtige Ergebnis erzielt wird, dann wäre auch der ggT von genügend vielen Rangminoren klein genug gewesen.

[4]Wenn die Determinanten unabhängig wären, würde sie sich bei jeder Determinante multiplizieren, doch davon kann nicht ausgegangen werden. Die Größenbeziehung der Wahrscheinlichkeiten läßt sich aber leicht zeigen

Unter Umständen ist hier der Punkt erreicht, wo das modulare Verfahren versagt und wir uns auf die Begrenzung der Koeffizentenexplosion in nichtmodularen Verfahren konzentrieren müssen.

2.4 Verfeinerungen des primzahlmodularen Algorithmus

2.4.1 Kombinierte Rang– und Determinantenberechnung

Bisher waren Rang berechnen, Rangteilmatrizen suchen und Determinanten berechnen noch unabhängige Operationen, die hintereinander ausgeführt werden. Da sie im wesentlichen dieselbe Berechnung beinhalten (nämlich einen GAUSSschen Algorithmus), lassen sie sich kombinieren. Dabei müssen einige Dinge berücksichtigt werden — darum gebe ich einen veränderten Grundalgorithmus an, der sich fast nur in der Formulierung von der einfachen Variante unterscheidet:

2.4.2 verbesserte Determinantenberechnung

Sei o. B. d. A. die Zeilenzahl größer als die Spaltenzahl.

Algorithmus 4 (verbesserter GAUSS–Algorithmus modulo Primzahl)

1. Wenn die Matrix aus lauter Nullen besteht, dann ist man fertig. Sonst setze man eine Variable d (für die Determinante) auf 1. Man setze eine Permutations–Variable σ auf die triviale Permutation. T sei eine noch leere Liste (von Vektoren), t (als die Länge dieser Liste) werde auf Null gesetzt. Die aktuelle Zeile Z ist die erste Zeile der Matrix.

2. Wenn die aktuelle Zeile Z nur aus Nullen[5] besteht, dann nehme man die nächste Zeile der Matrix als aktuelle Zeile; das wiederhole man so lange, bis man eine Zeile gefunden hat, die nicht aus lauter Nullen besteht. Ist die Matrix abgearbeitet, dann gehe zu 10.

3. Man wende die Permutation σ auf die Positionen von Z an.

4. Man eliminiere die ersten t Elemente von Z mithilfe der Vektoren in T. (Wenn t noch Null ist, entfällt dieser Schritt.) Da an der i–ten Position des i–te Vektors in T eine 1 steht, braucht dazu nur ein geeignetes Vielfaches des i–ten Vektors in T von Z abgezogen zu werden.

5. Man wähle ein beliebiges von Null verschiedenes Element von Z als Pivot und vertausche dessen Position mit dem Element an der $t{+}1$–ten Position[6], wenn es nicht schon an dieser Position gestanden hat. Falls eine Vertauschung stattgefunden hat, multipliziere man die entsprechende Transposition zu der Permutation σ.

6. Man multipliziere den Wert des Elements an der $t+1$–ten Position zu d in $\mathbb{Z}_p$.

7. Man invertiere das Elements an der $t+1$–ten Position in $\mathbb{Z}_p$ und multipliziere Z mit dem Inversen, so daß an der $t+1$–ten Position jetzt eine 1 steht.

8. Man hänge Z an das Ende der Liste T an und inkrementiere t.

9. Wenn t kleiner als die Spaltenzahl ist, gehe man zu 2.

10. Der p–modulare Rang der Matrix ist t. Ein nichtsingulärer t–Minor entsteht aus all den Zeilen, die in T eingegangen sind, und den Spalten, die unter Berücksichtigung der Spaltenpermutation σ als die jeweils ersten von Null verschiedenen Werte in den Vektoren von T hervorgegangen sind. Man erhält die Spaltenindices, indem man die Permutation σ auf die Zahlen $1, \ldots, t$ anwendet. Die p–modulare Determinante genau dieses Minors[7] steht jetzt in d.

[5] im Körper $\mathbb{Z}_p$. In Wirklichkeit müssen wir zunächst jeden Wert der Zeile durch seinen Rest modulo p ersetzen — das nenne ich Kanonisieren

[6] entweder nimmt man das erste von Null verschiedene Element von Z oder man sucht zunächst eine 1 in der Zeile um ggf. die Multiplikation im folgenden Schritt zu sparen.

[7] dessen Spalten aus den Spalten der Ursprungsmatrix mit Indices, die nicht der Größe nach geordnet sind, sondern in der Reihenfolge stehen, wie die Anwendung der Permutation σ auf die Zahlen $1, \ldots, t$ ergibt.

Man beachte, daß es für die Determinatenberechnung ausreicht, eine Dreiecksform mit Diagonalelementen 1 zu bilden. Ein Vorteil dieser zeilenweisen Vorgehensweise ist, daß nicht am Anfang eine Kopie der Matrix angelegt werden muß, sondern daß die triangonalisierte Normalform der Matrix sukzessive aufgebaut wird. Für die Berechnung der Determinante einer nichtsingulären quadratischen Matrix hat dieses Vorgehen noch keinen rechten Sinn, denn genau besehen werden exakt die gleiche Additionen und Multiplikationen ausgeführt wie bei der ersten Form des Algorithmus. Das wird aber anders, wenn der Rang noch nicht bekannt ist, wir also einen möglichst großen nichtsingulären Minor suchen, und ganz besonders für Matrizen, deren Zeilenzahl erheblich von der Spaltenzahl abweicht.

2.4.3 schmale Matrizen

Für die modulare Rang– und Determinantenberechnung von Matrizen, deren Zeilenzahl erheblich von der Spaltenzahl abweicht und die vollen Rang haben, kann man es sich häufig sparen, einen Teil der Matrix überhaupt anzusehen. Ich nehme wieder o. B. d. A. eine Matrix, deren Zeilenzahl nicht kleiner als die Spaltenzahl ist, und durchmustere sie zeilenweise. Wenn ich dabei von Null verschiedene Minoren von der Größe der Spaltenzahl finde, brauche ich den unteren Teil der Matrix nicht zu bearbeiten, ich berechne nur Rangminoren im oberen Teil der Matrix. Diese Vorgehensweise hat einen potentiellen Nachteil: Es könnte sein, daß im unteren Teil der Matrix, den ich mir nun überhaupt nicht angesehen habe, ein Rangminor zu finden wäre, der meinen ggT der Rangminoren noch verkeinert hätte. Im der Praxis fällt dieser Nachteil aber kaum ins Gewicht, zumal in der letztendlichen Version sowieswo nur eine begrenzte Zahl von Rangminoren berechnet wird.

2.4.4 Einsammeln der verschiedenen Rangminoren

Der Algorithmus 4 habe nun eine Rangteilmatrix herausgesucht. Wie bekommen wir aber nun weitere in der gewünschten Form, nämlich, daß sie alle Zeilen gemeinsam haben außer die letzte? Dazu sind zwei Fälle zu unterscheiden:

voller Rang

Der Algorithmus hat (möglicherweise) mitten in der Matrix abgebrochen, und seine Rangteilmatrx ausgegeben. Nun können wir einfach so tun, als hätte es die letzte Zeile gar nicht gegeben: Wir entfernen die letzte Zeile aus T, zetzen t eins zurück, nehmen auch die eventuelle Transposition aus der Permutation σ wieder heraus und setzen auch d wieder auf seinen alten Wert. Dann nehmen wir die nächste Zeile der Matrix als aktuelle Zeile, und bekommen so ganz leicht eventuell neue Rangteilmatrizen mit der geforderten Eigenschaft.

kleinerer Rang

Hier ist das Zurücksetzen nicht so einfach, denn wir sind jetzt schon am Ende der Matrix. Wir müssen uns erinnern, welches die letzte Zeile Z war, die zum Rang beigetragen hat, und setzen diese wie eben beschrieben zurück. Wir müssen also zu jedem Zeitpunkt genau den Zustand noch wissen, **der beim Finden der letzten Zeile Z, die zum Rang beigetragen hat, geherrscht hat.**

Schwierigkeit

Nun finden wir vielleicht noch eine weitere Zeile Z', die einen Rangminor bilden kann. Sie wird mit den noch in T verbliebenen Vektoren reduziert, d. h. alle Positionen $1, \ldots, t-1$ werden zu Null gemacht. Wir stellen nun aber fest, daß Z' an der Spaltenposition t, wo in Z der von Null verschiedenen Wert stand, Null ist. Wenn wir nun die Zeile Z' nur auf den Spalten, die unsere erste Rangteilmatrix bilden, betrachten, stellen wir fest, daß Z' gar nicht unabhängig zu den ersten $t-1$ Zeilen der Teilmatrix ist, und gar keinen weiteren Rangminor bildet. Wir dürfen also nur Zeilen suchen, die zu unserer ersten gefundenden Teilmatrix passen, also an der Position, wo in der letzten, zurückgesetzten Zeile der von Null verschiedene Wert stand, ebenfalls nicht Null wird. Man beachte, daß das im Fall vollen Ranges nicht passieren kann.

Verbesserung

Wenn man nun noch ein zweites Mal all die Zeilen betrachtet, die nach der letzten, zurückgesetzten Zeile stehen, macht man die ganze Arbeit des Reduzierens doppelt: einmal, als man eine Zeile suchte, die einen höheren Rang ergeben hätte, und nun, da man Zeilen sucht, die den selben Rang ergeben. Da man sowieso nur solche Zeilen gebrauchen kann, die an der Position, wo in der letzten Zeile, die zum Rang beigetragen hat, der von Null verschiedene Wert stand, nicht zu Null werden, kann man sich den Wert an dieser Position auch merken.

Die verbesserte Version geht also so vor: Während des ganzen Algorithmus sucht man einerseits Zeilen, die höheren Rang ergeben würden, andererseits merkt man sich für jede Zeile, wo sich der höhere Rang nicht ergeben hat, den Wert der ebengenannten Position[8] in einer Liste für den Fall, daß keine Zeile mehr den Rang erhöhen wird.

Wenn nun aber wirklich noch eine Zeile gefunden wird, die zu einem höheren Rang beiträgt, kann man die gemerkten Werte getrost wegwerfen und neu mit dem Sammeln beginnen. Mithilfe der gemerkten Werte können wir jetzt die Determinanten der neu hinzukommenden Rangteilmatrizen mit der Eigenschaft, daß sie identisch bis auf die letzte Zeile sind, rekonstruieren.

2.4.5 Auswahl von Rangminoren

Nun haben wir modulo einer Primzahl ein paar Rangteilmatrizen mit ihren zugehörigen Determinanten gefunden, modulo einer anderen vielleicht ganz andere. Für unsere Determinantenberechnung brauchen wir aber die Determinanten von immer denselben Teilmatrizen.

Wenn wir nun die kombinierte Rang– und Determinantenberechnung für verschiedene Primzahlen durchgeführt haben, dann liefert jede Primzahl nur eine *Hypothese* für den Satz von Rangminoren, die wir am Ende wirklich berücksichtigen. Aus diesen Hypothesen können wir uns eine auswählen, die uns am besten gefällt.

Sicher ist es günstig, wenn möglichst viele Rangteilmatrizen in diesem Satz sind, denn dann wird der ggT der letztendlichen ganzzahligen Determinanten schön klein. Andererseits ist es günstig, wenn wir den gleichen Satz von Rangteilmatrizen für verschiedene Primzahlen haben; denn schon wenn uns bei einer Hypothese *eine* Rangmatrix fehlt, dann fehlt uns auch deren Determinante modulo dieser Primzahl, und wir müssen für diese Primzahl nochmal von vorn beginnen.

2.5 Zur Parallelisierbarkeit des modularen Verfahrens

Das modulare Verfahren ist hervorragend parallelisierbar. Für die modulare Rang– und Determinantenberechnung kann jeder Prozessor bezüglich einer anderen Primzahl rechnen.

Auch innerhalb einer modularen Determinantenberechnung kann parallelisiert werden: Auf die Abfolge von

- Kanonisieren einer neuen Matrixzeile
- Anwenden der Permutation
- Eliminierung mit der ersten Positionen
- Suchen des von Null verschiedenen Eintrages
- Dividieren der Zeile

läßt sich das Pipelining–Prinzip anwenden.

Auch für die modularen Elementarteiler ergibt sich für die verschiedenen Primzahlpotenzmoduln eine natürliche Teilung. Man kann auch schon, bevor der Rang und der Rangminoren–ggT feststeht, modulare Elementarteiler bezüglich häufiger Primzahlpotenzmoduln wie 2^{14} berechnen.

[8] und natürlich die Zeilennummer

Kapitel 3

Nichtmodulare Ansätze

Im folgenden werde ich mich mit Algorithmen beschäftigen, die nicht auf dem modularen Verfahren der letzten beiden Kapitel basieren, sondern auf dem ganzzahligen GAUSS–Verfahren des ersten Kapitels. Es sind viele Strategien des GAUSS–Verfahrens denkbar, in welcher Reihenfolge die elementaren Operationen angewendet werden, die ich alle als **nichtmodulare Verfahren** bezeichne.

3.1 Allgemeines

3.1.1 Warum nichtmodulare Verfahren?

Es gibt einige Beispiele, wo das modulare Verfahren nicht besonders geeignet ist:

- Bei einem nichtmodularen Verfahren tritt gar keine Koeffizientenexplosion ein. Dann braucht das modulare Verfahren länger. Das sind aber nur kleine Beispiele, für die in jedem Falle nur sehr kurze Zeit gerechnet wird.

- Die Elementarteiler sind riesig groß. Dann ist auch der Modul für die modulare Elementarteilerberechnung sehr groß[1], so daß er möglicherweise nicht faktorisiert werden kann; außerdem ist das Rechnen in sehr großen Restklassenringen langsam. Wenn nicht jede Matrix über dem Restklassenring in den Speicher des Computers paßt, kann man beim modularen Verfahren genauso von einer Koeffizientenexplosion reden wie beim nichtmodularen. Schon das Inverse eines kleinen Elements kann riesig werden.

  Da ich aber im modularen Verfahren so vorgehe, daß ich die Zeile eines gewählten Pivotelementes zunächst mit dem Inversen des Pivotwertes multipliziere, um es zu 1 zu machen, kann es vorkommen, daß dieses Inverse so groß ist, daß es die anderen Elemente der Pivotzeile enorm vergrößert. Damit tritt die Koeffizientenexplosion schlagartig ein, das modulare Verfahren ist in diesem Falle viel schlechter als das nichtmodulare.

Ich gehe aber davon aus, daß in den meisten praktischen Fällen das modulare Verfahren dasjenige ist, das überhaupt und am schnellsten die Elementarteiler berechnet. Das trifft umso mehr zu, je größer die Matrizen werden. Gerade bei Matrizen, die so groß sind, daß allein das Halten der Matrix im Hauptspeicher problematisch wird, kann man trotzdem noch davon ausgehen, daß bei genügend langer Rechenzeit immerhin noch ein Ergebnis ausgegeben wird. Dagegen werden alle nichtmodularen Algorithmen irgendwann an der Koeffizentenexplosion scheitern.

Ich betrachte nun eine etwas andere Aufgabenstellung. Es sollen nicht nur die Elementarteiler ausgegeben werden, sondern auch die beiden Transformationsmatrizen. Wenn also M unsere Ausgangsmatrix ist und ihre SMITH–Normalform PMQ, dann möchten wir P und Q berechnen. Das Paar (P, Q) werde ich im folgenden als **Diagonalisierung** bezeichnen.

Wenn man sich an die Entsprechung der elementaren Operationen und der elementaren Matrizen für das GAUSS–Verfahren erinnert, dann sieht man sofort, daß P und Q nebenbei entstehen — man multipliziert in jedem Schritt die entsprechenden Elementarmatrizen auf. Wir starten also mit $\{L := I_m|M|R :=$

[1] Wenn die Elementarteiler klein sind, und der Modul groß, so kann man ihn meistens dadurch verkleinern, daß man den ggT von mehr Rangminoren berechnet.

$I_n\}$, wobei I_m die Einheitsmatrix der Größe m sei. Bei einer Zeilenoperation mit der elementaren Matrix E wird aus

$$\{\dot{L}|\dot{M}|\dot{R}\} \longrightarrow \{E\dot{L}|E\dot{M}|\dot{R}\}$$

und bei einer Spaltenoperation aus

$$\{\dot{L}|\dot{M}|\dot{R}\} \longrightarrow \{\dot{L}|\dot{M}E|\dot{R}E\}$$

Am Ende steht mit PMQ als die SMITH–Normalform:

$$\{P|PMQ|Q\}$$

Der entscheidende Nachteil des modularen Verfahres besteht nun darin, daß wir die Diagonalisierung nicht so einfach bekommen — von den modularen Schritten (das Multiplizieren der in $\mathbb{Z}_d$ invertierbaren Matrizen) kann nicht auf die Schritte des ganzzahligen GAUSS–Verfahrens gefolgert werden.

Anders gesagt: wenn M unsere Ausgangsmatrix ist und wir jetzt die Diagonalisierung für M_d als $P_d M_d Q_d$ ausrechnen, so können wir nicht davon ausgehen, daß P_d und Q_d in $\mathbb{Z}$ betrachtet M ebenfalls diagonalisieren. Gibt es nun aber eine clevere Möglichkeit, doch modular rechnen zu können?

Koeffizenten der Diagonalisierung

Wir betrachten jetzt die SMITH–Normalform PMQ von M mit der Diagonalisierung (P, Q). Man macht sich leicht klar, daß P und Q keineswegs eindeutig bestimmt sind. Die angesprochenen Wahlfreiheiten im GAUSS–Verfahren führen nicht im Endeffekt auf dieselbe Diagonalisierung. Wenn man beispielsweise eine Nullzeile in der Matrix hat, kann man sie beliebig oft zu einer anderen addieren, wobei die Koeffizienten der Transformationsmatrix steigen. Die Einträge der Diagonalisierung (d. h. die Einträge von P und Q) können beliebig groß werden.

Daher macht es Sinn, von einer **minimalen Diagonalisierung** bezüglich einer Norm[2] zu reden. Es ist nun aber offensichtlich, daß eine Strategie des GAUSS–Verfahrens nicht die minimale Diagonalisierung herausbekommt.

Es läßt sich leicht nachprüfen, daß die Zwischenkoeffizenten der Transformationsmatrizen beim oben angegebenen Verfahren genauso explodieren wie die Zwischeneinträge der zu diagonalisierenden Matrix. Doch während die Einträge der Matrix am Ende wieder klein werden, wenn die Elementarteiler klein sind, werden die Einträge der Diagonalisierung immer größer.

Eine Frage ist nun, ob dieses Explodieren unvermeidbar ist, ob also die minimale Transformationsmatrix ebenfalls riesige Einträge hat.

Eine andere Frage ist, wie man auf die minimale Diagonalisierung kommt — oder wenigstens auf eine hinreichend kleine. Es sieht so aus, als wenn das Finden der minimalen Diagonalisierung ein exponentiell schweres Problem ist, es also kein wesentlich besseres Verfahren als das einfache Probierverfahren gibt:

Man beginnt mit der kleinstmöglichen Summe von 2 Normen einer unimodularen $m \times m$–Matrix und einer $n \times n$–Matrix $s := n + m$. Man nimmt sich alle Paare aus einer $m \times m$–Matrix P und einer $n \times n$–Matrix Q, wo die Summe der Norm von P und der Norm von Q gleich s ist. Für jedes dieser Paare testet man, ob PMQ in der SMITH–Normalform ist. Wenn keines der Paare M diagonalisiert, erhöht man s um 1.

Aufsplitten

Wenn man nur die Elementarteiler berechnet, kann man bezüglich verschiedener Moduln rechnen und die Ergebnisse später zusammenführen. Um die Diagonalisierung zu berechnen, ist das illusorisch.

Wenn wir modulo d_1 auf $P_{d_1} M_{d_1} Q_{d_1}$ als SMITH–Normalform gekommen sind, modulo d_2 auf $P_{d_2} M_{d_2} Q_{d_2}$, so gibt es im allgeminen keine P und ein Q, so daß $P \equiv_{d_1} P_{d_1}$, $Q \equiv_{d_1} Q_{d_1}$, $P \equiv_{d_2} P_{d_2}$ und $Q \equiv_{d_2} Q_{d_2}$ gilt; nicht einmal, wenn die gleiche Strategie des modularen GAUSS–Verfahrens benutzt wurde. Spätestens, wenn im Laufe des GAUSS–Algorithmus irgendein Matrixelement durch den Modul teilbar wurde, werden sich die beiden Diagonalisierungen voneinander unterscheiden.

Es stellt sich also heraus, daß wir nur einen Modul verwenden dürfen.

[2] Ich nehme alle Koeffizienten der linken und der rechten Matrix als einen langen Vektor und berechne von diesem die 1–Norm — eine andere Norm ist auch möglich und sinnvoll.

glücklicher Modul

Ich betrachte jetzt eine (deterministische) Strategie S für das modulare GAUSS–Verfahren . Bezüglich dieser nenne ich d einen **glücklichen Modul**, falls die modulare Diagonalisierung auch eine ganzzahlige ist, d. h. wenn die Strategie M_d als $P_d M_d Q_d$ diagonalisiert, so daß P_d und Q_d, in $\mathbb{Z}$ betrachtet, $P_d M Q_d$ ebenfalls in SMITH–Normalform ist. Zunächst ist klar, daß ein glücklicher Modul ein echtes Vielfaches des größten Elementarteilers sein muß. Das ist aber längst nicht hinreichend.

Es gibt eine offensichtliche Schranke in Abhängigkeit von der Strategie und der Matrix, hinter der alle Moduln glücklich sind: das Maximum aller auftretenden Zwischeneinträge. Die Frage ist, ob es auch eine handhabbar kleine Schranke gibt. Ein typischer Fall, wo der modulare Algorithmus andere Wege geht als der ganzzahlige, ist, wenn im Verlauf irgendein Wert zufällig durch den Modul teilbar wird und dann dessen Eliminierung wegfällt. Weil es eine Koeffizientenexplosion gibt, scheint es völlig aussichtslos, das sicher zu verhindern.

Man beachte den Unterschied zur unsicheren Rangberechnung im modularen Verfahren. Dort genügte es, wenn die Determinante nicht durch den Modul teilbar ist, hier darf kein einziger der explodierenden Matrixeinträge durch den Modul teilbar sein.

Nehmen wir an, daß beim Test, ob der Modul glücklich war, noch einige Werte außerhalb der Diagonalen nicht Null sind. Diese lassen sich leicht mit ein paar zusätzlichen GAUSS–Schritten eliminieren. Wenn es aber sehr viele sind, so haben wir nichts gewonnen gegenüber einem rein nichtmodularen Verfahren.

Beziehung zwischen modularer und ganzzahliger Lösung

Man habe jetzt $P_d M_d Q_d$ als d–modulare SMITH–Normalform von M_d berechnet. Man sucht jetzt eine Diagonalisierung (P, Q) von M, daß $P \equiv_d P_d$ und $Q \equiv_d Q_d$.

Aber wie soll das aussehen? Man hat eigentlich ein Gleichungssystem mit so vielen Unbekannten zu lösen, wie P und Q zusammen Positionen haben, und eine Vereinfachung ist nicht zu sehen.

Man betrachte folgendes Beispiel, das die prinzipielle Schwierigkeit verdeutlichen soll: Ein ganzzahliges Verfahren lieferte folgende ganzzahlige Diagonalisierungen:

$$M_1 := \begin{pmatrix} 2 & 7 \\ 0 & 2 \end{pmatrix} \Longrightarrow \left\{ \begin{pmatrix} 2 & -7 \\ -1 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \right\}$$

$$M_2 := \begin{pmatrix} 2 & -1 \\ 0 & 2 \end{pmatrix} \Longrightarrow \left\{ \begin{pmatrix} 2 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \right\}$$

Beide Ausgangsmatrizen sind kongruent modulo 8, die Transformationsmatrizen sind es nicht.

Nun gibt es zu den Transformationsmatrizen der ersten Matrix keine modulo 8 kongruente Matrizen, die die zweite Matrix diagonalisieren (mit Maple©[2] bestätigt). Das Beispiel zeigt nun, daß es Moduln gibt, deren modularer Lösung keine ganzzahlige Lösung entspricht.

Zumindest müßte folgendes geklärt werden:

> Sei die SMITH–Normalform von M_d von einer bestimmten Strategie als $P_d M_d Q_d$ berechnet. Gibt es P und Q, so daß $P \equiv_d P_d$ und $Q \equiv_d Q_d$ und PMQ SMITH–Normalform von M ist?

Triangularisieren

Ich sehe folgende Idee, eine Diagonalisierung modular zu berechnen, wenn man sich davon löst, die d–modulare SMITH–Normalform zu berechnen, und zunächst ausschließlich mit Zeilenoperationen operiert. Ich habe die Idee aber noch nicht weiter ausgeführt.

Algorithmus 5

1. Sei T_0 die zu diagonalisierende Matrix, i eine ganze Zahl auf 1 gesetzt.

2. Man berechnet modular eine obere Dreiecksmatrix T_i von T_{i-1} nur mit Zeilenoperationen.

3. Es läßt sich R_i finden, so daß $M = R_i T_i$ gilt. Man nimmt sich daher einfach jede Zeile von M her und stellt sie als Linearkombination der Zeilen von T_i dar. Wegen der Dreiecksform von T_i ist das leicht realisierbar.

4. Wenn T_i noch nicht in Diagonalform ist, transponiere T_i und gehe zu 2.

5. Die Transponierungen berücksichtigend ergeben sich nun invertierbare P und Q, so daß $T_i = PT_0Q$.

6. Man invertiere P und Q modular, z. B. mithilfe p–adischer Berechnungen.

7. Man hat nun die Diagonalmatrix hergestellt. Man stelle die Teilbarkeitsbedingung analog zu Algorithmus 6 her. Dabei berücksichtige man die Transformationmatrizen. (Siehe Algorithmus zum Kannan–Bachem–Verfahren)

3.1.2 Begriffe

Um die Formulierung der verschiedenen nichtmodularen Algorithmen zu vereinfachen, möchte ich die folgenden Begriffe benutzen:

Pivotposition Eine ausgezeichnete Position der Matrix, die am Anfang jedes Schritts des Gauss–Verfahrens gewählt wird.

Pivotwert Der Eintrag der Matrix an der Pivotposition. Der Pivotwert kann sich aber innerhalb eines Schritts des Gauss–Verfahrens ändern, muß aber immer ungleich Null sein.

Pivot Pivotposition zusammen mit dem derzeitig aktuellen Pivotwert.

Eliminierung der Pivotzeile (analog –spalte) Das Verfahren, das die Matrix mithilfe elementarer Operationen in eine Matrix überführt, in der in der Zeile des Pivots nur noch ein einziger Wert ungleich Null ist — der Wert an der Pivotposition.

Isolierung Wenn die Zeile und Spalte des Pivots eliminiert sind, bezeichne ich ihn als isoliert; das Verfahren, das dahin führt, als Isolierung — es besteht aus ggf. mehreren Eliminierungen von Pivotzeile– und spalte.

(wirklicher) Schaden ist Steigen der Matrixnorm bei irgendeiner Matrixoperation, z. B. einer Isolierung. Hier ist die 1–Norm gemeint, also die Summe der absoluten Beträge aller Einträge der Matrix.

primärer/sekundärer Vektor/Zuordnung Verschiedene Algorithmen beschäftigen sich zuerst mit der Pivotspalte und danach mit der Pivotzeile. Es gibt aber keinen Grund, nicht zuerst die Zeile zu nehmen; m. a. W. zeitweilig an einer transponierten Matrix zu arbeiten. Man kann sogar innerhalb eines Algorithmus die Entscheidung, ob zuerst Spalte oder zuerst Zeile, wechseln. Manchmal kommt das mit Formulierungen wie *vice–versa* zum Ausdruck, was aber zu Mißverständnissen führen kann.

Ich verwende darum neue Begriffe, um diesen Sachverhalt auszudrücken. Wenn zuerst die Zeile bearbeitet wird, dann ist der primäre Vektor die Zeile und der sekundäre Vektor die Spalte. Wenn zuerst die Spalte bearbeitet wird, ist der primäre Vektor die Spalte und der sekundäre die Zeile. Die Entscheidung, was zuerst bearbeitet wird, bezeichne ich als Zuordnung.

Vektor steht hier also für Zeile oder Spalte und nicht die übliche mathematische Bedeutung des Begriffs — genaugenommen ist der Vektor hier ein Element eines $\mathbb{Z}$–Moduls.

Die allgemeine Strategie des Gauss–Verfahrens ist nun folgendermaßen:

- Suchen eines Pivots
- Isolierung des Pivots durch Eliminierung des primären und des sekundären Vektors.
- Rekursion mit der Matrix, die durch Streichen von Pivotzeile und –spalte entsteht.

3.1.3 Wahlentscheidungen

Der GAUSS–Algorithmus läßt bezüglich der Suche des Pivots beträchtlich Freiheit, lediglich, daß der Wert an dieser Position ungleich Null ist, ist gefordert. Wenn einmal ein Pivot gewählt ist, so ist es wiederum beliebig, welche Werte der Pivotzeile und –spalte zur Reduktion ausgewählt werden.

Diese Entscheidungen bedingen, wie schnell Zwischeneinträge wachsen, und somit, ob, wann und wie stark eine Koeffizentenexplosion stattfindet. Je größer (Zeilen–Spaltenzahl, Größe der Einträge) die Matrix ist, umso mehr lohnt es sich, Rechenzeit für die Suche des Pivots zu verbrauchen. Zunächst wird nur auf Kosten hoher Rechenzeit mit kleineren Zahlen gerechnet. Wenn die Zahlen nicht mehr in ein Datenwort passen, hat das auch zur Folge, daß weniger Speicherplatz verbraucht wird.

Nach der Koeffizentenexplosion wird das Rechnen mit den langen Zahlen viel langsamer, so daß die kleineren Zahlen auch in der Rechenzeit zu Buche schlagen. Es gibt dann einen Break–Even–Point[3], wo der aufwendigere Algorithmus insgesamt schneller wird. Dann gibt es aber noch einen weiteren Punkt, wo nur noch der aufwendigere Algorithmus zu überhaupt einem Ergebnis kommt.

Pivotsuche

Die gängigen Heuristiken für die Pivotsuche sind:

- Man nehme eine Pivotwert von minimalem absoluten Betrag, damit er möglichst viele Werte der Matrix bereits teilt, oder die Teilbarkeit leicht zu realisieren ist.
- Man nehme einen Pivot in einer Zeile und Spalte mit möglichst kleiner Norm, dann wird das Koeffizientenwachstum nicht so stark ausfallen.

Wenn in den nun folgenden Algorithmen nichts weiter zur Pivotsuche gesagt wird, so kommen diese Heuristiken zum Einsatz.

3.2 Das Polynomiale Verfahren

3.2.1 ggT–Transformation

In der ursprünglichen Version des GAUSS–Verfahrens gibt es nur 3 elementare Operationen, und zwar:

- Multiplikation eines Vektors mit –1
- Vertauschen zweier Vektoren
- Ein ganzzahliges Vielfaches eines Vektors zu einem anderen addieren

Diese elementaren Operationen (oder Transformationen) werden um eine weitere ergänzt, die **ggT–Transformation**: Man wähle zwei Vektoren Z_1 und Z_2 und zwei beliebige von Null verschiedene ganze Zahlen h_1 und h_2. Man berechne die 4 Kofaktoren des ggTs von h_1 und h_2: $c_1, \ldots, c_4$, die durch

$$\begin{aligned} c_1h_1 + c_2h_2 &= g \\ c_3h_1 + c_4h_2 &= 0 \\ c_3 &= -h_2/g \\ c_4 &= h_1/g \end{aligned}$$

bestimmt sind, wobei $g = ggT(h_1, h_2)$. Die ggT–Transformation beruht jetzt auf dem Ersetzen von Z_1 durch $c_1Z_1 + c_2Z_2$ und Z_2 durch $c_3Z_1 + c_4Z_2$. Wie leicht zu zeigen ist, gilt für die Determinante

$$\begin{vmatrix} c_1 & c_2 \\ c_3 & c_4 \end{vmatrix} = 1$$

Daraus läßt sich leicht schlußfolgern, daß diese Operation eine Matrix in eine äquivalente Matrix überführt, es damit erlaubt ist, diese Operation als zusätzliche elementare zu nehmen. Der Zweck der Transformation ist einsichtig, wenn man h_1 und h_2 als Wert der Matrix an einer bestimmten Postition in Z_1 und Z_2 resp.

[3] im Sinne einer Matrixgröße

vorstellt, und sich ansieht, welche Zahlen nach der Transformation an diesen Positionen stehen, an denen vorher h_1 und h_2 gestanden habe, nämlich $ggT(h_1, h_2)$ und 0. Sie ist also ein praktisches Werkzeug, Matrixpositionen schnell zu Null zu machen.

$$\begin{pmatrix} h_1 & \cdots \\ \vdots & \\ h_2 & \cdots \\ \vdots & \end{pmatrix} \Longrightarrow \begin{pmatrix} ggT(h_1, h_2) & \cdots \\ \vdots & \\ 0 & \cdots \\ \vdots & \end{pmatrix}$$

Es gibt aber noch eine andere Sichtweise auf die ggT–Transformation. Man kann sie sich nämlich entstanden denken aus einer Folge der oben erwähnten anderen drei elementaren Transformationen.

Stehe an der Position von h_1 das vom Betrag her größere der beiden Zahlen, ggf. müssen die Vektoren vertauscht werden. Man dividiert h_1 durch h_2 mit Rest, $h_1 = h_2 q + r$, und zieht dann vom ersten Vektor das q–fache des zweiten Vektors ab, so daß danach im ersten Vektor r an der Position des h_1 steht. Dann vertauscht man die beiden Vektoren und dividiert erneut, bis die Division aufgeht, und dann eine 0 an der Position des h_2 steht. Die Operationen, die man nun mit h_1 und h_2 ausgeführt hat, entsprechen genau dem EUKLIDischen Algorithmus zur Berechnung des ggT zweier Zahlen, darum steht an der Position des h_1 nun der ggT von h_1 und h_2. Wenn man sich nun ansieht, wie die Kofaktoren im erweiterten Euklidischen Algorithmus bestimmt werden, ist klar, daß das genau die Folge von Transformationen war, die einer ggT–Transformation entsprechen.

Die ggT–Transformation hat eine gute und eine schlechte Seite:

Die gute ist, daß man nun eine geringere Anzahl von Transformationen für das GAUSS–Verfahren benötigt, womit es eher möglich ist, das Verhalten der Matrix während des Algorithmus abzuschätzen. Genau diese Eigenschaft ermöglichte es, zu beweisen, daß das Koeffizientenwachstum des auf die ggT–Transformation beruhenden Algorithmus polynomial beschränkt ist.

Die schlechte Seite ist, daß beide Vektoren bei der Transformation mit relativ großen Koeffizenten multipliziert werden und im Normalfall sich auch beide Vektoren ändern.

3.2.2 Subtraktions–Transformation

Der Symmetrie zur ggT–Transformation formuliere ich eine elementare Operation neu, ohne Wesentliches zu ändern; aus

- Ein ganzzahliges Vielfaches eines Vektors zu einem anderen addieren
  wird
- Ein ganzzahliges Vielfaches eines Vektors von einem anderen subtrahieren

3.2.3 GAUSS–Algorithmus nach KANNAN–BACHEM

Dieser Algorithmus unterscheidet sich von dem Algorithmus 1 durch den Einsatz der ggT–Transformation[4]. Man beachte, daß hier das Herstellen der Teilbarkeit und das Eliminieren der Matrixelemente quasi gleichzeitig erfolgt.

Eine einfache Verbesserung ist, zunächst die Matrix nur zu diagonalisieren, ohne auf die Teilbarkeitsbedingung der Elementarteiler einzugehen. Eine Diagonalmatrix läßt sich sehr leicht und ohne nennenswertes Koeffizientenwachstum herstellen. Wenn man aber schon während der Diagonalisierung auf die Teilbarkeitsbedingung eingeht, kann dies das Koeffizentenwachstum verschärfen.

Algorithmus 6 (nach [8])

1. Man wähle ein beliebiges von Null verschiedenes Element der Matrix als Pivot und vertausche Zeilen und Spalten, daß es an der Position links oben steht.

2. Eliminiere mithilfe der ggT– und Subtraktions–Transformation jeden einzelnen Wert des primären Vektors (hier immer die Zeile).

[4] In KANNAN–BACHEM ist nicht von einer zusätzlichen Transformation die Rede, es wird aber auf die Abfolge der anderen elementaren Operationen verwiesen, um diese Operation einzuführen

3. Eliminiere mithilfe der beiden Transformationen einen Wert des sekundären Vektors.

4. Wenn der primäre Vektor weiterhin Null ist (das ist immer bei der Subtraktions–Transformation der Fall), gehe zu 3, sonst zu 2.

5. Man betrachte jetzt die Submatrix, die durch Streichen der Pivotzeile und Pivotspalte entsteht. Wenn sie nicht aus lauter Nullen besteht, dann gehe damit zu 1.

6. Man betrachte wieder die gesamte Matrix — sie ist jetzt in Diagonalform, die Teilbarkeit der einzelen Diagonalelemente ist aber noch nicht gewährleistet. Man nehme also die Diagonalelemente in eine Liste, und forme sie so um, daß ein weiter vorn in der Liste stehender Wert einen weiter hinten in der Liste stehenden teilt. Man kann dazu

   - die Liste permutieren und
   - wenn in der Primzahlzerlegung eines Elements die Primzahlpotenz p^k auftritt, der Primzahlzerlegung eines anderen p^l, so kann man ersteres Element mit $p^{(l-k)}$ und letzteres mit $p^{(k-l)}$ multiplizieren; quasi die p–Exponenten dieser beiden Zahlen vertauschen. (Man macht sich leicht klar, daß ein Vertauschen in der Liste einer Zeilen/Spaltenvertauschung entspricht, und das Schieben eines Primteilers sich aus einer Subtraktions–Transformation und einer darauffolgenden ggT–Transformation ergibt.)

Dieser Algorithmus setzte einen theoretischen Meilenstein. Bei ihm wurde bewiesen, daß die Koeffizenten der Zwischenmatrizen polynomial (in der Größe der Koeffizienten der Ausgangsmatrix) beschränkt sind, z. B. in [7]. Zwar wurde bei noch keiner sinnvollen Strategie eine untere exponentielle Schranke gefunden, aber die Beweisbarkeit deutet zumindest darauf hin, daß es hier leichter ist, die Zwischeneinträge zu kontrollieren.

3.2.4 Kritik

Man betrachte einmal, was passiert, wenn der primäre Vektor ausgeräumt[5] wurde, der dann aktuelle Wert des Pivots aber noch nicht sämtliche Einträge der sekundären Vektors teilt. Irgendwann kommt man dann zu dem nicht teilenden Wert und macht eine ggT–Transformation. Was geschieht aber dann?

Da mit der Transformation auch der primäre Vektor wieder mit verändert wird, muß dieser erst wieder neu ausgeräumt werden. Man kann sich aber vorstellen, was dann mit dem Rest der Matrix passiert, wenn die ganzen Operationen des ersten Reduzierens der Pivotspalte damit fast umsonst waren.

Eine andere Schwäche wurde schon unter der ggT–Transformation angesprochen. Das Vermengen von Teilbarmachen und Eliminieren kann zur Folge haben, daß das Teilbarmachen dabei (für die Matrix) zu teuer wird.

3.3 Best–Remainder–Strategy

3.3.1 Allgemeines

Dieses Verfahren baut nicht auf dem polynomialen Verfahren auf sondern geht zurück auf den ursprünglichen Algorithmus. Da auf die ggT-Transformation verzichtet wird, treten damit auch ihre eben erwähnten Schwächen nicht auf. Die Leistungsfähigkeit wird durch das Ausschöpfen der Wahlfreiheiten erzielt.

Der Algorithmus wählt zunächst einen Pivot von minimalem absoluten Betrag, bei gleichen Beträgen wird der Pivot so gewählt, daß das Produkt aus dessen Zeilen– und Spaltennorm minimal ist[6]. Nun wird jeder von Null verschiedene Wert der Pivotzeile und –spalte der Reihe nach abgearbeitet. Ist der Wert w durch den Pivotwert p teilbar, so wird er zu Null gemacht; durch das Addieren des $-\frac{w}{p}$–fachen der Pivotzeile bzw. –spalte. Ist er nicht teilbar, so wird mithilfe einer Restdivision ermittelt, welches Vielfache der Pivotzeile bzw. –spalte addiert wird, so daß nach der Operation der Rest mit minimalem absoluten Betrag an der Position von w steht — also der **beste Rest = Best Remainder**.

[5] alle Werte bis auf den Pivotwert zu Null gemacht wurden

[6] gemeint ist hier die 1–Norm. George Havas experimentierte mit vielen verschiedenen Normen und Normenverknüpfungen, die angegebene Variante ist wirkungsvoll und relativ schnell zu berechnen.

3.3.2 Best–Remainder–Algorithmus

Algorithmus 7 (nach [5])

1. Man wähle das vom Absolutbetrag kleinste von Null verschiedene Element der Matrix als Pivot, bei gleichen Beträgen hat das kleinere Produkt von Zeilen– und Spaltennorm Vorrang. Man vertausche Zeilen und Spalten, so daß der Pivot an der Position links oben steht.

2. Man führe für alle von 0 verschiedenen Einträge der Pivotspalte und –zeile mit diesem Eintrag und dem Pivotwert eine Restdivision durch. Wenn der Quotient q ist, dann ziehe man das q–fache der Pivotzeile von der Zeile bzw. der Spalte des Elementes ab.

3. Wenn die Pivotspalte und –zeile nun (bis auf den Pivot) noch nicht vollständig ausgeräumt sind, dann gehe man zu 1.

4. Man betrachte jetzt die Submatrix, die durch Streichen der Pivotzeile und Pivotspalte entsteht. Wenn sie nicht aus lauter Nullen besteht, dann gehe damit zu 1.

5. Man erhält die Elementarteiler aus den Diagonalelementen analog zum Algorithmus 6.

Man beachte, daß im Schritt 2 nicht feststeht, ob man bei dem aktuellen Pivotwert bis zur vollständigen Isolierung kommt oder nicht. Wenn nicht, dann werden bei der Reduktion mit Sicherheit Elemente erzeugt, deren absolute Beträge kleiner als der des Pivotwertes sind. Da immer der kleinste Wert als Pivot genommen wird, kommt man irgendwann zu einem Wert, der alle Werte seiner Zeile und Spalte teilt. Daraus läßt sich leicht folgern, daß der Algorithmus nach endlich langer Zeit zu einem Ergebnis kommt.

Im Unterschied zum Algorithmus 1 wird dafür nicht irgendein Wert genommen, sondern es werden einfach alle reduziert. Da dabei die ganze Matrix noch durcheinandergeschüttelt wird, ist viel wahrscheinlicher, daß recht schnell ein schön kleines Element entsteht, womöglich gar 1 oder -1.

Der eigentliche Grund, daß nicht nach jeder einzelnen Reduktion nach einem neuen Pivot gesucht wird, ist jedoch folgender: Die Heuristik für die Suche des Pivots ist relativ teuer, denn es müssen alle Zeilen– und Spaltennormen berechnet werden; der Aufwand dafür in der angegebene Version des Best–Remainder–Algorithmus beträgt deutlich mehr als die Hälfte der gesamten Rechenzeit. Die Best–Remainder–Strategie ist diesbezüglich ein sehr leistungsfähiger Kompromiß.

Die Stärke des Algorithmus besteht darin, daß er nicht auf einem zufällig ungünstig geratenen Pivot beharrt, sondern beim nächsten Durchlauf an einer völlig anderen Stelle der Matrix einen besseren findet, und das einmalige Reduzieren[7] der Pivotzeile und –spalte hat noch nicht solch großen Schaden angerichtet. Das ist anders beim Algorithmus 6, wo der Pivot bis zur vollständigen Isolierung erduldet werden muß.

3.3.3 Kritik

Das Best–Remainder–Verfahren begrenzt die Koeffizientenexplosion gegenüber dem KANNAN–BACHEM–Verfahren erheblich. Solange die Koeffizienten noch klein bleiben, ist er genauso schnell, wird beim Beginn der Koeffizientenexplosion zwar erheblich langsamer — das wird aber schon nach kurzem durch die kleineren Zahlen wettgemacht.

Trotz aller dieser Vorzüge kann man doch auch sehen, in welchen Situationen er Chancen zur Kleinhaltung der Matrixeinträge vergibt. Man betrachte folgendes Beispiel:

$$\begin{pmatrix} 1 & \cdots \\ \vdots & \\ 48 & \cdots \\ \vdots & \\ 96 & \cdots \\ \vdots & \end{pmatrix}$$

Nehmen wir an, als Pivots kommen diese 3 Werte in Frage. Best Remainder nimmt natürlch die 1 als Pivot. Nehmen wir nun weiter an, daß in der Zeile der 1 relativ große Zahlen stehen. Mit dem großen

[7] ich meine damit entweder Eliminierung oder Größenreduktion

Multiplikator 96 werden diese noch größer, in diesem Fall wird es dann sicher besser sein, die 48 als Pivot zu wählen, die 96 mit kleinem Multiplikator zu beseitigen und dann mit der 1 eine ggT–Transformation zu machen. Diese bedeutet hier nichts anderes, als daß das 48–fache der Zeile mit der 1 abgezogen wird und implizit die Zeile mit der 1 zur Pivotzeile wird (an der Position der Zeile mit der 48).

Unabhängig davon ist das Best–Remainder–Verfahren im Rechenzeit–Koeffizientenwachstum–Kompromiß das beste bisher implementierte Verfahren, darum habe ich es implementiert, obwohl ich den Schwerpunkt meiner Arbeit auf ein neues Verfahren gesetzt habe.

3.4 Der Preview–Algorithmus

Ich habe eben in der Kritik zum Best–Remainder–Verfahren auf vergebene Chancen zur Kleinhaltung der Matrixeinträge hingewiesen — hier setze ich für ein neues Verfahren an.

Meine Idee ist, bei der Pivotsuche den gigantischen Aufwand zu treiben, alle Operationen, die ich mit einem Pivot machen würde, vorauszuberechnen, und danach einzuschätzen, welchen Pivot ich auswähle.

In [10] wurde gezeigt, daß die Suche nach einer Strategie, die ein minimales Koeffizientenwachstum der Zwischenmatrizen für eine bestimmte Matrix erzeugt, ein NPV–Problem ist. Das bedeutet m. E., daß die einzige Möglichkeit, die minimale Strategie zu finden, das Verfolgen jeder einzelnen Wahlmöglichkeit bis zur Diagonalisierung und der anschließende Vergleich der Zwischeneinträge wäre.

Das ist lächerlich, aber es ist genau diese Strategie, die ich zu approximieren versuche. Ich möchte das in einigen Schritten tun:

Zunächst will ich mich darauf beschränken, den Schaden jeder einzelnen Isolierung zu minimieren. Es ist natürlich denkbar, daß die optimale Strategie zwischendurch einen stärkeren Schaden zuläßt, was sich dann erst später auszahlt — worauf ich verzichte.

Dann will ich mich darauf beschränken, nur die Operationen, die im Algorithmus 6 angewendet werden, zu benutzen, das sind also das Wählen eines Pivotelements und die Folge von ggT– und Subtraktions–Transformationen, die jeweils einen Eintrag sofort eliminieren. Das ist einer der Knackpunkte des Verfahrens. Man könnte auf die Idee kommen, den Best–Remainder–Algorithmus als Grundlage zu nehmen, und dessen Schritte vorherzusehen und zu minimieren. Das ist zweifellos ein Ansatzpunkt, aber im weiteren wird klar, wie stark ich von der Einfachheit der Schritte im KANNAN–BACHEM–Verfahren Gebrauch mache.

Weiterhin will ich nicht vorausberechnen, wie sich die Einträge der Matrix wirklich verhalten, sondern ich bin mit einem Schätzwert zufrieden, den ich schnell berechnen kann.

3.4.1 einfach isolierbare Pivots

einfach isolierbarer Pivot Ein Pivot (als Matrixposition) heißt einfach isolierbar, wenn bei irgendeiner Zuordnung des primären Vektors und irgendeiner Reihenfolge der Eliminierung der Elemente des primären Vektors (im Schritt 4. des Algorithmus 6) der dann entstehende Wert an der Pivotposition alle einzelnen Werte des sekundären Vektors teilt. Das ist gleichbedeutend damit, daß nach der Eliminierung des sekundären Vektors der primäre Vektor nicht noch einmal eliminiert werden muß; mithin, daß der primäre Vektor nur ein einziges Mal eliminiert wird.

Wie schon angedeutet, ist die nichteinfache Isolierung der ultimative Garant für eine Koeffizentenexplosion. **Die einfach–Isolierbarkeit hat erste Priorität bei der Suche des Pivots.**

Abstieg Der Abstieg bestimmt die Reihenfolge, in der die einzelnen Werte des primären Vektors eliminiert werden. Dabei wird jeweils entweder eine Subtraktions–Transformation oder eine ggT–Transformation benutzt, je nachdem ob der derzeit aktuelle Pivotwert den Wert im primären Vektor teilt oder nicht.

Für jeden einfach isolierbaren Pivot mit einer Zuordnung kann es unterschiedliche Abstiege geben. Ein Abstieg ist durch Indices des primären Vektors definiert.

Schritte des Abstiegs Das sind die einzelnen Indices des Abstiegs.

Stufen des Abstiegs Das sind alle die Schritte des Abstiegs, die eine ggT–Transformation erfordern. An einer Stufe wird also der Pivotwert kleiner, während bei den Schritten, die nur eine Subtraktions–Transformation erfordern, der Pivotwert unverändert bleibt.

Schaden eines Abstieges Dies ist ein Schätzmaß, wie stark die Koeffizenten der Restmatrix durch einen bestimmten Abstieg steigen werden. **Die Schadensfunktion hat die zweite Priorität bei der Auswahl des Pivot.**

Vektorteiler ggT aller Einträge des Vektors.

Typen einfach isolierbarer Pivots

Um zu prüfen, ob ein Pivot einfach isolierbar ist, müssen wir die entsprechenden Abstiege explizit ausführen, um dann zu testen, ob der Pivotwert, der nach der Eliminierung des primären Vektors an der Pivotposition steht, alle Einträge des sekundären Vektors teilt, die sich auch im Laufe der Eliminierung des primären Vektors geändert haben können. Da aber dafür der Rechenaufwand viel zu hoch ist, möchte ich schon im vornherein einschätzen können, ob ein Pivot einfach isolierbar ist.

Dazu berechne ich alle Zeilen– und Spaltenteiler zu jeder Pivotsuche neu.

Satz 10 (Typ 1) *Wenn ein Matrixeintrag alle Einträge seiner Zeile und seiner Spalte teilt, dann ist er ein einfach isolierbarer Pivot. Das ist genau dann der Fall, wenn dieser Matrixeintrag $\pm$ gleich seinem Zeilenteiler und Spaltenteiler ist.*

Der Beweis liegt einfach darin, daß der primäre Vektor ausschließlich mit Subtraktions–Transformationen eliminiert wird, sich also der sekundäre Vektor noch nicht geändert hat, wenn der primäre Vektor eliminiert worden ist. Da auch der Pivotwert sich nicht geändert hat, teilt er nach wie vor alle Einträge des sekundären Vektors.

Satz 11 (Typ 2) *Sei S der ggT aller Einträge des primären Vektors. Sei der ggT von r Einträgen $z_1, \ldots, z_r$ des primären Vektors ebenfalls schon S. Seien die den Einträgen $z_1, \ldots, z_r$ des primären Vektors zugeordneten (parallel zum sekundären Vektor) Vektoren $Z_1, \ldots, Z_r$. Wenn S alle Einträge des sekundären Vektors und der Vektoren $Z_1, \ldots, Z_r$ teilt, dann ist der entsprechende Pivot einfach isolierbar.*

Zum Beweis betrachte ich den Abstieg, der $z_1, \ldots, z_r$ als Stufen hat. Es wird nur mit den Vektoren $Z_1, \ldots, Z_r$ eine ggT-Transformation ausgeführt, nur dabei kann sich der sekundäre Vektor ändern.

Bei einer ggT-Transformation mit Z_i wird der sekundäre Vektor durch eine Linearkombination aus sekundärem Vektor und dem entsprechenden Z_i ersetzt. Da aber der ursprüngliche sekundäre Vektor durch S teilbar war, und alle Z_i durch S teilbar sind, ist auch der sekundäre Vektor am Ende noch durch S teilbar.

Satz 12 (Typ 3) *Wenn der primäre Vektor Vektorteiler 1 hat, dann ist der entsprechende Pivot einfach isolierbar.*

Typ 3 ist ein einfach zu berechnender Spezialfall vom Typ 2, genau wie Typ 1.

Es stellt sich heraus, daß es in vielen Matrizen eine Fülle von einfach isolierbaren Pivots dieser 3 Typen gibt.

3.4.2 Schadensmaß der Abstiege

Um den Schaden eines Abstiegs auszurechnen, könnte man die entsprechende Isolierung ausführen und dann die Matrixnormen vergleichen. Ich ziehe aber eine Approximation vor, die in der Berechnungskomplexität linear statt quadratisch ist[8]. Man rechnet statt mit der ganzen Matrix mit dem Vektor der Normen. Man führt dieselben Transformationen, die man auf der Matrix ausführen würde, auf dem Normenvektor aus, mit den selben Multiplikatoren.

Konkreter. Ich betrachte hier ausschließlich einfach isolierbare Pivots. Das heißt, daß für das Eliminieren des sekundären Vektors kein Anstieg irgend eines Koeffizenten der Matrix möglich ist, ich betrachte also nur die einzelnen Schritte des Abstiegs. Und hier addiere ich den Schaden für jeden einzelnen Schritt.

Was passiert bei einer Subtraktions–Transformation ? Der entsprechende Vektor wird mit dem $-q$–fachen des sekundären Vektors (den ich im folgenden als Pivotvektor bezeichnen werde) multipliziert. Was ich dazu brauche, ist einerseits ein Schätzwert für die Multiplikation einer Zeile mit einer Zahl, andererseits für die Addition zweier Vektoren.

[8] Abgesehen von der Zahlengröße. Ich gehe hier von der Größe der Matrix als Bezugsgröße und einer Multiplikation als Bezugseinheit aus.

Die Multiplikation ist einfach abzuschätzen, die Norm wird einfach mit der selben Zahl multipliziert; das ist dann nicht geschätzt, sondern ganz genau.

Die Addition ist komplizierter, die Norm der Summe zweier Vektoren liegt zwischen 0 und der Summe der Normen beider Vektoren. Ich werde aber gleich eine Funktion α definieren, die aus den Normen zweier Vektoren den Schätzwert für die Norm der Summe dieser beiden Vektoren liefert.

Sei P der Pivotvektor mit dem Pivotwert p, Z der Vektor, in dem der Eintrag z zu eliminieren sei. Als Schaden einer Subtraktions–Transformation nehme ich also

$$\alpha(\|Z\|, z/p\|P\|) - \|Z\|.$$

Für die ggT–Transformation ist es etwas komplizierter. Wenn $c1, c2, c3, c4$ die Kofaktoren des ggTs vom Pivotwert p und dem entsprechenden Matrixwert w, nehme ich als Schaden

$$\alpha(c4\|Z\|, c3\|P\|) - \|Z\|,$$

außerdem ersetze ich meinen Wert für die Pivotvektornorm

$$\|P\| \Longleftarrow \alpha(c2\|Z\|, c1\|P\|) - \|Z\|,$$

die ich ebenfalls nicht neu berechne, sondern nach selbem Schema schätze. Der Pivotwert für die nun folgenden Stufen des Abstiegs ist der ggT vom altem Pivotwert p und w.

3.4.3 Schätzung der Norm einer Summe zweier Vektoren

Wie schon gesagt, kann ich die Norm des Summenvektors nicht genau vorhersehen, wenn nur die Normen der Summanden gegeben sind. Aber ich kann sie relativ gut abschätzen.

Ich betrachte den Erwartungswert zweier zufälliger Vektoren unter dem statistischen Modell, daß die Einträge gleichverteilt zwischen $-n$ und n sind[9]. Um diesen Erwartungwert ohne eine statistische Rechnung zu approximieren, nehme ich *Prototypen* für zufällige Vektoren. Dazu zwei Überlegungen:

Wenn die Norm des einen Vektors viel größer ist als die des anderen, dann wird die Norm des Summenvektors ziemlich genau der Norm des größeren entsprechen. Der kleinere Vektor produziert hier nur ein Schwanken um die Norm des größeren, also ist die Norm des größeren als Abschätzung für die Norm der Summe gut genug.

Der andere Extremfall ist, wenn zwei gleich große Vektoren addiert werden. Hierfür rechne ich gleich einen Prototyp explizit aus. Das Ergebnis wird ein Faktor f sein, mit der die Norm der beiden Vektoren multipliziert werden muß, um auf den Schätzwert der Summe zu gelangen.

Die Zwischenstufen behandle ich so, daß ich die kleinere Norm mit diesem Faktor f multipliziere und dann das Maximum aus der größeren Norm und diesem Produkt als Schätzung für die Norm der Summe nehme.

Ich betrachte jetzt folgenden Prototyp von gleichverteiltem Vektor:

$$v_1 := \left.\begin{array}{llllllll} (-n, & -n+1, & \dots & 0, & \dots & n-1, & n, \\ -n, & -n+1, & \dots & 0, & \dots & n-1, & n, \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ -n, & -n+1, & \dots & 0, & \dots & n-1, & n) \end{array}\right\} 2n+1\text{–mal}$$

Ich betrachte die 1–Norm, also die Summe der absoluten Beträge aller dieser Einträge. Der Vektor hat die Norm:

$$\begin{array}{rcl} 2(2n+1)\sum_{i=1}^{n} i & = & 2(2n+1)(-1/2n - 1/2 + 1/2(n+1)^2) \\ & = & (2n+1)n(n+1) \end{array}$$

Ich addiere v_1 zu einem Vektor v_2, der die gleichen Einträge wie v_2 hat, die nur in einer anderen Reihenfolge stehen. Und zwar so, daß jeder Wert $-n, \dots, n$ des ersten Vektors genau einmal zu jedem Wert $-n, \dots, n$ des zweiten addiert wird. Man macht sich klar, daß diese Zuordnung die wahrscheinlichste unter allen möglichen Zuordnungen ist. Dieser Vektor hat die selbe Norm wie der erste.

[9] Diese Annahme ist durchaus vernünftig. Die diskretisierte GAUSS–Verteilung mag zwar für viele Anwendungsfälle durchaus genauer sein, aber hierfür scheint es nicht möglich zu sein, ohne weitere Annahmen zu einem *universellen* Faktor zu kommen.

Die Einträge des Summenvektors schwanken dabei natürlich von $-2n, \ldots, 2n$ und sehen folgendermaßen aus:

$$\begin{array}{llllllll} v_+ := (& -2n, & -2n+1, & \ldots & -n, & \ldots & 1, & 0, \\ & -2n+1, & -2n+2, & \ldots & -n+1, & \ldots & 0, & -1, \\ & \vdots & \vdots & & \vdots & & \vdots & \vdots \\ & 0, & 1, & \ldots & n, & \ldots & 2n-1, & 2n \quad) \end{array}$$

Ich summiere also:

$$\begin{array}{rll} 1 & \text{mal} & |-2n| \\ 2 & \text{mal} & |-2n+1| \\ \vdots & & \vdots \\ 2n & \text{mal} & |-1| \\ 2n+1 & \text{mal} & |0| \\ 2n & \text{mal} & |1| \\ \vdots & & \vdots \\ 2 & \text{mal} & |2n-1| \\ 1 & \text{mal} & |2n| \end{array}$$

Das ergibt für die Norm der Summe:

$$\begin{array}{rl} & 2\sum_{i=1}^{2n} i(2n+1-i) \\ = & -2(2n+1)n + 2n(2n+1)^2 - 8/3n - 4/3 + 2(2n+1)^2 - 2/3(2n+1)^3 \\ = & 4/3(2n+1)n(n+1) \end{array}$$

Wenn ich nun die Norm der Summe durch Norm eines Vektors dividiere, erhalte ich 4/3, das ist der Faktor, den ich suche.

Für meine α_1–Funktion ergibt sich:

$$\alpha_1(x,y) = \begin{cases} x & \text{falls } x > 4/3y \\ 4/3y & \text{falls } 4/3y \geq x \geq y \\ 4/3x & \text{falls } 4/3x \geq y > x \\ y & \text{falls } y > 4/3x \end{cases}$$

Der Vollständigkeit halber ergänze ich hier die Ergebnisse der analogen Rechnungen für andere Normen:
2–Norm[10]: $\sqrt{2}$
0–Norm[11]: 1
∞–Norm[12]: 2

3.4.4 Sinnlose Abstiege

Es stellt sich heraus, daß unter den $(n-1)!$ Möglichkeiten für die Reihenfolge der Transformationen zur Eliminierung des primären Vektors die meisten wegen drei Schadensabschätzungen herausfallen.

- Ich betrachte zwei Vektoren, die beide für eine ggT–Transformation in Frage kommen. Es macht keinen Sinn, den Vektor mit der höheren Norm zu nehmen, denn die ggT-Transformation mit ihm würde (im Hinblick auf meine Schadensabschätzung) die Norm der Pivotzeile mehr vergrößern als die ggT-Transformation mit dem Vektor kleinerer Norm. Vorausgesetzt, daß noch einige Transformationen folgen, wird der Schaden der Eliminierung der anderen Elemente des primären Vektors vergrößert.

- Ich betrachte jetzt den Abstieg $p, p_1, \ldots, p_k, g$, wobei p der anfängliche Wert an der Pivotposition, g der Teiler des primären Vektors und $p_1, \ldots, p_k$ die Werte an der Pivotposition sind, die nach den einzelnen ggT–Transformationen entstehen. Eine bestimmter Wert des primären Vektors, der nicht

[10] d–Norm von $(v_1, \ldots, v_n)$ ist definiert als $\sqrt[d]{\sum_{i=1}^n v_i^d}$
[11] Anzahl der von 0 verschiedenen Einträge
[12] maximaler Eintrag

für die Eliminierung durch eine ggT–Transformation vorgesehen ist, kann nur dann durch eine Subtraktions–Transformation eliminiert werden, wenn der Pivotwert auf dieser Stufe des Abstiegs diesen Wert teilt, er könnte dann aber auch auf allen folgenden stufen des Abstieges eliminiert werden.

Man kann nun aber mit Sicht auf meine Schadensabschätzung sagen, daß die entsprechende Subtraktions–Transformation genau dann ausgeführt werden muß, wenn der Pivotwert so groß wie möglich ist. Erstens ist dann der Multiplikator so klein wie möglich; zweitens sind dann so wenig ggT-Transformationen wie möglich ausgeführt worden, die meinen Schätzwert der Norm des Pivotvektors vergrößert haben, denn wegen meiner Schadensdefinition kann die Norm des Pivotvektors sich nicht verkleinern.

So kommt also für die Eliminierung eines solchen Wertes nur die höchstmögliche Stufe in Betracht.

- Die Reihenfolge auf einer Abstiegs–Stufe spielt keine Rolle, weil sich der Pivotvektor nicht durch eine Subtraktions–Transformation ändern kann.

Also z. B.:

$$\begin{array}{l} \begin{pmatrix} 4 & \cdots \\ \vdots & \\ 6 & \cdots \\ \vdots & \\ 4 & \cdots \\ \vdots & \end{pmatrix} \end{array} \begin{array}{l} \text{Pivot, Norm: 21} \\ \vdots \\ \text{ggT, Norm: 18} \\ \vdots \\ \text{Sub, Norm: 22} \\ \vdots \end{array}$$

Wenn die zweite 4 gleich zu Beginn eliminiert wird, dann ist der Schaden (da Multiplikator 1) $6 (= 3/4 \cdot 21 - 22)$. Wenn zuerst die ggT–Transformation ausgeführt wird, wird die Norm der Pivotzeile nun auf 24 geschätzt. Mit Multiplikator 2 wird die zweite 4 nun mit Schaden $26 (= 2 \cdot 24 - 22)$ reduziert.

Man hat nun für jede *absteigende Folge von Teilern des Pivotwertes* genau einen Abstieg, der wirklich in Frage kommt.

Unter den Abstiegen, die wir nun noch nicht ausgeschlossen haben, müssen wir nun den besten auswählen. Können wir noch weitere Abschätzungen machen, um von möglichst wenig Abstiegen den Schaden explizit berecchnen zu müssen?

schneller/langsamer Abstieg sind Synonyme für Abstiege mit wenigen/vielen Stufen. Es läßt sich hier nicht von vornherein sagen, ob es besser ist, wenige ggT–Transformationen zu haben (um die Norm des Pivotvektors klein zu halten) oder viele (weil sich dann viele der Subtraktions–Transformations–Vektoren mit einem kleineren Multiplikator eliminieren lassen).

3.4.5 Algorithmus zur Suche der Abstiege eines Pivots

Der entscheidende Punkt für die Realisierbarkeit des Preview–Verfahrens liegt in der effektiven Berechenbarkeit der Abstiege.

Sei ein Pivot mit einer Zuordnung gewählt. Wir möchten alle Abstiege ermitteln, die nun möglich sind. Dabei dürfen wir Abstiege ausschließen, die wegen der im letzten Unterpunkt angeführten Schadensabschätzungen nicht in Fragen kommen. Der Abstieg ist durch die Indices des primären Vektors bestimmt. Wir möchten aber als Ergebnis dieser Prozedur zusätzlich für jeden einzelnen Index wissen, ob die zu seiner Eliminierung nötige Transformation eine Subtraktions–Transformation oder eine ggT–Transformation ist.

Ich formuliere den Algorithmus als Graphalgorithmus. Die Knoten des betrachteten gerichteten Graphen sind Zahlen (als vorhergesehene Pivotwerte), die Kanten sind die Indices der Positionen des primären Vektors, mit denen die ggT–Transformation ausgeführt wird.

Zunächst wird der Graph initialisiert, der Pivotwert ist der einzige Knoten, es gibt noch keine Kanten.

Nun werden alle Indices, in der Reihenfolge der Größe der Normen[13], beginnend mit dem kleinsten, des primären Vektors daraufhin geprüft, ob mit dem dazugehörigen Vektor eine ggT–Transformation ausgeführt werden würde — das ist der Fall, wenn der ggT aus Pivotwert und dem Wert an der Position des primären Vektors vom absoluten Betrag her kleiner ist als der Pivotwert. Dieser ggT wäre der neue Pivotwert.

[13]Die Normen werden am Beginn jeder Pivotsuche sortiert. Das mache ich mir noch an einer anderen Stelle zunutze.

Würde nun nun mit dem vorliegenden Index i eine ggT–Transformation ausgeführt werden, und sei der ggT aus Pivotwert p und dem Wert des primären Vektors an der i–ten Position g, dann erzeugen wir einen neuen Knoten g und eine Kante vom Knoten p zum Knoten g mit der Aufschrift i. Wenn nun aber der Knoten g und die Kante von p nach g schon existiert, dann erzeugen wir keine neue Kante (denn dann wäre die Norm des Vektors, mit dessen Position diese bereits existierende Kante beschriftet wäre, kleiner; und der Vektor i würde einen höheren Schaden ergeben).

Wir wiederholen das Durchgehen durch alle Indices, bis keine Änderungen mehr am Graphen vorgenommen werden. Das ist recht schnell der Fall, denn es werden immer nur neue Knoten erzeugt, deren Wert kleiner als der des Pivotwertes ist.

Nun gilt aber folgendes:

> Wenn im Graphen ein Knoten vorhanden ist, dessen Wert den Teiler des sekundären Vektors teilt, so ist der Pivot zusammen mit der Zuordnung einfach isolierbar.
>
> Die (gerichteten) Wege im Graphen vom Pivotwert bis zum Teiler des primären Vektors sind genau die Abstiege, die nach unseren Schadensabschätzungen in Frage kommen.

In meiner Implementierung verifiziere ich schon in der aufrufenden Umgebung, daß die übergebene Pivotposition mit der Zuordnung ein Kandidat für Typ 2 oder Typ 3 einfach isolierbarer Pivots ist. Es gibt einen Parameter `poslist` der speziell auf den Typ 2 zugeschnitten ist. In ihm sind die Indices von Vektoren enthalten, deren Teiler Vielfache des Teilers des primären Vektors sind.

Am Ende des Algorithmus werden für jeden berechneten Abstieg noch die Vektoren den Stufen des Abstieges zugeordnet, die nur für Subtraktions–Transformationen vorgesehen sind.

3.4.6 Beschränkte Pivotsuche

Nun stellt sich aber heraus, daß im Normalfall sehr viele unterschiedliche Abstiege existieren, so daß die Berechnung sehr viel Zeit in Anspruch nimmt. Deshalb werden wir mit einer Approximation des minimalen Abstiegs leben müssen. Es gibt etliche sinnvolle Ansätze dazu, die darauf basieren, Abstiege, die höchstwahrscheinlich einen hohen Schaden haben, von vornherein zu meiden.

Wie ausführlich in [5] argumentiert, macht es für den Best–Remainder–Algorithmus viel Sinn, Pivots mit kleiner Zeilen– und Spaltennorm für den Pivot zu nehmen. Das gilt für den Preview–Algorithmus ganz genauso. Es hat sich bei einigen kleinen[14] Beispielen herausgestellt, daß der Abstieg mit dem minimalen Schaden bei einem Pivot gefunden wurde, der eine der kleinsten Zeilen– und Spaltennormen hatte.

Ich werde aber nicht das Produkt von Zeilen– und Spaltennorm zur Vorauswahl der in Frage kommenden Pivotpositionen heranziehen, sondern ein effektiveres Verfahren anwenden, wobei ich mir zunutze mache, daß die Normen der Größe nach sortiert vorliegen.

Ich durchmustere die Matrixpositionen mit einer dem CANTORschen Dreiecksverfahren ähnlichen Methode. Zuerst nehme ich die Position, deren Zeile und Spalte die jeweils kleinste Norm haben. Danach die Position der zweitkleinsten Spaltennorm und der kleinsten Zeilennorm. Danach die Position mit der kleinsten Spaltennorm und der zweitkleinsten Zeilennorm. Danach die Position der drittkleinsten Spaltennorm und der kleinsten Zeilennorm. Und immer so weiter, daß ich eine Position eher berücksichtige, wenn die Summe der Platzziffern unter den Zeilen– und Spaltennormen für diese Position kleiner ist als die entsprechende Summe einer anderen Position.

Wenn ich nun die Anzahl der von Null verschiedenen Pivots begrenze, kann ich trotzdem sicher sein, daß ich die Pivots mit den kleinsten Zeilen– und Spaltennormen berücksichtigt habe. Daneben bietet es sich an, die Zahl der berechneten Abstiege zu begrenzen, da die Rechenzeit im wesentlichen von der Anzahl der berechneten Abstiege abhngig ist.

Es hat sich auch herausgestellt, daß besonders kurze Abstiege Kandidaten für kleine Schäden sind. Die langen Abstiege sind zudem schwieriger zu berechnen. Ich begrenze also die Anzahl der Stufen der Abstiege. Eine sehr wirkungsvolle Methode, nicht zu viele Schäden berechnen zu müssen, ist, sich eine bestimmte Zielvorgabe für einen kleinen Schaden zu geben. Wird dieser bei einem Abstieg unterschritten, wird kein weiterer Abstieg berechnet.

Ich habe in meinem Programm folgende Parameter zur Begrenzung der Pivotsuche:

tolerate Wenn der Schaden für eine bestimmten Abstieg kleiner als dieser Wert ist, wird er sofort genommen. (`pivot tolerated`)

[14] bei großen dauert das Berechnen wirklich aller Abstiege ewig

pivotbound Wenn so viele von 0 verschiedene Pivots analysiert worden sind, breche die Suche ab und nehme den bis dahin besten Abstieg.

descentbound Wenn so viele Abstiege berechnet worden sind, breche man die Suche ab und nehme den bis dahin besten Abstieg.

intolerate Sei jetzt bereits ein Abstieg ausgewählt worden. Wenn der Schaden dieses Abstiegs diesen Wert überschreitet, nimmt man diesen Abstieg nicht zur Isolierung, sondern denkt sich etwas anderes aus.

stepbound Grenze für die Länge der Abstiege.

3.4.7 gute Parameterwerte erlernen

In meiner Implementierung sind die Parameter `stepbound`, `tolerate`, `intolerate`, `pivotbound` und `descentbound` nicht ein für alle Male festgelegt, sondern können sich im Laufe des Programms ändern, und zwar in Abhängigkeit davon, was im letzten Schritt geschehen ist. Das hat folgenden Hintergrund:

Es ist günstig, wenn die unterschiedlichen Arten, die Pivotsuche abzubrechen, abwechselnd vorkommen. Wird der Abbruch immer durch den Parameter `tolerate` vorgenommen, so ist anzunehmen, daß er für die entsprechende Matrix zu hoch angesetzt wurde. Andererseits kann man auch denken, daß man jetzt weniger Pivotpositionen berücksichtigen muß, da man ja eh Abstiege mit kleinem Schaden findet. Also setzt man den Parameter `tolerate` herab, daneben verkleinert man `pivotbound` und `descentbound`.

Genauso umgekehrt, wenn kein Abstieg einen tolerierbaren Schaden aufweist, so ist man vielleicht beim Schaden zu streng, andererseits hat man vielleicht nicht genug Pivots berücksichtigt.

Der Parameter `stepbound` verändert das Verhältnis von Pivotpositionen zu Abstiegen; wenn alle langen Abstiege verboten werden, dann gibt es eben weniger Abstiege zu einer Pivotposition. Um auch dieses Verhältnis der Matrix anzupassen, wird beim `pivotbound`–Abbruch der Parameter herauf–, beim `descentbound`–Abbruch herabgestzt.

`intolerate` wird heraufgesetzt, wenn wirklich einmal ein nicht zu akzeptierender Schaden ausgerechnet wurde, ansonsten immer heruntergesetzt.

Das Herauf– und Heruntersetzen folgt einem einheitlichen Schema, zu jedem dieser Parameter gibt es einen Satz Parameter der Form:

up Soviel wird der Parameter erhöht.

down Soviel erniedrigt.

bot Er wird aber nie kleiner als dieser Wert.

top Nie größer als dieser Wert.

3.4.8 Auflösung der Kannan–Bachem–Starre

Ich hatte schon vorher angesprochen, daß mitunter eine vollständige Isolierung ganz enormen Schaden verursachen kann, während einige Subtraktions–Transformationen zwischendurch diesen erheblich mildern können. Es ist einer der Gründe für die erhebliche Überlegenheit des Best–Remainder–Algorithmus gegenüber dem Kannan–Bachem–Verfahren.

Ich möchte diese Starre beheben, ohne die grundsätzliche Strenge aufzugeben, die mir ermöglicht, den Schaden so genau vorherzusagen. Die Starre kommt daher, daß ein Wert des primären Vektors nur mit dem Pivotvektor eliminiert werden kann, nicht aber mit einem anderen.

Als Veranschaulichung ein Beispiel:

$$\begin{pmatrix} 1 & \cdots \\ \vdots & \\ 24 & \cdots \\ \vdots & \\ 24 & \cdots \\ \vdots & \end{pmatrix} \begin{matrix} \text{Pivot} \\ \vdots \\ \text{Sub} \\ \vdots \\ \text{Sub} \\ \vdots \end{matrix}$$

Nehmen wir an, die 1 wurde als Pivot gewählt. Beim strengen Algorithmus würden beide Zeilen mit der 1 eliminiert werden, jedesmal mit hohem Multiplikator. Da macht es mehr Sinn, erst die zweite Zeile mit der dritten zu eliminieren und danach erst den Pivotvektor heranzuziehen. Es läßt sich leicht ein Kriterium angeben, wann das Sinn macht: man rechnet den entsprechenden Schaden aus.

Ich unterscheide verschiedene Typen. Je höher die Typnummer, desto mehr Werte kommen in Betracht, also umso aufwendiger die Berechnung, und desto unwahrscheinlicher, daß ein günstiger Wert gefunden wird.

Typ 1: Eliminierung von gleichen Werten

Sei der Abstieg festgelegt, und auf einer bestimmten Stufe des Abstiegs stehen nun zwei gleiche Werte als mit einer Subtraktions–Transformation zu eliminierende Werte — der Pivotwert auf dieser Stufe (nach der zu dieser Stufe gehörenden ggT–Transformation) sei p.

$$\begin{array}{c} P \\ \vdots \\ Z_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p & \cdots \\ \vdots & \\ d & \cdots \\ \vdots & \\ d & \cdots \\ \vdots & \end{pmatrix}$$

Hier lohnt sich eine Eliminierung von Z_1 durch Z_2, wenn

$$d/p \cdot \|P\| > \|Z_2\|.$$

Typ 2: Eliminierung von einander teilenden Werten einer Stufe

Sei hier die Situation wie bei Typ 1, beide Werte sollen durch eine Subtraktions–Transformation mit der Pivotzeile P eliminiert werden.

$$\begin{array}{c} P \\ \vdots \\ Z_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p & \cdots \\ \vdots & \\ d_1 & \cdots \\ \vdots & \\ d_2 & \cdots \\ \vdots & \end{pmatrix}$$

Hier ist eine Eliminierung von Z_1 durch Z_2 möglich, wenn

$$d_2 \mid d_1,$$

und lohnt sich, wenn

$$d_1/p \cdot \|P\| > d_1/d_2 \cdot \|Z_2\|.$$

Typ 3: Eliminierung von einander teilenden Werten verschiedener Stufen

Der Unterschied zum Typ 2 ist, daß die Elimnierung von Z_2 auf einer anderen, niedrigeren Stufe vorgesehen ist. Sei die Situation so, daß Z_1 auf der Stufe steht, wo der Wert p an der Pivotposition steht.

$$\begin{array}{c} P \\ \vdots \\ Z_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p & \cdots \\ \vdots & \\ d_1 & \cdots \\ \vdots & \\ d_2 & \cdots \\ \vdots & \end{pmatrix}$$

Hier ist eine Eliminierung von Z_1 durch Z_2 möglich, wenn

$$d_2 \mid d_1,$$

und lohnt sich, wenn

$$d_1/p \cdot \|P\| > d_1/d_2 \cdot \|Z_2\|.$$

Der wesentliche Unterschied zum Typ 2 besteht darin, daß die Suche nach günstigen Werten nicht auf die Werte einer Stufe beschränkt ist, sondern alle Werte niedrigerer Stufen berücksichtigt werden müssen. Wegen der höheren Multiplikatoren ist es hier aber unwahrscheinlicher, einen geeigneten Wert zu finden; die entsprechende Operation ist demnach uneffektiver.

Typ 4: Größenreduktion auf einer Stufe

Es geht hier nicht um die sofortige Eliminierung eines Wertes des primären Vektors, sondern darum, die spätere Eliminierung zu vereinfachen.

Seien Z_1 und Z_2 zwei Vektoren einer Abstiegsstufe, auf der die Pivotposition den Wert p hat. Es gelte für eine gewisse Zahl q, daß $e_1 = d_1 + qd_2 < d_1$. Ich nehme nun weiter an, daß die Zeile Z_1 nach der Reduktion ebenfalls auf dieser Stufe gestanden hätte.

$$\begin{array}{c} P \\ \vdots \\ Z_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p & \cdots \\ \vdots & \\ d_1 & \cdots \\ \vdots & \\ d_2 & \cdots \\ \vdots & \end{pmatrix}$$

Es lohnt sich, Z_1 mit dem q–fachen der Zeile Z_2 zu reduzieren, wenn

$$q \cdot \|Z_2\| + e_1/p \cdot \|P\| < d_1/p \cdot \|P\|.$$

Typ 5: Größenreduktion auf verschiedenen Stufen

Sei Z_1 ein Vektor einer Abstiegsstufe, auf der die Pivotposition den Wert p_1 hat. Es gelte für eine gewisse Zahl q, daß $e_1 = d_1 + qd_2 < d_1$. Ich gehe jetzt davon aus, daß die Zeile Z_1 nach der Reduktion, $\hat{Z}_1$, auf einer Abstiegsstufe stehen würde, wo an der Pivotposition ein p_2 stehen würde. Im Unterschied zu den Eliminierungen kann das hier auch eine höhere Stufe sein.

$$\begin{array}{c} P_1 \\ \vdots \\ Z_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p_1 & \cdots \\ \vdots & \\ d_1 & \cdots \\ \vdots & \\ d_2 & \cdots \\ \vdots & \end{pmatrix} \cdots \begin{array}{c} P_2 \\ \vdots \\ \hat{Z}_1 \\ \vdots \\ Z_2 \\ \vdots \end{array} \begin{pmatrix} p_2 & \cdots \\ \vdots & \\ e_1 & \cdots \\ \vdots & \\ d_2 & \cdots \\ \vdots & \end{pmatrix}$$

Es lohnt sich, Z_1 mit dem q–fachen der Zeile Z_2 zu reduzieren, wenn

$$q \cdot \|Z_2\| + e_1/p_2 \cdot \|P_2\| < d_1/p_1 \cdot \|P_1\|.$$

In meinem Programm hielt ich nur die Implementierung der Typen 1 bis 3 für sinnvoll, um die Rechenzeit nicht unnötig zu verlängern.

3.5 Previewed Best Remainder?

Ich möchte mich hier noch einmal mit der Möglichkeit auseinandersetzen, das bisher beste Verfahren, den Best–Remainder–Algorithmus, mit den Ideen des Preview–Algorithmus zu verbessern.

Wir wollen also einen Best–Remainder–Schritt vorhersehen. Das Problem ist nun dabei, daß er mehrere Aufgaben gleichzeitig erfüllen soll. Wenn die Matrix das erlaubt, dann soll ein Pivot isoliert werden. Wenn nicht, dann sollen Einträge erzeugt werden, die kleiner sind als der kleinste bisherige, am besten gleich eine 1.

Der Pivot ist mit einem Schritt isolierbar, wenn er vom Typ 1 einfach isolierbar ist (Siehe Preview–Verfahren.) Die Vorhersage darüber ist genauso wie dort am ehesten über alle Zeilen– und Spaltenteiler zu treffen.

Wenn das nicht geht, könnte man den besten Rest suchen, ohne alle Einträge des primären und sekundären Vektors in eine Reduktionsoperation einzubeziehen. Anders als im originalen Best–Remainder–Verfahren wird dabei nicht die ganze Matrix durcheinandergeschüttelt; die Wahrscheinlichkeit, daß kleine Einträge zufällig entstehen, ist entsprechend kleiner. Dafür wird diese Operation viel weniger Schaden verursachen.

Die entscheidende Schwierigkeit ist zu entscheiden, ob man lieber einen einfach isolierbaren Pivot mit hohem Schaden isoliert oder lieber auf einen anderen, besseren einfach isolierbaren Pivot wartet, und statt der Isolierng lieber eine Operation mit geringerem Schaden auswählt, die keinen Pivot isoliert. Woher aber die Gewißheit nehmen, daß danach das Isolieren besser geht? Und wenn ja, ob sich der Aufwand gelohnt hat?

Der Knackpunkt sind die unterschiedlichen Herangehensweisen von Kannan–Bachem und Best–Remainder, ersteres geht gradlinig auf das Ziel der Isolierung zu, während letzteres mit vielen unübersichtlichen Transformationen auf zufällig entstehende günstige Pivots hofft und meistens Glück hat. Aber dieses *Glück haben* ist schwer vorauszusehen.

3.6 Evolutions–Strategien

3.6.1 Pivot–Populationen

Dies ist eine Herangehensweise, die auf folgenden Postulaten beruht:

> Es ist Zufall, wie stark die Koeffizienten bei einer bestimmten Vorgehensweise im Gauss–Verfahren anwachsen werden. Es ist nicht vorherzusehen (auch unter Zurhilfenahme eines Preview–Verfahrens), welche Isolierungen die optimale (im Sinne des Koeffizientenwachstums) Strategie ausführt. Die günstigste Möglichkeit einer Isolierung (also die, die die optimale Strategie ausführt) ist aber immer unter den Isolierungen zu finden, die einen relativ geringen Schaden angerichtet haben.

Das führt zu folgender grundsätzlichen Idee: Ich betrachte gleichzeitig ablaufende Gauss–Verfahren mit anderen Pivotwerten bei den einzelnen Isolierungen als Population. Ich wähle bei einem Schritt nicht nur einen Pivot, sondern immer mehrere. Das kann eine feste Zahl von Pivots sein, aber auch eine variable. Für diese führe ich unabhängig voneinander die Isolierung durch. Das ist der Weg, wie sich die Population vermehrt.

Wenn bei manchen der Mitglieder der Population die Matrixnorm zu groß wird, dann sollen sie sterben. Das kann konkret so aussehen, daß nur eine bestimmte Anzahl der Population bei jedem Schritt überlebt, oder aber daß nur innerhalb eines gewissen Spielraums über der Matrixnorm des besten Vertreters ein Überleben gesichert ist.

Ich habe eine Evolutionsstretegie für die Pivotsuche beschrieben, sie ist genausogut für jede andere Wahlentscheidung des Gauss–Verfahrens anwendbar. Ich kann die Evolutionsstrategie mit den anderen hier beschriebenen Verfahren in unzähligen Kombinationen koppeln, das Ergebnis wird immer so aussehen, daß bei einer Vervielfachung der Rechenzeit ein etwas geringeres Wachstum der Koeffizenten zu erwarten ist.

3.6.2 Backtracking

Eine ähnliche Idee liegt dem Backtracking zugrunde. Wenn sich eine ausgeführte Folge von Operationen als katastrophal für die Matrixeinträge herausgestellt hat, dann gehe man zu einem gewissen Punkt des Ablaufs zurück, und starte dann von dort mit einer etwas anderen Stratregie von neuem.

Dazu ist es selbstverständlich nötig, in regelmäßigen Abständen Kopien der Zwischenmatrix und aller anderen relevanten Daten abzulegen, die dann zurückgeladen werden.

Dieses Verfahren habe ich implementiert.

3.7 Reduktionen

Man läßt das Ziel, die Diagonalform der Matrix herzustellen, für eine Augenblick außer acht, und kümmert sich nur darum, die Einträge der Matrix etwas zu verkleinern.

3.7.1 Gauss–Reduktion

Dieser Algorithmus berechnet von 2 Vektoren eine Linearkombination, die minimale euklidische Norm besitzt. Er ist in [8] ausführlich auf Seite 23 beschrieben, so verzichte ich hier darauf. Ich betrachte ihn hier vor allem als Möglichkeit, die Matrixnorm zu verkleinern.

Das geschieht folgendermaßen: Ich wähle 2 Vektoren (Zeilen oder Spalten) aus, für die ich die minimale Linearkombination berechne. Die Smith–Normalform ändert sich nicht, wenn ich jetzt diese den größeren der beiden ersetzen lasse.

Ich bezeichne jetzt als einen **Reduktions–Durchlauf**, daß sukzessive jeder Vektor mit jedem reduziert wird. Es ist klar, daß damit noch nicht alle Möglichkeiten der Reduktion erschöpft sind, da ich immer nur jeweils 2 Vektoren betrachte. Wenn ich aber jetzt soviele Reduktions–Durchläufe mache, bis sich nichts mehr ändert (das ist möglich, da sich bei jedem Schritt die 2–Norm der gesamten Matrix verkleinert), dann ist die Matrix bezüglich der 2–Norm minimal klein.

Das wäre für unsere Berechnung zwar ein idealer Zustand, leider sind dafür so viele Durchläufe nötig, daß das Verfahren nicht praktikabel wäre. Es stellt sich aber heraus, daß die ersten Durchläufe am stärksten die Matrixnorm reduzieren. In meiner Implementation benutze ich daher immer nur einige Durchläufe. Das ist auch der entscheidende Vorteil dieses Verfahrens, es ist *dosierbar*.

Ein Nachteil ist, daß es die 2–Norm minimiert, ich aber an allen anderen Stellen des Programms (besonders bei der Berechnung des Schadens) die 1–Norm berücksichtige. Allerdings gibt es dafür nicht solch effektive Verfahren — andererseits ist der Unterschied der Normen nicht so gravierend.

Ich habe diese Routine implementiert.

3.7.2 LLL–Reduktion

Dieser Algorithmus wird z. B. in [8] beschrieben, ist auch schon in GAP von Thomas Breuer implementiert (siehe [1]). Er ist ein wesentlich effektiveres Verfahren als die im letzten Abschnitt dargestellte Gauss–Reduktion für Matrizen, um eine minimierte äquivalente Matrix zu berechnen (wenn auch nicht in einem solch scharfen Sinne, das wäre für mein Problem aber auch gar nicht nötig).

Leider hat er sich als nicht praktikabel für mich erwiesen, und zwar aus folgenden Gründen:

- LLL hat natürlich einen entscheidenden Vorteil: Die Zwischeneinträge bleiben beschränkt (da in jedem Schritt eine gewisse Determinante kleiner wird, siehe [8]). Allerdings hat LLL mit einer anderen Koeffizentenexplosion zu kämpfen die der Gram–Schmidt–Koeffizenten (Details in [8]). Dort wird vorgeschlagen, die Gram–Schmidt–Koeffizenten mit Gleikommazahlen zu approximieren — leider gibt es *die* gegenwärtig in GAP noch nicht.

- Der LLL–Algorithmus löst ein Problem, daß in seiner Schwierigkeit durchaus mit der Bestimmung der Smith–Normalform vergleichbar ist; die Anwendung als Reduktionsroutine ist problematisch, weil nicht von vornherein gesagt werden kann, ob in einer verhältnismäßigen Zeit ein Ergebnis ausgegeben wird.

  Wie steht es mit der Möglichkeit, nur ein bißchen LLL anzuwenden? Es stellt sich heraus, daß dies nicht in einer solch sinnvollen Weise möglich ist, wie bei der Gauss–Reduktion.

3.8 Mischen

Wir lassen auch hier das Ziel Diagonalisierung außer acht und machen einige Transformationen, die keinen großen Schaden erwarten lassen. Das soll kein Witz sein, sondern kann Sinn machen, wenn man eingesteht, daß es Zufall ist, ob eine günstige Isolierung existiert oder nicht. Dann mischt man einmal gut durch, und hofft auf Glück beim nächsten Spiel.

Für das Mischen bietet sich vor allem eine dosierte Reduktion an.

3.9 Zur Parallelisierbarkeit nichtmodularer Verfahren

Die Parallelisierung nichtmodularer Algorithmen ist nicht offensichtlich. Die Isolierung der einzelnen Diagonalelemente *muß* nacheinander erfolgen. Doch verschiedene Teilalgorithmen lassen sich hervorragend parallelisieren. Das sind im besonderen:

Das Berechnen der Normen Alle etwas aufwendigeren Verfahren benutzen zur Pivotsuche und als Heuristik für andere Wahlentscheidungen die Normen von Vektoren der Matrix. Das läßt sich mit einer Prozessorenzahl bis zur Summe aus Zeilen– und Spaltenzahl sehr einfach parallelisieren, aber auch darüber hinaus über das Berechnen von Teilnormen.

Abstiegssuche und Schadensberechnung im Preview Einfach isolierbare Pivots können auf jeder Matrixposition unabhängig voneinander gesucht werden, gegenseitig beeinflussend ist lediglich das Vergleichen der Schadenszahlen.

Pivotpopulationen Populations–Verfahren bieten sich hervorragend zu paralleler Berechnung an, ja sie werden erst bei paralleler Berechnung überhaupt sinnvoll.

Subtraktions–Transformationen Hier wird der Aufwand zur Parallelisierung schon höher. Wenn einmal festgestellt wurde, das das Pivotelement nun alle Werte seiner Zeile teilt, können die einzelnen Subtraktions–Transformationen aufgeteilt werden. Für die ggT–Transformation ist das nicht möglich, da sich hierbei der Pivotvektor ändert.

Reduktionen Das hier vorgestellte Reduktionsverfahren mit jeweils 2 Vektoren ist parallelisierbar, allerdings sind noch einige Gedanken für eine sinnvolle Strategie nötig. Im LLL–Algorithmus sind nur einige Teilalgorithmen parallelisierbar.

Kapitel 4

Bedienungsanleitung

Im folgenden werden die Prozeduren beschrieben, mit denen man die Elementarteiler auf unterschiedliche Arten berechnen kann.

4.1 ElementaryDivisorsIMat, grundlegende Aufrufe

Aufruf: ElementaryDivisorsIMat(<mat|imat> [,optionrec]);

Parameter: mat: eine ganzzahlige GAP–Matrix.
imat: ein IMR (ein spezieller Record) der aus einer ganzzahligen GAP–Matrix erzeugt werden kann.
optionrec: ein Record, dessen Felder den Algorithmus steuern. Wenn nicht angegeben, wird mit Default–Werten gerechnet.

Ergebnis: Die Elementarteiler der Matrix als Liste.

Ich werde jetzt zunächst einige Beispiele für die Möglichkeiten, die der Options–Record bietet, geben; weiter unten werde ich dann auf jedes einzelne Record–Feld eingehen.

4.1.1 Default — sicheres Verfahren

Aufruf: ElementaryDivisorsIMat(mat);

Parameter: mat: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Elementarteiler werden mit dem sicheren modularen Verfahren berechnet.

Beispiel:

```
gap> ElementaryDivisorsIMat[[2,1],[0,2]]);
[ 1, 4 ]
```

4.1.2 Schnelles Verfahren

Aufruf: ElementaryDivisorsIMat(mat, rec(fastdet:=true, fastrank:=true));

Parameter: mat: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Elementarteiler werden mit dem schnellen modularen Verfahren berechnet. Diese Methode kann ein falsches Ergebnis ausgeben, die Wahrscheinlichkeit dafür ist aber sehr gering und bisher nur bei speziell dafür konstruierten Beispielen aufgetreten.

4.1.3 Fast sicheres Verfahren

Aufruf: `ElementaryDivisorsIMat(mat, rec(fastrank:=true) );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Dieser Aufruf ist ein Kompromiß zwischen den letzten beiden. Die Fehlerwahrscheinlichkeit ist um viele Größenordnungen geringer geworden als beim schnellen Verfahren, man braucht weniger Zeit als beim sicheren Verfahren.

4.1.4 nichtmodulares Verfahren, um Diagonalisierung zu erhalten

Aufruf: `ElementaryDivisorsIMat(mat ,rec(algorithm="bestrem", isom:=true) );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Ein GAP–Record mit dem Feld `divisors`, in dem die Elementarteiler der Matrix stehen, und den Feldern `left` und `right`, die die Transformationsmatrizen enthalten. Die Elementarteiler werden mit dem Best–Remainder–Verfahren berechnet. Nichtmodulare Verfahren benötigen ab einer bestimmten Matrixgröße erheblich mehr Zeit als modulare Verfahren und kommen bei noch größeren Matrizen überhaupt nicht mehr zum Ergebnis, weil sie an der Koeffizientenexplosion scheitern. Die Diagonalisierung läßt sich aber nur mit nichtmodularen Verfahren ermitteln.

Beispiel:

```
gap> m:=[[2,1],[0,2]];
[ [ 2, 1 ], [ 0, 2 ] ]
gap> ElementaryDivisorsIMat(m ,rec(algorithm="bestrem", isom:=true) );
rec(
  divisors := [ 1, 4 ],
  left := [ [ 1, 0 ], [ 2, -1 ] ],
  right := [ [ 0, 1 ], [ 1, -2 ] ] )
gap> last.left*m*last.right;
[ [ 1, 0 ], [ 0, 4 ] ]
```

4.1.5 nichtmodulares Verfahren ohne Diagonalisierung

Aufruf: `ElementaryDivisorsIMat(mat ,rec(algorithm="bestrem") );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Elementarteiler der Matrix mit dem Best–Remainder–Algorithmus. Dieser Aufruf kann einerseits benutzt werden, um zu testen, ob man die Transformationsmatrizen ermitteln *könnte*, denn deren Berechnung verlangsamt das Programm erheblich. Andererseits gibt es seltene Fälle, wo das nichtmodulare Verfahren dem modularen überlegen ist.

4.1.6 Preview–Algorithmus

Das ist ein anderer nichtmodularer Algorithmus, auch mit ihm können die Transformationsmatrizen berechnet werden. Er bremst die Koeffizientenexplosion stärker als der Best–Remainder–Algorithmus, braucht dafür aber erheblich mehr Zeit. Auch mit ihm kann man sich die Diagonalisierung ausgeben lassen.

Aufruf: `ElementaryDivisorsIMat(mat ,rec(algorithm="preview") );`

4.2 RankIMat

Aufruf: RankIMat(<mat|imat> [,optionrec]);

Parameter: mat: eine ganzzahlige GAP–Matrix.
imat: ein IMR
optionrec: ein Record, dessen Felder den Algorithmus steuern. Wenn nicht angegeben, wird mit Default–Werten gerechnet.

Ergebnis: Der Rang der Matrix mit dem modularen Verfahren berechnet.

Hier gibt es nicht viele Möglichkeiten für den Options–Record, ich gebe jetzt also alle Möglichkeiten dafür an.

4.2.1 Default — sicheres Verfahren

Aufruf: RankIMat(mat);

Parameter: mat: eine ganzzahlige GAP–Matrix.

Ergebnis: Der Rang der Matrix wird mit dem sicherenen modularen Verfahren berechnet.

Beispiel:

```
gap> RankIMat([[1,2],[2,0]]);
2
```

4.2.2 Schnelles Verfahren

Aufruf: RankIMat(mat, rec(fastrank:=true));

Parameter: mat: eine ganzzahlige GAP–Matrix.

Ergebnis: Der Rang der Matrix wird mit dem schnellen modularen Verfahren berechnet. Diese Methode kann ein falsches Ergebnis ausgeben, die Wahrscheinlichkeit dafür ist aber sehr gering und bisher nur bei speziell dafür konstruierten Beispielen aufgetreten.

4.2.3 Schnelle Berechnung der Hadamard–Schranke

Aufruf: RankIMat(mat, rec(fasthadamard:=true));

Parameter: mat: eine ganzzahlige GAP–Matrix.

Ergebnis: Der Rang der Matrix wird mit dem sicherenen modularen Verfahren berechnet. Die Hadamard–Schranke wird mit einem einfacheren, schnelleren Verfahren ausgerechnet; es werden aber möglicherweise mehr modulare Rangberechnungen benötigt. Darum ist nicht von vornherein zu sagen, welches Verfahren besser ist.

4.3 DeterminantIMat

Aufruf: DeterminantIMat(<mat|imat> [,optionrec]);

Parameter: mat: eine ganzzahlige GAP–Matrix.
imat: ein IMR
optionrec: ein Record, dessen Felder den Algorithmus steuern. Wenn nicht angegeben, wird mit Default–Werten gerechnet.

Ergebnis: Die Determinante der Matrix mit dem modularen Verfahren berechnet.

Beispiel:

```
gap> DeterminantIMat([[1,2],[2,0]]);
-4
```

Hier gibt es nicht viele Möglichkeiten für den Options–Record, ich gebe jetzt also alle Möglichkeiten dafür an.

4.3.1 Default — sicheres Verfahren

Aufruf: `DeterminantIMat(mat);`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Determinante der Matrix wird mit dem sicherenen modularen Verfahren berechnet.

4.3.2 Schnelles Verfahren

Aufruf: `DeterminantIMat(mat, rec(fastrank:=true) );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Determinante der Matrix wird mit dem schnellen modularen Verfahren berechnet. Diese Methode kann ein falsches Ergebnis ausgeben, die Wahrscheinlichkeit dafür ist aber sehr gering und bisher nur bei speziell dafür konstruierten Beispielen aufgetreten.

4.3.3 Schnelle Berechnung der Hadamard–Schranke

Aufruf: `DeterminantIMat(mat, rec(fasthadamard:=true) );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix.

Ergebnis: Die Determinante der Matrix wird mit dem sicherenen modularen Verfahren berechnet. Die Hadamard–Schranke wird mit einem einfacheren, schnelleren Verfahren ausgerechnet; es werden aber möglicherweise mehr modulare Determinanten benötigt. Darum ist nicht von vornherein zu sagen, welches Verfahren besser ist.

4.4 InfoIMat

Aufruf: `InfoIMat(true);` oder `InfoIMat(false);`

Parameter: ein boolescher Wert

Ergebnis: Bei `true` werden bei den Aufrufen `ElementaryDivisorsIMat`, `RankIMat` und `DeterminantIMat` verschiedene Ablaufinformationen und Zwischenergebnisse ausgegeben, bei `false` nichts.

Beispiel:

```
gap> ElementaryDivisorsIMat([[2,1],[0,2]]);
[ 1, 4 ]
gap> InfoIMat(true);
gap> ElementaryDivisorsIMat([[2,1],[0,2]]);
#I    Rank guessing......fullrank
#I  Rank:2
#I  Considering 1 submatrices.
#I    Hadamarding ...done.
#I  New Hadamard bound:10^0
#I  Considering 0 primes.
#I    Determinants calculating......done
#I  Rankminor gcd multiple:4
#I    Factors:2^2
```

```
#I    Taking factor...2^3 ...done
#I  Elementary Divisors:[ 1, 4 ]
[ 1, 4 ]
```

Die Ausgabe wird in Übereinstimmung mit vielen anderen GAP–Prozeduren mithilfe von Funktionen InfoIMat1, InfoIMat2 und InfoIMat3 (für die nichtmodularen auch InfoIMat4) übernommen, die innerhalb des Programms aufgerufen werden. Sie werden entweder auf Print oder auf Ignore gesetzt.

So kann die Ausgabe auch selektiert werden:

```
gap> InfoIMat2:=Print;;InfoIMat3:=Ignore;;
gap> ElementaryDivisorsIMat([[2,1],[0,2]]);
#I    Rank guessing......fullrank
#I    Hadamarding ...done.
#I    Determinants calculating......done
#I    Factors:2^2
#I    Taking factor...2^3 ...done
[ 1, 4 ]
```

Die Prozedur InfoIMat steuert nur die beiden Funktionen InfoIMat2 und InfoIMat3.

4.5 IMR–Datenstruktur

4.5.1 IMRIMat

Der erste Parameter von ElementaryDivisorsIMat konnte alternativ eine Matrix oder ein spezieller Record sein. Diese Prozedur erzeugt einen solchen Record. Man benutzt ihn vor allem, wenn nicht zu erwarten ist, daß das Ergebnis schnell ausgegeben wird.

Man kann dann bereits berechnete Zwischenergebnisse speichern und wieder laden, und vor allem während der Berechnung in den Ablauf des Algorithmus eingreifen. Genaueres zu den IMR–Records ist im Kapitel **Implementation in** GAP zu finden.

Wird die Funktion ElementaryDivisorsIMat mit einem IMR aufgerufen, dann werden ggf. bereits errechnete Zwischenergebnisse benutzt.

Aufruf: `IMRIMat(mat[,optionrec] );`

Parameter: `mat`: eine ganzzahlige GAP–Matrix. `optionrec`: ein Record, dessen Felder den Algorithmus steuern. Wenn nicht angegeben, wird ein IMR mit Default–Werten erzeugt.

Ergebnis: Der IMR–Record der Matrix.

Beispiel:

```
gap> IMRIMat([[2,1],[0,2]]);
IntegerMatrix(
  matrix  := ... ,
  transposed  := false,
  rows  := 2,
  cols  := 2,
  minrank  := 0,
  maxrank  := 2,
  ElementaryDivisorsOptions  := rec(
  algorithm := "modular",
  maxfivers := 20,
  safeexp := false,
  tolerate := 1/50,
  tolbot := 0,
  toltop := 1/3,
  tolup := 1/25,
```

```
toldown := 1/100,
intolerate := 1/2,
intolbot := 2/5,
intoltop := 4,
intolup := 1/10,
intoldown := 1/50,
avoidrate := 5/2,
pivotbound := 30,
pivbot := 24,
pivtop := 46,
pivup := 3,
pivdown := 1,
descentbound := 90,
desbot := 74,
destop := 100,
desup := 6,
desdown := 2,
stepbound := 4,
stepbot := 2,
steptop := 6,
stepup := 1,
stepdown := 1,
shuffles := [ 1, 2 ],
redurate := 1,
isom := false ),
RankOptions  := rec(
fastrank := false,
fasthadamard := false ),
DeterminantOptions  := rec(
fastdet := false,
fasthadamard := false )  )
```

Die Beschreibung aller Felder ist weiter hinten in diesem Kapitel zu finden.

4.5.2 SaveIMRTo

Aufruf: `SaveIMRTo("file","name",imat);`

Parameter: `"file"`: ein Name für ein File, an die der IMR angehängt wird.
`"name"`: ein Name, der beim nächsten Einlesen als Variablenname des IMRs fungiert.
`imat`: ein IMR.

Ergebnis: Der IMR wird mit allen evtl. schon eingetragenen Zwischenwerten an das File angehängt. Dieses File kann dann später direkt wieder in GAP eingelesen werden. Existiert das File noch nicht, wird es angelegt.

Beispiel:

```
gap> SaveIMRTo("heineken","h120",h120);
```

Bemerkungen: Beim späteren Wiedereinlesen muß vorher das IMat–Programm geladen sein. Der Grund dafür ist die Prozedur `PrintIMR`, die im File `imat.g` definiert ist und die jedem IMR in das `operations`–Feld eingehängt wird.

Auch wenn kein IMR explizit erzeugt wurde, kann gespeichert werden, denn in diesem Fall wird ein IMR implizit erzeugt. Wenn man das Programm während der Bearbeitung stoppt, ist der aktuelle IMR in der lokalen Variable `imat` zu finden.

4.6 Algorithmus von Hand steuern

4.6.1 Im modularen Verfahren

Nachprüfen

Man berechnet die Elementarteiler zunächst mit dem schnellen modularen Verfahren. Danach möchte man sie noch einmal mit dem sicheren Verfahren nachprüfen. Das Programm kann bestimmte Zwischenergebnisse wiederverwenden und bei einem Aufruf mit dem IMR und nicht mit der Matrix erheblich Zeit sparen, wenn man folgendermaßen vorgeht:

```
gap> im:=IMRIMat([[2,1],[0,2]]);;
gap> ElementaryDivisorsIMat(im,rec(fastdet:=true,fastrank:=true));
[ 1, 4 ]
gap> ElementaryDivisorsIMat(im);
[ 1, 4 ]
```

Erleichtern

Man weiß zufällig den Rang der Matrix schon vorher. Dann kann man ihn mitteilen, um ggf. die Berchnung zu erleichtern.

```
gap> im:=IMRIMat([[2,1],[0,2]]);;
gap> im.rank:=2;,
gap> ElementaryDivisorsIMat(im);
[ 1, 4 ]
```

mehr Rangminoren

Wenn der Rangminoren–ggT zu groß ist, dann kann das daran liegen, daß zu wenig Rangminoren berechnet wurden. Ich zeige, wie man mehr Rangminoren berechnen lassen kann. Die erste Zuweisung ändert Default–Options — das wird später ausführlich beschrieben. Hier wird sie nur dazu benutzt, um den Fehler zu provozieren; denn der Default–Wert ist eigentlich 20.

```
gap> ElementaryDivisorsDefaultOptions.maxfivers:=1;,
gap> im:=IMRIMat(m3);;
gap> ElementaryDivisorsIMat(im);
Error, could not factor1145975385541822385457703 in
FactorsRho( n, 1, 16, 8192 ) called from
FactorsInt( AbsInt( det ) ) called from
ElDivIMR( imat ) called from
ElementaryDivisorsIMat( im ) called from
main loop
brk>
gap> ElementaryDivisorsIMat(im, rec( maxfivers:=20) );
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Am Exponenten gespart

```
gap> ElementaryDivisorsIMat([[2,1],[0,2^24]]);
#I    Rank guessing......fullrank
#I  Rank:2
#I  Considering 1 submatrices.
#I    Determinants calculating...GG...done
#I  Rankminor gcd multiple:33554432
#I    Factors:2^25
#I    Taking factor...2^14 2^15 2^16 2^17 2^18 2^19 2^20 2^21 2^22 2^23
2^24 2^25 2^26 ...done
```

```
#I  Elementary Divisors:[ 1, 33554432 ]
[ 1, 33554432 ]
gap> ElementaryDivisorsIMat([[2,1],[0,2^24]],rec(safeexp:=true));
#I    Rank guessing......fullrank
#I  Rank:2
#I  Considering 1 submatrices.
#I    Determinants calculating...GG...done
#I  Rankminor gcd multiple:33554432
#I    Factors:2^25
#I    Taking factor...2^26 ...done
#I  Elementary Divisors:[ 1, 33554432 ]
[ 1, 33554432 ]
```

Hier wird die Funktion des Parameters `safeexp` verdeutlicht. Im ersten Durchlauf wurde der modulare Elementarteileralgorithmus viele Male vergeblich durchgeführt, weil ein Primzahlmodul bevorzugt wurde, der ein GAP–immediate–integer ist. In den meisten Fällen ist aber die standardmäßige Vorgehensweise von Vorteil.

DefaultOptions ändern

Wenn man verschiedene Matrizen mit einem gleichen Satz von Parametern bearbeiten möchte, braucht man die Änderungen nicht für jede Matrix neu auszuführen. Man ändert dafür einmal den entsprechenden Parameter in `ElementaryDivisorsDefaultOptions`, `RankDefaultOptions` oder `DeterminantDefaultOptions`. Bei allen folgenden `IMRIMat`–Aufrufen (und Aufrufen, die diesen benutzen, wie `ElementaryDivisorsIMat`, `RankIMat` oder `IMat`) wird dann mit diesen Optionen gerechnet.

Beispiel:

```
gap> ElementaryDivisorsDefaultOptions.safeexp:=true;
gap> ElementaryDivisorsIMat([[2,1],[0,2^24]]);
#I    Rank guessing......fullrank
#I  Rank:2
#I  Considering 1 submatrices.
#I    Determinants calculating...GG...done
#I  Rankminor gcd multiple:33554432
#I    Factors:2^25
#I    Taking factor...2^26 ...done
#I  Elementary Divisors:[ 1, 33554432 ]
[ 1, 33554432 ]
```

4.6.2 Im nichtmodularen Verfahren

Im modularen Verfahren gibt es eine enorme Vielfalt, den Lauf des Programms zu beeinflussen; bei den meisten Situationen gibt es viele unterschiedliche Möglichkeiten, die Parameter während des Verlaufs zu ändern, wobei man nicht von vornherein sagen kann, was genau das Richtige in einer bestimmten Situation ist. Ich gebe hier nur einige Fälle an, aus denen man wichtige Vorgehensweisen in anderen Beispielen ableiten kann.

keine Reduktionen mehr

Wenn kein akzeptabler Pivot gefunden wird, dann werden normalerweise Zeilen– und Spaltenreduktionen versucht. Wenn auch nach diesen kein akzeptabler Pivot gefunden wurde, wird ein Best–Remainder–Schritt ausgeführt. Ich zeige, wie man erreicht, daß in diesem Falle auf die Reduktionen verzichtet wird.

Die Erzeugung der Beispielmatrix ist im Kapitel **Praxis** zu finden.

```
gap> InfoIMat1:=Print;;InfoIMat2:=Print;;InfoIMat3:=Print;;InfoIMat4:=Print;;
gap> ElementaryDivisorsIMat(h120,rec(algorithm:="preview"));
.
```

```
.
#I  predicted damage: 137327
#I But not taking it, intolerable damage!
#I   Bounds: tolerate:19% intolerate:52% pivots:29 descents:96 steps:3
#I Row reduction .....................................................................\
.............................................................
#I  realised damage:  -5652 (-3%)
#I   maximal entry:   1335, matrix norm: 145324, sparsity: 0%
#I Convinced of 9th descent, length 1, from 97: m[101][9]:=2 (col)
#I  predicted damage: 134927
#I But not taking it, intolerable damage!
#I   Bounds: tolerate:23% intolerate:62% pivots:28 descents:100 steps:2
#I Column reduction .............
#I  realised damage:  -44280 (-30%)
.
.
gap> ElementaryDivisorsIMat(h120,rec(algorithm:="preview",shuffles:=[0,0]));
.
.
#I  predicted damage: 137327
#I But not taking it, intolerable damage!
#I   Bounds: tolerate:19% intolerate:52% pivots:29 descents:96 steps:3
#I Try to isolate m[11][9]:=1
#I 9th element isolated.
#I  realised damage:  388247 (257%)
.
.
```

nicht so oft Backtracking

```
gap> ElementaryDivisorsIMat(h120,rec(algorithm:="preview"));
.
.
#I 9th element isolated.
#I  realised damage:  374020 (370%)
#I Taking the last step back
#I Row reduction .....................................................................\
.............................................................
#I  realised damage:  -4184 (-4%)
#I   maximal entry:   828, matrix norm: 96860, sparsity: 1%
.
.
gap> ElementaryDivisorsIMat(h120,rec(algorithm:="preview",avoidrate:=10));
.
.
#I 9th element isolated.
#I  realised damage:  374020 (370%)
#I   maximal entry:   7843, matrix norm: 475064, sparsity: 0%
#I Convinced of 10th descent, length 1, from 101: m[16][6]:=6 (col)
.
.
```

schnellere Reduktionen

Zuweilen wird unverhältnismäßig viel Rechenzeit für die Reduktionen verbraucht.

```
gap> ElementaryDivisorsIMat(h120,rec(algorithm:="preview",redurate:=1/5));
```

```
.
.
#I  predicted damage: 137327
#I But not taking it, intolerable damage!
#I   Bounds: tolerate:19% intolerate:52% pivots:29 descents:96 steps:3
#I Row reduction .........................
#I  realised damage:  -1665 (-1%)
#I   maximal entry:   1335, matrix norm: 149311, sparsity: 0%
#I Convinced of 9th descent, length 1, from 97: m[101][9]:=2 (col)
#I  predicted damage: 132919
#I But not taking it, intolerable damage!
#I   Bounds: tolerate:23% intolerate:62% pivots:28 descents:100 steps:2
#I Column reduction ...
#I  realised damage:  -23429 (-15%)
#I   maximal entry:   1029, matrix norm: 125882, sparsity: 0%
.
.
```

Umschalten von Preview und Best–Remainder

```
gap> im:=IMRIMat(h120);;
gap> ElementaryDivisorsIMat(im,rec(algorithm:="bestrem"));
#I   maximal entry:   634, matrix norm: 111414, sparsity: 10%
#I Try to isolate m[7][16]:=-1
#I 1st element isolated.
#I  realised damage:  89 (0%)
#I   maximal entry:   613, matrix norm: 111503, sparsity: 7%
#I Try to isolate m[97][15]:=1
#I 2nd element isolated.
#I  realised damage:  -2036 (-1%)
^C#I   maximal entry:   568, matrix norm: 109467, sparsity: 6%
Error, user interrupt at
until imat.done ... in
BRemIMR( imat ) called from
ElDivIMR( imat ) called from
ElementaryDivisorsIMat( im, rec(
    algorithm := "bestrem" ) ) called from
main loop
brk>
gap> ElementaryDivisorsIMat(im,rec(algorithm:="preview"));
#I  realised damage:  0 (0%)
#I   maximal entry:   568, matrix norm: 109467, sparsity: 6%
#I Tolerate 3rd descent, length 0, from 1: m[48][10]:=-1 (row)
#I  predicted damage: 0
#I   Bounds: tolerate:1% intolerate:48% pivots:29 descents:88 steps:4
#I   Isolate:===============
#I  realised damage:  -7166 (-6%)
#I   maximal entry:   532, matrix norm: 102301, sparsity: 5%
#I Persuaded of 4th descent, length 1, from 84: m[93][16]:=2 (row)
#I  predicted damage: 3495
#I   Bounds: tolerate:5% intolerate:48% pivots:32 descents:86 steps:5
#I   Isolate:==========
^CError, user interrupt at
for col ... in
ActualiseNorms( imat ) called from
DiagIMR( imat ) called from
```

```
ElDivIMR( imat ) called from
ElementaryDivisorsIMat( im, rec(
    algorithm := "preview" ) ) called from
main loop
brk>
gap> ElementaryDivisorsIMat(im,rec(algorithm:="bestrem"));
#I  realised damage:  -6986 (-6%)
#I   maximal entry:   424, matrix norm: 95315, sparsity: 6%
#I Try to isolate m[1][11]:=-1
#I 5th element isolated.
#I  realised damage:  3052 (3%)
```

sorgfältigere Pivotsuche

```
gap> im:=IMRIMat(h120,rec(algorithm:="preview"));;
gap> ElementaryDivisorsIMat(im,rec(toltop:=1/25,pivbot:=50,pivtop:=100,
> desbot:=150,destop:=250));
.
.
#I Tolerate 5th descent, length 0, from 152: m[1][11]:=-1 (row)
#I  predicted damage: 0
#I   Bounds: tolerate:3% intolerate:42% pivots:50 descents:154 steps:3
#I   Isolate:===============
#I  realised damage:  1592 (1%)
#I   maximal entry:   423, matrix norm: 97774, sparsity: 2%

.
.
```

schnellere Pivotsuche

```
gap> im:=IMRIMat(h120,rec(algorithm:="preview"));;
gap> ElementaryDivisorsIMat(im,rec(toltop:=1,tolup:=1/10,pivtop:=30,
> desbot:=55,destop:=70));
.
.
#I Tolerate 5th descent, length 1, from 2: m[98][16]:=-2 (col)
#I  predicted damage: 7406
#I   Bounds: tolerate:9% intolerate:42% pivots:29 descents:80 steps:5
#I   Isolate:=================================================================\
+++++++++++++++++++++++++++++++++++++++++++=+=++====
#I  realised damage:  26716 (27%)
#I   maximal entry:   2165, matrix norm: 122898, sparsity: 4%
.
.
```

4.7 IMR–Felder

4.7.1 allgemeine Felder

matrix Enthält die Matrix im gewöhnlichen GAP–Format — ggf. transponiert, damit die Zeilenzahl nicht kleiner als die Spaltenzahl ist.

transposed BOOLEscher Wert. Für die effektive Behandlung von asymmetrischen Matrizen möchte ich nur mit Matrizen arbeiten, deren Zeilenzahl nicht kleiner als die Spaltenzahl ist. Um nicht viel identischen Code doppelt zu haben, transponiere ich die Matrix vor der Berechnung — die Elementarteiler ändern sich dadurch nicht. Dieses IMR–Feld zeigt an, ob in `matrix` die originale oder transponierte

Matrix steht. Um Speicherplatz zu sparen, möchte ich die Matrix nicht kopieren, wenn ich sie nicht transponieren muß. Beim Wert `false` steht also in `matrix` nur ein Zeiger auf die Matrix, die bei der Erzeugung herangezogen wurde. Man darf also diese Matrix nicht verändern oder löschen, da sonst der IMR zerstört wird. Das ist nicht der Fall beim Wert `true`, da beim Transponieren eine neue Matrix erschaffen wird.

rows Zeilenanzahl der Matrix — nur zur Information.

cols Spaltenanzahl der Matrix.

minrank Nichtnegative ganze Zahl. Gibt untere Schranke für den Rang der Matrix an, ist am Anfang 0. Dieser Wert kann heraufgesetzt werden, wenn eine andere untere Schranke für den Rang bekannt ist. Der Wert wird in `RankIMR` benutzt.

maxrank Positive ganze Zahl. Gibt obere Schranke für den Rang der Matrix an, ist am Anfang gleich der Anzahl der Spalten (=`cols`), da die Zeilenzahl nicht kleiner als die Spaltenzahl ist. Dieser Wert kann herabgesetzt werden, wenn eine andere obere Schranke für den Rang bekannt ist. Der Wert wird in `HadamardIMR` benutzt.

4.7.2 Organisation der Parameter

Die Parameter stehen in drei Feldern des IMR: `RankOptions` (für die modulare Rangberechnung), `DeterminantOptions` (für die modulare Determinantenberechnung) und `ElementaryDivisorsOptions` (für modulare und nichtmodulare Verfahren zur Elementarteilerberechnung), die wiederum Records sind.

Die Felder des Options–Records, die beim Aufruf von `IMRIMat` oder `ElementaryDivisorsIMat` als Parameter übergeben werden, werden auf 3 Options–Records des IMRs aufgeteilt, wobei die meisten, nämlich `algorithm`, `maxfivers`, `safeexp`, `tolerate`, `tolbot`, `toltop`, `tolup`, `toldown`, `intolerate`, `intolbot`, `intoltop`, `intolup`, `intoldown`, `avoidrate`, `pivotbound`, `pivbot`, `pivtop`, `pivup`, `pivdown`, `descentbound`, `desbot`, `destop`, `desup`, `desdown`, `stepbound`, `stepbot`, `steptop`, `stepup`, `stepdown`, `shuffles` und `redurate` im Feld `ElementaryDivisorsOptions` steht, wogegen `fastrank` in `RankOptions` steht und `fastdet` in `DeterminantOptions`. `fasthadamard` wird gleichzeitig in `RankOptions` und `DeterminantOptions` übertragen[1]. Der wichtigste Parameter ist der, der die Art des Algorithmus angibt:

ElementaryDivisorsOptions.algorithm Bestimmt, welcher Algorithmus zur Berechnung der Elementarteiler genutzt werden soll: `"modular"` für das modulare Verfahren, `"bestrem"` für das nichtmodulare Best–Remainder–Verfahren, `"preview"` für das nichtmodulare Preview–Verfahren. Default–Wert: `"modular"`.

Man beachte, daß die Zwischenergebnisse des modularen Verfahrens nichts mit denen des nichtmodularen Verfahrens zu tun haben, so daß beispielsweise Zwischenergebnisse des modularen Verfahrens ignoriert werden, wenn man den Wert des Feldes `algorithm` auf `"bestrem"` ändert.

Die Zwischenergebnisse der beiden nichtmodularen Verfahren sind kompatibel, d. h. z. B. , daß man die Berechnung zwischen Best–Remainder–Verfahren und Preview–Verfahren hin– und herschalten kann und dabei stets die zuletzt berechnete Zwischenmatix weiterbearbeitet.

4.7.3 Parameter des modularen Verfahrens

Da das modulare Verfahren die modulare Rang– und Determinantenberechnung benutzt, werden hier Parameter aller 3 Options–Felder des IMRs benutzt.

RankOptions.fastrank BOOLEscher Parameter, also zwei mögliche Werte: `true` und `false`. Wenn `true`, dann wird der Rank mit der schnellen, unsicheren Methode bestimmt, also nur modulo einer Primzahl berechnet, ansonsten modulo sovieler Primzahlen, daß deren Produkt die HADAMARD–Schranke übertrifft. Ich habe die Verfahren im vorigen Kapitel diskutiert. Default–Wert: `false`.

RankOptions.fasthadamard BOOLEscher Parameter. Wenn `false`, wird mit einem aufwendigeren Verfahren eine bessere HADAMARD–Schranke für die Rangbestimmung berechnet. Default–Wert: `false`.

[1] Das hat zur Folge, daß mit Options–Records als zweites Funktionsargument von `IMRIMat` oder `ElementaryDivisorsIMat` die schnelle HADAMARD–Schranken–Berechnung für die modulare Rangberechnung einerseits und die modulare Determinantenberechnung andererseits nicht unabhängig voneinander geschaltet werden können.

DeterminantOptions.fastrank BOOLEscher Parameter. Wenn `true`, dann werden die Determinanten mit der schnellen, unsicheren Methode berechnet. Default–Wert: `false`.

DeterminantOptions.fasthadamard BOOLEscher Parameter. Wenn `false`, wird mit einem aufwendigeren Verfahren eine bessere HADAMARD–Schranke für die Determinantenbestimmung. Default–Wert: `false`.

ElementaryDivisorsOptions.maxfivers Positive ganze Zahl. Obere Grenze für die Anzahl von Rangteilmatrizen, die für den FiveStep (das synchrone Berechnen von Determinanten von Rangteilmatrizen, die bis auf die letzte Zeile identische sind) ausgewählt werden. Diese Grenze berücksichtigend wird ansonsten eine möglichst große Zahl von Rangteilmatrizen ausgewählt. Default–Wert: `20`.

ElementaryDivisorsOptions.safeexp BOOLEscher Parameter. Wenn `false`, wird für die Berechnung der modularen Elementarteiler eine Primzahlpotenz vorgezogen, die so klein ist, daß alle Ergebnisse von Multiplikationen in ein Datenwort passen — auf die Gefahr hin, daß die gewählte Potenz zu klein war, und die Rechnung mit einer höheren Primzahlpotenz wiederholt werden muß. Dieses Verfahren ist nicht unsicher, da ein Fehler zwingend erkannt wird. Default–Wert: `false`.

4.7.4 Parameter für den Preview–Algorithmus

ElementaryDivisorsOptions.tolerate Schaden, der kleiner als dieser Anteil bezüglich der Matrixnorm ist, wird als klein betrachtet, daß die entsprechende Isolierung sofort ausgeführt wird. Default–Wert: `1/50`.

ElementaryDivisorsOptions.<tolbot—toltop—tolup—toldown> Grenzen (`tolbot` nach unten und `toltop` nach oben) und Schrittmaß (`tolup` nach oben und `toldown` nach unten), mit denen der Parameter `tolerate` geändert wird. Default–Werte: `0,1/3,1/25,1/100`.

ElementaryDivisorsOptions.intolerate Schaden, der größer als dieser Anteil bezüglich der Matrixnorm ist, wird als so groß betrachtet, daß die entsprechende Isolierung nicht ausgeführt wird, obwohl sie die beste gefundene war. Statt dessen wird die Matrix gemischt. Default–Wert: `1/2`.

ElementaryDivisorsOptions.<intolbot—intoltop—intolup—intoldown> Grenzen und Schrittmaß, mit denen der Parameter `intolerate` geändert wird. Default–Werte: `2/5,4,1/10,1/50`.

ElementaryDivisorsOptions.avoidrate Wenn das soviel–fache des dem Parameter `intolerate` entsprechenden Schadens durch den real eingetretenen Schaden nach der erfolgten Isolierung überschritten wird, wird Backtracking eingeleitet. Default–Wert: `5/2`.

ElementaryDivisorsOptions.pivotbound Wenn soviele von Null verschiedene Pivotpositionen nach Abstiegen durchsucht worden sind, wird der bis dahin beste Abstieg genommen. Default–Wert: `30`.

ElementaryDivisorsOptions.<pivbot—pivtop—pivup—pivdown> Grenzen und Schrittmaß, mit denen der Parameter `pivotbound` geändert wird. Default–Werte: `24,46,3,1`.

ElementaryDivisorsOptions.descentbound Wenn soviele Abstiege berechnent worden sind, wird der bis dahin beste Abstieg genommen. Default–Wert: `90`.

ElementaryDivisorsOptions.<desbot—destop—desup—desdown> Grenzen und Schrittmaß, mit denen der Parameter `descentbound` geändert wird. Default–Werte: `74,100,6,2`.

ElementaryDivisorsOptions.stepbound Kein Abstieg wird berücksichtigt, der länger als diese Zahl ist. Default–Wert: `4`.

ElementaryDivisorsOptions.<stepbot—steptop—stepup—stepdown> Grenzen und Schrittmaß, mit denen der Parameter `stepbound` geändert wird. Default–Wert: `2,6,1,1`.

ElementaryDivisorsOptions.shuffles Um das Mischen zu steuern, wird ein Zähler `shufflecount` geführt, der bei jedem Mischen inkrementiert wird und bei jeder gelungenen Suche nach einem Abstieg (die nicht durch `intolerate` verhindert wurde) auf Null rückgesetzt wird. Wenn nun gemischt werden soll, dann wird dieser Zähler mit den Werten dieser Liste verglichen. Ist er kleiner als der erste Wert, wird eine Zeilenreduktion vorgenommen; ist er nicht kleiner als der erste aber kleiner als der zweite, wird eine Spaltenreduktion ausgeführt; ist er aber nicht kleiner als alle beide, wird ein Best–Remainder–Schritt zur Isolierung getan. Der Wert `[1,2]` bedeutet also, daß vor dem Best–Remainder–Schritt noch eine Zeilen– , dann eine Spaltenreduktion ausgeführt wird; `[0,0]` bedeutet, daß niemals reduziert werden soll. Default–Wert: `[1,2]`.

ElementaryDivisorsOptions.redurate Die maximale Zahl von Durchläufen für eine Zeilenreduktion ist das Produkt aus dieser Zahl und der Anzahl der Zeilen. Die maximale Zahl von Durchläufen für eine Spaltenreduktion ist das Produkt aus dieser Zahl und der Anzahl der Spalten. Default–Wert: `1`.

ElementaryDivisorsOptions.isom Falls `true`, dann werden die Transformationmatrizen berechnet und ausgegeben. Default–Wert: `false`.

4.7.5 DefaultOptions

Bei jedem Aufruf von `IMRIMat` oder `ElementaryDivisorsIMat`, wenn einer neuer IMR erzeugt wird, dann werden alle Parameter, die nicht explizit im Funktionsargument mit angegeben wurde, mit Default–Werten gefüllt, und zwar werden alle Werte aus `ElementaryDivisorsDefaultOptions` in das Feld `ElementaryDivisorsOptions` des IMRs kopiert, aus `RankDefaultOptions` in `RankOptions` und aus `DeterminantDefaultOptions` in `DeterminantOptions`.

Sollen Default–Werte geändert werden, so kann das mit Zuweisungen an die Felder dieser Records geschehen. Jedesmal, wenn das IMat–Programm neu geladen wird, werden diese Werte wieder mit den ursprünglichen Defaults überschrieben. Mit veränderten Defaults können die in diesem Kapitel angegebenen Beispiele andere Ergebnisse ergeben.

4.7.6 Zwischenergebnisse des modularen Verfahrens

fivehyp Record mit 5 Feldern, die je eine Liste enthalten. Definiert eine Menge von möglichen synchron berechenbaren Rangteilmatrizensätzen. Die 5 Listen des Records sind gleich lang, die i–ten Elemente jeder Liste definieren den i–ten Rangteilmatrizensatz, der aus f_i Rangteilmatrizen bestehe. Ich beschriebe die Felder im einzelnen mit Sicht auf den i–ten Eintrag jeder Liste. Sei r der Rang der Matrix[2].

fivehyp.rows[i] Liste von $r-1$ Zeilenindices. Diese Zeilen bilden die gemeinsamen Zeilen der Rangteilmatrizen des i–ten Rangteilmatrizensatzes.

fivehyp.cols[i] Liste von r Spaltenindices. Diese Spalten bilden die Spalten der Rangteilmatrizen des i–ten Rangteilmatrizensatzes.

fivehyp.lastrows[i] Liste von f_i Zeilenindices. Jede dieser Zeilen bildet zusammen mit den Zeilen, die durch `fivehyp.rows[i]` definiert sind, eine Rangteilmatrix.

fivehyp.modules[i] Primzahl p_i, modulo der der i–te Rangteilmatrizensatz bestimmt wurde.

fivehyp.dets[i] Liste der f_i p_i–modularen Determinanten der Rangteilmatrizen des i–ten Rangteilmatrizensatzes.

zeroprimes Liste von Primzahlen p, daß der p–modulare Rang der Matrix kleiner als der ganzzahlige Rang ist.

minrank Nichtnegative Zahl. Der Rang der zunächst als gültig angesehen wird. Der wirkliche Rang kann nicht kleiner als `minrank` sein.

bound HADAMARD–Schranke für alle Teilmatrizen der Matrix als ganze Zahl.

[2]der zum Zeitpunkt als gültig angesehen wird, wo dieser `fivehyp` als Feld im IMR existiert. Wird dieser als gültig angesehene Rang als falsch erkannt, wird auch der gesamte `fivehyp` verworfen. Alle hier definierten Rangteilmatrizensätze beziehen sich auf den selben Rang, der auch in `minrank` steht.

rank Rang der Matrix als ganze Zahl. Bei der sicheren Rangberechnung wird dieses Feld belegt, wenn der Rang modulo allen nötigen Primzahlen bestätigt wurde — er ist dann mit Sicherheit richtig. Bei der schnellen Rangberechnung wird dieses Feld belegt, wenn der Rang modulo einer Primzahl nicht kleiner als `minrank` war — er kann dann also durchaus falsch sein.

fivemat Record mit 3 Feldern, die den Rangteilmatrizensatz definieren, der zur Berechnung des Rangminoren–ggTs herangezogen wird. Sei r der Rang der Matrix, f die Anzahl der Rangteilmatrizen des Satzes,

fivemat.rows Liste von $r-1$ Zeilenindices. Diese Zeilen bilden die gemeinsamen Zeilen der Rangteilmatrizen des Satzes.

fivemat.cols Liste von r Spaltenindices. Diese Spalten bilden die Spalten der Rangteilmatrizen des Satzes.

fivemat.lastrows Liste von f Zeilenindices. Jede dieser Zeilen bildet zusammen mit den Zeilen, die durch `fivemat.rows` definiert sind, eine Rangmatrix.

fiveprimes Liste von Primzahlen, modulo derer die Determinanten des gültigen Rangteilmatrizensatzes bereits berechnet wurden.

fivedets Liste von Listen von modularen Determinanten. Sei p_i die i–te Primzahl in `fiveprimes`; dann steht in `fivedets[i][j]` die p_i–modulare Determinante der j–ten Teilmatrix des durch `fivemat` definierten Satzes.

det Das aus diesem Prozeß gewonnene Vielfache des Rangminoren–ggTs.

factors Liste von Zweierlisten von ganzen Zahlen, die dem Ergebnis des GAP–Aufrufes `Collected(FactorsInt(det))` entspricht, also der Faktorisierung des Rangminoren–ggTs.

eldivconst Liste von Records, die Primzahlpotenz–modulare nichttriviale Elementarteiler definieren. Ich beschriebe die Felder im einzelnen mit Sicht auf den i–ten Eintrag der Liste.

eldivconst[i].prime Die Primzahl, deren Potenz als Modul verwendet wird.

eldivconst[i].exp Die höchste Potenz, die als modularer Elementarteiler auftritt.

eldivconst[i].list Liste der r Elementarteiler aufsteigend der Größe nach geordnet, wobei r der Rang der Matrix ist.

noteldiv Liste von Primzahlen, deren Potenz zwar in der Faktorisierung von `det` auftritt, modulo derer die Matrix aber nur triviale modulare Elementarteiler besitzt

eldivfail Liste von Records, die Primzahlpotenz–modulare nichttriviale Elementarteiler anzeigen, aber noch nicht explizieren. Sie entstehen, wenn die Potenz der Primzahl zu klein gewählt wurde und einige Elementarteiler die Primzahl in einer noch höheren Potenz enthalten. So muß der modulare Elemnentarteileralgorithmus für diese Primzahl mit einer höheren Potenz wiederholt werden. Ich beschriebe die Felder im einzelnen mit Sicht auf den i–ten Eintrag der Liste.

eldivfail[i].prime Die Primzahl, deren Potenz als Modul verwendet wird.

eldivfail[i].exp Die Potenz der Primzahl, die sich als zu klein erwiesen hat.

eldivisors Liste der r Elementarteiler der Matrix aufsteigend der Größe nach geordnet, wobei die trivialen Elementarteiler als 1 angegeben werden und r der Rang der Matrix ist.

4.7.7 Zwischenwerte der nichtmodularen Verfahren

amat Die aktuelle Matrix, an der die Änderungen direkt vorgenommen werden. Am Anfang wird sie aus `matrix` kopiert, wird im Verauf durch das Entfernen der Zeilen und Sp[alten bereits isolierter Elemente immer kleiner und wird gestrichen, wenn sie leer ist.

step Gibt an, das wievielte Diagonalelement gerade isoliert wird.

diags Hierhinein werden die isolierten Diagonalelemente gesammelt.

div Wenn die ganze Matrix durch eine Zahl geteilt wird, dann wird `div` mit dieser Zahl multipliziert. Alle kommenden Diagonalelemente müssen dann mit `div` multipliziert werden, bevor sie in `diags` aufgenommen werden.

left, right Hier werden die Transformationsmatrizen aufgebaut.

done Wird gesetzt, wenn die Matrix vollständig isoliert ist.

rownorm, colnorm Die Zeilen– und Spaltennormen.

rowdiv, coldiv Die Zeilen– und Spaltenteiler.

rowsort, colsort Die nach der Norm sortierten Zeilen– und Spaltenindeces.

matnorm Die Norm über die ganze Matrix.

oldmat, oldleft, oldright, olddiags, oldstep, olddiv Hier werden die entsprechenden Werte für das Backtracking zwischengespeichert.

age Wenn Backtracking ausgeführt werden mußte, dann wird diese Variable um 1 hochgesetzt. Am Anfang ist sie 0, und wenn sie 3 erreicht, dann wird die Preview–Strategie aufgegeben und statt dessen mit Best–Remainder–Strategy weitergerechnet.

shufflecount Zähler, der unter `ElementaryDivisorsOptions.shuffles` beschrieben wurde.

4.8 Info-System

Man will die Elementarteiler einer ganzzahligen Matrix berechnen. Wenn die Matrix klein, das Beispiel einfach ist, dann wird nach kurzer Zeit das Ergebnis ausgegeben. Wenn die Berechnung länger dauert, dann stellt sich sofort die Frage: Gibt er jetzt gleich das Ergebnis aus? Oder morgen? Oder niemals? Um einen Anhaltspunkt dafür zu geben, informiert das Programm über den gegenwärtigen Stand der Berechnung. Ein gutes Programm läßt den Nutzer nicht im Unklaren. Doch auch für den Programmierer ist ein gutes Info-System eine unschätzbare Hilfe für das Debugging.

Ich unterscheide grob zwei Arten von Informationsausgaben, die ich unabhängig voneinander schalten möchte: Einerseits Ausgaben, die den Nutzer in regelmäßigen Abständen darüber informieren, wie weit das Programm mit seiner Berechnung ist, um einschätzen zu können, ob es sich zu warten lohnt, oder ob vielleicht ein anderer Algorithmus zu probieren ist. Andererseits die Ausgabe von Zwischenergebnissen, damit man einschätzen kann, ob die gegenwärtige Einstellung der Parameter ungünstig ist, und sie so zu ändern, daß das Programm effektiver rechnet.

Wie auch bei vielen anderen GAP–Modulen schaltet man das Info System mit `InfoIMat1:=Print;` und `InfoIMat2:=Print;` ein und mit `InfoIMat1:=Ignore;` und `InfoIMat2:=Ignore;` aus.

`InfoIMat1` gibt nur eine Nachricht aus, wenn `ElementaryuDivisorsIMat`, `RankIMat` oder `DeterminantIMat` gerade mit ihrer Berechnung beginnen oder gerade damit fertig sind.

`InfoIMat2` gibt wichtige Zwischenergebnisse aus.

`InfoIMat3` ist viel geschwätziger und gibt ein Fülle von Informationen mit mehr oder weniger großem Nutzen aus.

4.8.1 Infos des modularen Verfahrens

InfoIMat2: Zwischenwerte–Information

`#I Rank:12`
Das ist der Rang, mit dem er weiterrechnet, egal ob er sicher ist oder nicht (`RankIMR`).

`#I Remembered Rank:12`
Das ist der Rang, der schon in der Datenstruktur stand. Wenn man ihm nicht traut, dann lösche man ihn manuell mit `Unbind(imat.rank);` oder man starte neu mit einem `IMRIMat`-Aufruf (`RankIMR`).

`#I Considering 20 submatrices.`
So viele Rangminoren werden berechnet (`FiveSelectIMR`).

`#I Improved Hadamard bound:10^99`
Die HADAMARD–Schranke , die für die Rangberechnung genommen wird (`HadamardIMR`).

`#I Hadamard bound:10^134`
Das ist die schneller zu berechnende aber größere HADAMARD–Schranke , wenn `fasthadamard` gesetzt ist (`HadamardIMR`).

`#I New Hadamard bound:10^73`
Das ist die HADAMARD–Schranke , die für die Determiantenberechnung benutzt wird. Sie gilt für alle Submatrizen, deren Determinante berechnet wird (`FiveHadamardIMR`).

`#I Remembered bound:10^134`
Hier wird eine schon berechnete HADAMARD–Schranke aus der Datenstruktur entnommen (`HadamardIMR`, `FiveHadamardIMR`).

`#I  Considering 12 primes.`
Das Produkt von so vielen Primzahlen übersteigt die HADAMARD–Schranke (`DetIMR`).

`#I  Rankminor gcd:3814791552`
berechnet von `DetIMR`.

`#I  Remembered rankminor gcd:3814791552`
falls es schon vorher berechnet wurde (`DetIMR`).

`#I  Elementary Divisors:[ 1, 1, 1, 1, 1 ]`
Das Ergebnis von `ElDivModIMR`.

`#I  Remembered Elementary Divisors:[ 1, 1, 1, 1, 1 ]`
Das Ergebnis aus einem früheren Durchlauf.

InfoIMat3: Ich rechne!–Information

`#I Rank guessing... 54%`
Der Rang wird gerade berechnet, und zwar mit der unsicheren Methode (`RankIMR,ModRankIMR`). Die Prozentangabe bezieht sich auf eine primzahlmodulare Rangbestimmung, in diesem Fall wurde schätzungsweise 54 Prozent einer Rangbestimmung erledigt. Die Zahl ergibt sich aus dem Quotienten der Anzahl der schon bearbeiteten Zeilen zu der Anzahl aller Zeilen der Matrix.

`#I Rank guessing......done`
Jetzt ist er fertig damit (`RankIMR`).

`#I Rank guessing......fullrank`
Hier kann man sogar sicher sein, daß der Rang richtig ist (`RankIMR`).

`#I Rank confirming...G 33%`
Der Rang wird gerade bestätigt — das ist die sichere Methode. Fehlt diese Zeile, wurde der Rang mit der schnlellen Methode berechnet (`RankIMR,ModRankIMR`). Jedes **G** steht für einen modularen GAUSS–**Algorithmus** .

`#I Rank confirming...GGGGG...done`
Jetzt ist er fertig damit (`RankIMR`). Wenn er jetzt nicht wieder anfängt, den Rang zu raten, hat er jetzt den Rang sicher bestimmt. Ansonsten ist man bei der Bestätigung auf eine Primzahl gestoßen, die einen größeren Rang als den zuerst erratenen hat.

`#I Rank confirming...GG...fullrank`
Die Matrix hat vollen Rang, das wurde aber nicht schom beim Raten mit der ersten Primzahl herausgefunden (`RankIMR`).

`#I Hadamarding rows`
Er berechnet gerade die HADAMARD–Schranke aus den Zeilennormen (`HadamardIMR`).

`#I Hadamarding ...done`
Jetzt ist er fertig damit (`HadamardIMR`).

`#I Determinants calculating...GG 45%`
Er berechnet gerade Determinanten (`DetIMR,ModFiveStepIMR`).

`#I Determinants calculating......done`
Jetzt ist er fertig damit (`DetIMR`).

`#I Taking factor...2^14 3^3 56%`
Er rechnet gerade die Elementarteiler modulo 3^3 aus (`ElDivModIMR,ModElDivIMR`).

`#I Taking factor...2^14 3^3 7^2 19^2 109^2 ...done`
Jetzt ist er fertig damit.

`#I Taking factor...2^14 2^15 2^16 21%`
Der Parameter `safeexp` ist `false` und jetzt hat er einen Exponenten gewählt, der zu klein war. So muß er jetzt mit einer größeren Potenz wiederholen.

4.8.2 Infos der nichtmodularen Verfahren

InfoIMat2: Besondere Ereignisse

Hier wird im wesentlichen für jeden isolierten Pivot ein Punkt gezeichnet. Folgende Situationen ergeben aber eine richtige Nachricht:

`#I Row reduction`
Zeilenreduktion.

`#I Column reduction`
Spaltenreduktion.

`#I whole rest of matrix divided by 4`
Alle Einträge der Matrix waren durch 4 teilbar.

`#I Taking the last step back`
Nach der Isolierung hat sich der Schaden als so groß erwiesen, daß Backtracking gemacht werden mußte (`avoidrate`).

`#I Giving up Preview Strategy`
Jetzt arbeitet er mit Best Remainder Strategy weiter.

InfoIMat3: Die Isolierungen, Größen und Schäden

`#I Convinced of 1st descent, length 3, from 101: m[18][14]:=72 (row)`
Der Pivot an der Position (18,14) mit der Zeile als primären Vektor wurde zusammen mit einem Abstieg der Länge 3 (3 ggT–Transformationen) ausgewählt, weil genug Abstiege durchgerechnet wurden (`descentbound`). 101 Abstiege wurden durchgerechnet. Das war der erste Pivot der Matrix, der isoliert wurde.

`#I Persuaded of 2nd descent, length 1, from 55: m[11][6]:=2 (col)`
Hier wurde ausgewählt, weil genug Pivotpositionen durchforstet wurden (`pivotbound`). Die Zuordnung des primären Vektors war die die Spalte.

`#I Tolerate 3rd descent, length 0, from 1: m[2][31]:=-1 (row)`
Hier war der vorhergesagte Schaden klein genug. Ein Abstieg der Länge 0 bedeutet, daß der Pivot einfach isolierbar vom Typ 1 war (`tolerate`).

`#I But not taking it, intolerable damage!`
Ein berechneter Abstieg wird nicht genommen, da der vorhergesagte Schaden zu groß war (`intolerate`).

`#I Try to isolate m[18][20]:=1`
Eine Isolierung mit dem Best–Remainder–Verfahren wird versucht (`TryIsolateIMR`).

`#I 12th element isolated.`
Sie war erfolgreich.

`#I   maximal entry:   216, matrix norm: 6611, sparsity: 82%`
Allgemeine Informationen über die Matrix, mit Sparsity ist der Anteil der Nulleinträge gemeint.

`#I  predicted damage: 97900`
Vorhergesagter Schaden.

`#I  realised damage:  22811 (1851%)`
Das Anwachsen der Matrixnorm absolut und prozentual.

InfoIMat4: Sonstiges

`#I   Bounds: tolerate:1% intolerate:48% pivots:17 descents:88 steps:4`
Das sind die veränderlichen Parameter mit ihren aktuellen Wertnn.

`#I   Isolate:++=++===+=====`
Bei einem `=` wurde wie vorhergesagt mit dem Pivotvektor reduziert, bei `+` wurde etwas besseres zum Reduzieren gefunden. Jedes Zeichen steht für einen eliminierten Wert.

`#I Row reduction ......`
Zeilenreduktion. Für jeden Durchlauf wird ein Punkt gezeichnet.

`#I Column reduction .............`
Spaltenreduktion.

Kapitel 5

Praxis

In diesem Kapitel gebe ich Testbeispiele meiner Programme: wie sie in GAP erzeugt wurden, welche Elementarteiler ausgerechnet werden, wieviel Rechenzeit von meinen Programmen benötigt wurde. Daneben vergleiche ich die Rechenzeit mit Ergebnissen anderer Programme, die die Elementarteiler berechnen.

5.1 Gruppentheoretische Matrizen

Unter diesem Unterpunkt fasse ich die Beispiele zusammen, die aus gruppentheoretischen Berechnungen hervorgehen.

Zu den jeweiligen Zeitvergleichen sind folgende Bemerkungen zu machen:

- Die Einträge für **schnell modular, sicher modular, Preview, Preview mit Trafo, Best Remainder, Best Remainder mit Trafo** beziehen sich auf die in meinem Programm implementierten Funktionen.

- GAP **3.4** bezieht sich auf die in der GAP–Version 3.4 implementierte Funktion ElementaryDivisorsMat. Sie realisiert ein Verfahren, das ich in meiner Arbeit als KANNAN–BACHEM–Verfahren bezeichnet habe.

- **Havas modular, Havas nichtmodular** beziehen sich auf das Programms von GEORGE HAVAS [3]. Dieses Programm ist eine FORTRAN–Implementation eines modularen und eines nichtmodularen Algorithmus. Der modulare Algorithmus ist in 2 getrennt aufzurufenden Prozeduren realisiert: das Berechnen des Moduls für die Elementarteilerberechnung und die eigentliche modulare Elementarteilerberechnung.

- **Havas neu** bezieht sich auf GEORGE HAVAS neue Implementation eines nichtmodularen Verfahrens, **Havas neu shortint** bezieht sich auf eine Version dieses neuen Programms, die nur mit kurzen Zahlen rechnet, um Rechenzeit zu sparen. Bei dieser Version kann es zum **overflow** kommen.

- Zahl in Klammern hinter einer Zeitangabe. Ich benutze normalerweise (d.h. wenn keine Zahl in Klammern steht) für GAP–Programme die in der Umgebungsvariable time eingespeicherte Zeit, auf einer DEC station 5000/120 gerechnet. Für die auf anderen Rechnern laufenden Programme gebe ich einen Vergleichsfaktor bezüglich der Rechenleistung der Maschine in GAP–stones an. Normalerweise rechne ich auf einer DECstation mit etwa 25000 GAP–stones, gebe Faktor 4 für eine DEC Alpha 3000/500 mit etwa 100000 GAP–stones, Faktor 1/2 für eine HPstation mit 12000 GAP–stones (Auf dieser Maschine benutze ich die UNIX-Prozeßzeit. Darum sind kleine Zeitwerte sehr ungenau.) und eine Sparc II station mit 25000 GAP–stones.

- Die Angabe von ∞ anstatt einer Zeit besagt, daß trotz langem Warten kein Fortschreiten in der Berechnung mehr beobachtet werden kann. Bei GAP 3.4 habe ich z. B. irgendwann angehalten, und einen Eintrag <<an integer too large to be printed>> gesehen.

5.1.1 Campbell(3), 125*6-Matrix

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> gens := [ a, b ];;
gap> G := Group (gens, IdWord);;
gap> G.relators := [ a^2*b*a*b*a^-3*b^-1, b^2*a*b*a*b^-3*a^-1];;
gap> H := Subgroup ( G, [ ] );;
gap> cam3 := RelatorMatrixAbelianizedSubgroup( G, H );;
gap> ElementaryDivisorsIMat(cam3);
[ 1, 1, 1, 1, 1, 1 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	160ms	
	sicher modular	171ms	
	Havas modular	1730ms	HP station (1/2)
nichtmodular	Preview	7562ms	
	Best Remainder	1258ms	
	GAP 3.4	786ms	
	Havas nichtmodular	400ms	HP station (1/2)
	Havas neu shortint	20ms	DEC Alpha (4)
	Preview mit Trafo	16542ms	
	Best Remainder mit Trafo	5129ms	

5.1.2 Macdonald G(3,3), 22*7-Matrix

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> gens := [ a, b ];;
gap> G := Group (gens, IdWord);;
gap> G.relators :=[b^-1*a^-1*b*a*b^-1*a*b*a^-3,a^-1*b^-1*a*b*a^-1*b*a*b^-3];;
gap> H := Subgroup ( G, [ ] );;
gap> g33 := RelatorMatrixAbelianizedSubgroup( G, H );;
Error, the coset enumeration has defined more than 64000 cosets.
Type 'return;' if you want to continue with a new limit of
128000 cosets, type 'quit;' if you want to quit the coset enumeration,
type 'maxlimit := 0; return;' in order to continue without a limit
in
CosetTableFpGroup( F, TrivialSubgroup( F ) ) called from
RelatorMatrixAbelianizedNormalClosure( G, H ) called from
main loop
brk> return;
gap> ElementaryDivisorsIMat(g33);
[ 1, 1, 1, 1, 1, 1, 1 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	274ms	
	sicher modular	1418ms	
	Havas modular	1550ms	HP station (1/2)
nichtmodular	Preview	867ms	
	Best Remainder	317ms	
	GAP 3.4	179ms	
	Havas nichtmodular	270ms	HP station (1/2)
	Havas neu shortint	50ms	DEC Alpha (4)
	Preview mit Trafo	2067ms	
	Best Remainder mit Trafo	336ms	

5.1.3 Heineken, Untergruppe Index 60, 80*21-Matrix

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> c := AbstractGenerator("c");;
gap> gens := [ a, b, c ];;
gap> G := Group (gens, IdWord);;
gap> G.relators := [
>  a^-1 * Comm( b, Comm( b, c ) ),
>  b^-1 * Comm( c, Comm( c, a ) ),
>  c^-1 * Comm( a, Comm( a, b ) )
> ];;
gap> H := Subgroup ( G, [
>  c * b * a,
>  c * b^-1 * c^-1 * a^-1 * c^-1,
>  c * b^-1 * a^-1 * b^-1 * c^-1 * b * c^-1,
>  c * b^-1 * a^-1 * c^-1 * b,
>  c * b^-1 * a^-1 * b * a * b,
>  c * b^-1 * a^-1 * b * c * a * c^-1,
>  a^-2 * c^-1 * a * b^-1 * a * b * c^-1,
>  a^-1 * c^-1 * b^-2 * a * b * c^-1,
>  a * c^-1 * a^-1 * b^-1 * a^-1,
>  a * c^-1 * b^-1 * a^-1 * c,
>  a * c^-1 * b^-1 * c^-1 * a^-1 * c * a^-1,
>  c * b^-1 * a^-1 * b * c^-1 * a * b * a^-1,
>  b * a^-1 * b^-1 * c^-1 * b^-1,
>  a^-1 * b * a^-1 * b^-1 * a * b^-1,
>  b^-1 * a * b * a^-1 * c^-1,
>  a * b^-1 * a * c * b^2 * a^-1 ] );;
gap> h60:= RelatorMatrixAbelianizedSubgroupRrs( G, H );;
gap> ElementaryDivisorsIMat(h60);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	6652ms	
	sicher modular	9660ms	
	Havas modular	4450ms	HP station (1/2)
nichtmodular	Preview	86162ms	
	Best Remainder	9839ms	
	GAP 3.4	7570ms	
	Havas nichtmodular	6960ms	HP station (1/2)
	Havas neu shortint	100ms	DEC Alpha (4)
	Preview mit Trafo	135343ms	
	Best Remainder mit Trafo	17284ms	

5.1.4 Heineken, Untergruppe Index 120, 139*20-Matrix

```
gap> H := Subgroup ( G, [ a^5, b^5, b * a * b^-1 * b^-1,
>  a * a * a * b * a * b * a^-1 * a^-1 * b * a * b^-1 * a^-1 * b,
>  a * a * b * a^-1 * b * a * b * a^-1 * b * a * b^-1 * a * b^-1 ]);;
gap> h120:= RelatorMatrixAbelianizedNormalClosureRrs( G, H );;
gap> ElementaryDivisorsIMat(h120);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	6754ms	
	sicher modular	11230ms	
	Havas modular	4800ms	HP station (1/2)
nichtmodular	Preview	236485ms	
	Best Remainder	22549ms	
	GAP 3.4	21284ms	
	Havas nichtmodular	20420ms	HP station (1/2)
	Havas neu shortint	150ms	DEC Alpha (4)
	Preview mit Trafo	255714ms	
	Best Remainder mit Trafo	60844ms	

5.1.5 Heineken, Untergruppe Index 960, 1341*382-Matrix

```
gap> H := Subgroup ( G, [
>  a^5,
>  b^5,
>  ( a * b^-1 * b^-1 )^2,
>  a * a * b * a * b * a^-1 * a^-1 * b * a * b^-1 * a^-1 * b,
>  a * b * a^-1 * b * a * b * a^-1 * b * a * b^-1 * a * b^-1 ] );;
gap> h960 := RelatorMatrixAbelianizedNormalClosure( G, H );;
```

Diese Matrix ist im Gegensatz zu den vorigen ein relativ schweres Beispiel. Bei meinen Experimenten ist allein das schnelle modulare Verfahren zu einem Ergebnis gekommen. Die Matrix hat Rang 382 mit den Elementarteilern 4,4,2,2,2,2,2,2,2,2,2,1,1, . . . ,1. Der Rang ist sicher, da 382 auch der maximal mögliche Rang ist. Auch die Determinantenberechnung hat ein extrem geringe Fehlerwahrscheinlichkeit, da 20 Sätze von Rangminoren ausgewählt wurden. Ich mußte die Berechnung über mehrere Sitzungen verteilen, wobei etwa 20 Stunden für die Berechnung notwendig waren (zum Vergleich mit der noch folgenden Aufstellung der Rechenzeiten).

Noch interessanter ist an dieser Matrix, wie sich die nichtmodularen Verfahren verhalten; ich habe es im letzten Kapitel **Fazit** nacherzählt. Keines der nichtmodularen Verfahren kam zu einem Ergebnis.

Wie weit die einzelnen Verfahren dennoch kamen, habe ich in folgender Tabelle zusammengestellt, in der ich aufgelistet habe, nach der wievielten Isolierung (oder der wievielten Rekursion über die Matrixgröße) der maximale Eintrag der jeweiligen Zwischenmatrizen eine bestimmte Koeffizientengröße überschritten hat:

Verfahren	GAP 3.4	Best Remainder	Preview
maximaler Koeffizient größer als	nach Isolierung Nummer		
10^5	132	131	132
10^6	133	132	139
10^7	134	133	140
$268 * 10^6$	135	134	141
10^{10}	136	136	145
10^{12}	136	137	146
10^{20}	137	141	149

$268 * 10^6$ *ist die Grenze von* GAP*–immediate–integers.*

Mein kombiniertes Preview–Programm hat bei der 140. Isolierung auf die Best–Remainder–Strategy umgeschaltet. Das Programm in GAP 3.4 konnte den 140. Pivot nicht mehr isolieren, während die anderen beiden Algorithmen (die ja nun den selben Algorithmus ausführten) nach gewissen (allerdings immer länger werdenden) Zeitabschnitten erfolgreich isolierten.

Auffällig war jedoch, daß das Steigen der Matrixnorm beim Best–Remainder–Verfahren nach der Isolierung des 125. Pivot keineswegs gleichmäßig verlief, sondern extremen Schwankungen unterworfen war, einige Schritte[1] bewirkten Verhundertfachung, andere Hundertstelungen der Matrixnorm. Schritte, die eine Isolierung brachten, hatten durchweg besonders hohen Schaden. Eine Isolierung wurde in etwa fünf bis zehn Schritten erreicht, wobei die ersten entweder sehr geringen Schaden, oder aber auch hohen negativen Schaden (Verkleinerung der Matrixnorm) hatten; die letzten jedoch meist hohen Schaden hatten.

Bis zur 140. Isolierung hatte der Preview–Algorithmus noch eine relativ deutlich kleinere Matrixnorm als Best–Remainder aufzuweisen. Ab etwa der 120. Isolierung brauchte er dafür aber auch erheblich mehr Rechenzeit. In den starken Schwankungen der Matrixnorm nach dem 140. Schritt machte dieser Vorsprung nach etwa der 155. Isolierung keinen großen Unterschied mehr aus, war aber bei der 164. Isolierung, bei der ich den Test abgebrochen habe, noch erkennbar.

5.1.6 Heineken, Untergruppe Index 3840, 4778*939-Matrix

```
gap> H := Subgroup ( G, [
>  a * b * b * b * a * b^-1 * b^-1,
>  a * a * a * a * b * b * a^-1 * b * b,
>  a * a * a * a * a * b^-1 * b^-1 * b^-1 * b^-1 * b^-1,
>  a * b * a^-1 * b * a * b * a^-1 * b * a * b^-1 * a * b^-1,
>  a * a * b * a * a * b^-1 * a * b * a^-1 * a^-1 * b^-1 * a^-1 * a^-1 * b^-1
> ] );;
gap> h3840 := RelatorMatrixAbelianizedNormalClosureRrs( G, H );;
```

Dieses Beispiel ist noch ganz erheblich schwerer als das letzte Beispiel. Ich denke aber, wenn man lange genug wartet, kommt das schnelle modulare Verfahren zum Ziel. Ich habe die Berechnung mit Rücksicht auf die anderen am Lehrstuhl abgebrochen, nachdem voller Rang bestätigt wurde, und 16 Sätze von 20 Rangminoren berchnet wurden. Für einen Satz Determinanten brauchte er etwa 100 Minuten.

5.1.7 Fournelle, Untergruppe Index 42336, 1062*79-Matrix

```
gap> a := AbstractGenerator("a");; b := AbstractGenerator("b");;
gap> gens := [ a, b ];;
gap> G := Group (gens, IdWord);;
gap> G.relators := [
> a^2,
> b^7,
> Comm(a,a^b),
```

[1] Als einen Schritt bezeichne ich das suchen eines Pivots und das Eliminieren *oder* Reduzieren der Pivotzeile und Pivotspalte. Nicht in jedem Schritt wird auch ein Pivot isoliert; nach einem Schritt, der keine Isolierung brachte, kann aber ein kleinerer Pivot für den nächsten Schritt gefunden werden.

```
> Comm(a,a^(b^2))*(a^b)^-1,
> ];;
gap> H := Subgroup ( G, [ (a*a^(b^3))^3 ] );;
gap> fourn:= RelatorMatrixAbelianizedNormalClosureRrs( G, H );;
gap> ElementaryDivisorsIMat(fourn);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	512s	
	sicher modular	4475s	
	Havas modular		Zu große Determinante.
nichtmodular	Preview	7565s	
	Best Remainder	1123s	
	GAP 3.4	662s	
	Havas nichtmodular	∞	
	Havas neu shortint	6.9s	DEC Alpha (4)
	Preview mit Trafo	9605s	
	Best Remainder mit Trafo	2304s	

Im modularen Havas–Programm wurde bei der Determinantenberechnung ein Ergebnis ausgegeben, das sich nicht für Elementarteilerberechnung eingeben ließ. Das ist ein Problem der hardwarenahen FORTRAN–Programmierung, die eine Eingabe von langen Zahlen nicht zuläßt.

Die folgende Tabelle gibt den größte Matrixeintrag und die Matrixnorm nach einer bestimmten Isolierung an. Die Zeile **max** gibt das Maximum des größten Matrixeintrages und der Matrixnorm der Zwischenmatrizen aller Schritte an. Die Zeile **0** gibt die entsprechenden Größen vor der Bearbeitung an.

	Best Remainder		Preview	
Isolierung	maximaler Eintrag	Matrixnorm	maximaler Eintrag	Matrixnorm
0	124	395T	124	395T
20	151	622T	113	487T
30	1032	2800T	2518	1069T
35	40322	104Mio	30596	3250T
40	751Mio	45Bio	29300	3570T
max	1200Mio	61Bio	115574	4140T
Zeit		1123s		7565s

In diesem Beispiel mildert der Preview–Algorithmus die Koeffizientenexplosion stärker als das Best–Remainder–Verfahren ab; auf Kosten von siebenfacher Rechenzeit.

5.1.8 Fibonacci(2,9), Untergruppe Index 152, 148*31-Matrix

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> c := AbstractGenerator("c");;
gap> d := AbstractGenerator("d");;
gap> e := AbstractGenerator("e");;
gap> f := AbstractGenerator("f");;
gap> g := AbstractGenerator("g");;
gap> h := AbstractGenerator("h");;
gap> i := AbstractGenerator("i");;
gap> gens := [ a, b ,c ,d ,e ,f ,g ,h ,i];;
gap> G := Group (gens, IdWord);;
gap> G.relators := [a*b*c^-1,b*c*d^-1,c*d*e^-1,d*e*f^-1,e*f*g^-1,f*g*h^-1,
>   g*h*i^-1,h*i*a^-1,i*a*b^-1];;
gap> elems:=[i^76, i^-30*h^-2, h^-1*i*h*i^37, i^-2*h*i^2*h^-1];;
gap> H := Subgroup ( G, elems );;
```

```
gap> f152:= RelatorMatrixAbelianizedNormalClosureRrs( G, H );;
gap> ElementaryDivisorsIMat(f152);;
#I    Rank guessing...G...fullrank
#I  Rank:31
#I  Considering 20 submatrices.
#I    Determinants calculating...GGGG...done
#I  Rankminor gcd:3814697265625
#I    Factors:5^18
#I    Taking factor...5^6 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 5, 5, 5,
  5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
  5, 5, 5, 5, 5, 5 ]
```

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	14.2s	
	sicher modular	18.1s	
	Havas modular		Zu große Determinante
nichtmodular	Preview	143.2s	
	Best Remainder	21.5s	
	GAP 3.4	15.1s	
	Havas nichtmodular	7.8s	HP station (1/2)
	Havas neu shortint	0.2s	DEC Alpha (4)
	Preview mit Trafo	203.6s	
	Best Remainder mit Trafo	28.8s	

5.1.9 Tippfehler–Gruppe, Untergruppe Index 152, 71*71-Matrix

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> c := AbstractGenerator("c");;
gap> d := AbstractGenerator("d");;
gap> e := AbstractGenerator("e");;
gap> f := AbstractGenerator("f");;
gap> g := AbstractGenerator("g");;
gap> h := AbstractGenerator("h");;
gap> i := AbstractGenerator("i");;
gap> gens := [ a, b ,c ,d ,e ,f ,g ,h ,i];;
gap> G := Group (gens, IdWord);;
gap> G.relators := [a*b*c^-1,c*d*e^-1,d*e*f^-1,e*f*g^-1,f*g*h^-1,
>   g*h*i^-1,h*i*a^-1,i*a*b^-1];;
gap> elems:=[i^76, i^-30*h^-2, h^-1*i*h*i^37, i^-2*h*i^2*h^-1];;
gap> H := Subgroup ( G, elems );;
gap> t152:= RelatorMatrixAbelianizedNormalClosureRrs( G, H );;
gap> ElementaryDivisorsIMat(t152);;
#I    Rank guessing......done
#I  Rank:69
#I  Considering 2 submatrices.
#I    Determinants calculating...GGGGGGGG...done
#I  Rankminor gcd multiple:260770368415253147125244140625
#I    Factors:5^18*571^1*35531^1*46133^1*73037^1
#I    Taking factor...5^6 571^2 35531^2 46133^2 73037^2 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
```

```
  5, 341796937289240605 ]
```

Dieses Beispiel entstand aus einem Tippfehler bei der Eingabe der Fibonacci–Gruppe. Trotz des Tippfehlers ist die Gruppe interessant und es entstehen Matrizen, die eine heftige Koeffizientenexplosion erzeugen.

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	220s	
	sicher modular	222s	
	Havas modular		Zu große Determinante.
nichtmodular	Preview	432s	
	Best Remainder	81s	
	GAP 3.4	∞	
	Havas nichtmodular		gestoppt
	Havas neu shortint		overflow
	Havas neu	3.6s	Sparc II (1)

Im nichtmodularen Havas–Verfahren wurde mit der Meldung abgebrochen:
`stop diagonalisation because overflow very likely.`

5.2 Zufallsmatrizen

Ich möchte die bisher betrachteten Beispiele um einige ergänzen, die zwar keine gruppentheoretische Relevanz haben, dafür aber in gewisser Hinsicht sich *extremer* als die bisher betrachteten Beispiele verhalten.

5.2.1 Zuffallsmatrix 1, 25*25–Matrix

```
gap> ElementaryDivisorsIMat(m1,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 1, 1, 1, 2, 2, 6, 6, 12, 12, 96, 96, 7968 ]
```

Diese Matrix entstand aus der Multiplikation einer Diagonalmatrix mit zufälligen invertierbaren Matrizen (mit `RandomMat` hergestellt).

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	4.7s	
	sicher modular	7.4s	
	Havas modular		Eingabeformat
nichtmodular	Preview	57s	
	Best Remainder	6.8s	
	GAP 3.4	146s	
	Havas nichtmodular		Eingabeformat
	Havas neu shortint	0.06s	DEC Alpha (4)

Beim Havas–Programm ließ sich die Matrix wegen ihrer großen Koeffizienten nicht eingeben, da das Eingabeformat begrenzt ist.

5.2.2 Zuffallsmatrix 2, 10*10–Matrix

```
gap> ElementaryDivisorsIMat(m2,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 4, 8, 16, 16, 1744 ]
```

Diese Matrix entstand aus der Multiplikation einer Diagonalmatrix mit zufälligen invertierbaren Matrizen (mit `RandomMat` hergestellt).

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	0.34s	
	sicher modular	0.35s	
	Havas modular	0.46s	HP station (1/2)
nichtmodular	Preview	5.3s	
	Best Remainder	0.27s	
	GAP 3.4	0.13s	
	Havas nichtmodular	1.1s	HP station (1/2)
	Havas neu shortint	0.03s	DEC Alpha (4)

5.2.3 Zuffallsmatrix 3, 40*70–Matrix

```
gap> ElementaryDivisorsIMat(m3,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Eine zufällige Matrix mit kleinen Koeffizienten (mit `RandomMat` hergestellt).

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	27.8s	
	sicher modular	28.9s	
	Havas modular	13.9s	HP station (1/2)
nichtmodular	Preview	156s	
	Best Remainder	59s	
	GAP 3.4	∞	
	Havas nichtmodular		gestoppt
	Havas neu shortint	0.26s	DEC Alpha (4)

5.2.4 Zuffallsmatrix 4, 40*80–Matrix

```
gap> ElementaryDivisorsIMat(m4,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Eine sehr dünn bessetzte Zufallsmatrix mit kleinen Koeffizienten (aus einer mit `RandomMat` hergestellten Matrix viele Koeffizienten durch 0 ersetzt).

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	5.5s	
	sicher modular	5.8s	
	Havas modular	5.2s	HP station (1/2)
nichtmodular	Preview	25.9s	
	Best Remainder	12.8s	
	GAP 3.4	10.8s	
	Havas nichtmodular	2.2s	HP station (1/2)
	Havas neu shortint	0.11s	DEC Alpha (4)

5.2.5 Zuffallsmatrix 5, 50*50–Matrix

```
gap> ElementaryDivisorsIMat(m5,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 6, 72,
  816323186515605299618511418791520004651935766159645902151569283 52 ]
```

Meist waren die mit `RandomMat` erzeugten Matrizen invertierbar; diese Matrix bildete eine Ausnahme.

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	146s	
	sicher modular	144s	
	Havas modular		zu wenig Primzahlen
nichtmodular	Preview	262s	
	Best Remainder	123s	
	GAP 3.4	∞	
	Havas nichtmodular	∞	
	Havas neu shortint		overflow
	Havas neu	111s	Sparc II (1)

Beim modularen Havas–Programm wurde mit der Meldung abgebrochen:
`insufficient number of primes provided`.

5.2.6 Zuffallsmatrix 6, 120*100–Matrix

```
gap> ElementaryDivisorsIMat(m6,rec(fastdet:=true,fastrank:=true));
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Mit `RandomMat` erzeugte Matrix.

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular	713s	
	sicher modular	713s	
	Havas modular	439s	HP station (1/2)
nichtmodular	Preview	3113s	
	Best Remainder	2025s	
	GAP 3.4		Kommt zu keinem Ergebnis.
	Havas nichtmodular		gestoppt
	Havas neu shortint		overflow

5.2.7 Zuffallsmatrix 7, 61*61–Matrix

```
gap> ElementaryDivisorsIMat(m8,rec(algorithm:="bestrem"));
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  568592417783337956111230945608512735194469607423422367145223559093575448227 9\
8372468286749084396685819248118764271361741811576957225994506536971137977264 31\
9980774186844539145437128236610578440126347911624689346303021723861222882729 29\
6429698862605202428143505970521190056730853162045497518104025648124293084268 65\
6159054114107154873566450588174919224323928521523718114236659652026808061355 32\
1285579706127668002417556709106462728071907290363782017729922472343585159034 86\
07767314116745642377216 0 ]
```

Eine Matrix die aus einem Bug in einer alten GAP–Version entstanden ist. Sie hat eine Determinante, die nur wenige Größenordnungen unter der HADAMARD–Schranke liegt.

Zeitvergleiche			
Verfahrenstyp	Programm	Rechenzeit	Bemerkung
modular	schnell modular		Determinante zu groß
	sicher modular		Determinante zu groß
	Havas modular		zu wenig Primzahlen
nichtmodular	Preview	1782s	
	Best Remainder	1571s	
	GAP 3.4		Kommt zu keinem Ergebnis.
	Havas nichtmodular		gestoppt
	Havas neu shortint		overflow

Bei den modularen Verfahren war der Rangminoren–ggT so groß, daß er nicht faktorisiert werden konnte.

5.3 Koeffizientenvergleiche

Im folgenden vergleiche ich die beiden von mir in GAP implementierten nichtmodularen Verfahren anhand der maximalen Koeffizienten der nach einer bestimmten Isolierung auftretenden Zwischenmatrizen.

Auf der einen Seite steht das Best–Remainder–Verfahren, auf der anderen Seite das kombinierte Verfahren, das so lange dem Preview–Algorithmus folgt, bis der Rechenaufwand für dieses Verfahren zu hoch wird, und danach mit dem Best–Remainder–Algorithmus weiterrechnet. Genaures zu der kombinierten Strategie ist im Kapitel **Bedienungsanleitung** zu finden.

In der Spalte **Umschalten bei** gebe ich an, nach der wievielten Isolierung das kombinierte Verfahren den Preview–Algorithmus abbricht. Danach gebe ich die maximalen Koeffizienten der Zwischenmatrizen an, die vor dieser Isolierung aktuell waren. Bei einigen Beispielen wurde nicht umgeschaltet, dort steht ein - .

	am Beginn	insgesamt		beim Abbruch des kombinierten Verfahrens		
Matrix		Best Rem.	kombiniertes	Umschalten bei	Best Rem.	Preview
cam3	20	38	24	-		
g33	7	78	17	-		
h60	193	$8.4*10^{7}$	$3.7*10^{12}$	11(von 21)	14400	744
h120	634	$1.8*10^{11}$	$1.0*10^{10}$	9(von 20)	4439	468
fourn	124	$1.2*10^{9}$	$1.2*10^{5}$	-		
m1	$8*10^{7}$	$5.7*10^{11}$	$8.7*10^{15}$	2(von 14)	$2.1*10^{9}$	$8.7*10^{6}$
m2	2548	1818016	25248	-		
m3	8	$8*10^{22}$	$4.5*10^{26}$	9(von 40)	4358	789
m4	216	232	216	-		
f152	15	60	60	-		

Bezüglich der Koeffizientengröße kann folgendes observiert werden:

- Bei praktisch allen Schritten in allen Beispielen treten im Preview–Algorithmus kleinere Koeffizienten als im Best–Remainder–Verfahren auf.
- Beim Umschalten des kombinierten Verfahrens kann es trotz besseren Zwischenergebnisses des Preview–Verfahren vorkommen, daß am Ende ein schlechteres Ergebnis erzielt wird. Man kann also vermuten, daß im Preview–Verfahren die Koeffizientenexplosion aufgeschoben werden kann.

5.4 Killermatrizen

Es ist wirklich sehr selten, daß das schnelle modulare Verfahren zu falschen Ergebnissen kommt. Ich zeige hier kurz, wie mein Programm auf speziell dafür konstruierte Beispiele reagiert. Sie basieren auf der Kenntnis der Primzahlen, die als Moduln präferiert werden.

5.4.1 unsichere Rangberechnung

```
gap> ElementaryDivisorsIMat(k3);
```

```
#I    Rank guessing......done
#I  Rank:18
#I  Considering 7 submatrices.
#I    Determinants calculating...GGGG...done
#I  Rankminor gcd multiple:1
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1 ]
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
gap> ElementaryDivisorsIMat(k3,rec(fastrank:=false,fastdet:=false));
#I    Rank guessing......done
#I    Hadamarding...done.
#I  Improved Hadamard bound:10^52
#I  Maximal 13 primes considered.
#I    Rank confirming...GGGGGGGGGGGGG...done
#I    Rank guessing...GGG...done
#I    Hadamarding...done.
#I  Improved Hadamard bound:10^55
#I  Maximal 14 primes considered.
#I    Rank confirming...GGGGGGGGGGGGGG...done
#I  Rank:21
#I  Considering 4 submatrices.
#I    Hadamarding ...done.
#I  New Hadamard bound:10^42
#I  Considering 0 primes.
#I    Determinants calculating......done
#I  Rankminor gcd multiple:27078981135318280780783274
#I    Factors:2^1*15427^1*15439^3*15443^2
#I    Taking factor...2^2 15427^2 15439^2 15443^2 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 15439, 238424477, 3678174406679 ]
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 15439, 238424477,
  3678174406679 ]
gap> ElementaryDivisorsIMat(k3,rec(maxfivers:=2));
#I    Rank guessing......done
#I  Rank:18
#I  Considering 2 submatrices.
#I    Determinants calculating...GGGG...done
#I  Rankminor gcd multiple:2
#I    Factors:2^1
#I    Taking factor...2^2 100%
Fatal: Rang was wrong, start all over
#I    Rank guessing...G...done
#I  Rank:19
#I  Considering 2 submatrices.
#I    Determinants calculating...GGGG...done
#I  Rankminor gcd multiple:15439
#I    Factors:15439^1
#I    Taking factor...15439^2 100%
Fatal: Rang was wrong, start all over
#I    Rank guessing...GG...done
#I  Rank:20
#I  Considering 2 submatrices.
#I    Determinants calculating...GGGGG...done
#I  Rankminor gcd multiple:29448284003224
#I    Factors:2^3*15439^2*15443^1
#I    Taking factor...2^4 100%
```

```
Fatal: Rang was wrong, start all over
#I    Rank guessing...GGG...done
#I  Rank:21
#I  Considering 2 submatrices.
#I    Determinants calculating...GGGGGGG...done
#I  Rankminor gcd multiple:27078981135318280780783274
#I    Factors:2^1*15427^1*15439^3*15443^2
#I    Taking factor...2^2 15427^2 15439^2 15443^2 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 15439, 238424477, 3678174406679 ]
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 15439, 238424477,
  3678174406679 ]
```

Bei fehlerhafter Rangberechnung ist es in vielen Fällen möglich, daß der Fehler erkannt und korrigiert wird. Dieses Beispiel vereint einen günstigen und ungünstigen Fall – durch die Änderung eines an sich unbeteiligten Parameters.

5.4.2 unsichere Determinantenberechnung

```
gap> ElementaryDivisorsIMat(k5);
#I    Rank guessing......fullrank
#I  Rank:11
#I  Considering 1 submatrices.
#I    Determinants calculating...G...done
#I  Rankminor gcd multiple:440
#I    Factors:2^3*5^1*11^1
#I    Taking factor...2^4 5^2 11^2 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2 ]
[ 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2 ]
gap> ElementaryDivisorsIMat(k5,rec(fastrank:=false,fastdet:=false));
#I    Rank guessing......fullrank
#I  Rank:11
#I  Considering 1 submatrices.
#I    Hadamarding ...done.
#I  New Hadamard bound:10^22
#I  Considering 5 primes.
#I    Determinants calculating...GGGGG...done
#I  Rankminor gcd multiple:1907396256
#I    Factors:2^5*3^1*23^4*71^1
#I    Taking factor...2^6 3^2 23^3 71^2 ...done
#I  Elementary Divisors:[ 1, 1, 1, 1, 1, 1, 2, 46, 46, 46, 9798 ]
[ 1, 1, 1, 1, 1, 1, 2, 46, 46, 46, 9798 ]
```

In diesem Beispiel scheint das erste, falsche Ergebnis besonders plausibel. Außerdem sind hier nicht so auffällig große Primteiler der Elementarteiler zu finden, was einmal mehr unterstreicht, daß die unsichere Rangberechnung viel sicherer ist als die unsichere Determinantenberechnung. Nichtsdestoweniger sind aber auch solche Killermatrizen außerordentlich selten.

Kapitel 6

Implementierung in GAP

In vorhergehenden Kapiteln habe ich die Algorithmen im Detail beschrieben. Ich will hier darstellen, von welchen grundsätzlichen Prinzipen ich bei der Implementierung ausgegangen bin.

Ich beginne mit der Einleitung und Lesehilfe zum Studieren des Quelltextes. Denn mein Ziel war es unter anderem auch, einen les– und änderbaren Programmtext zu schreiben; vor allem, um *mir* das Finden der Fehler und das Hineinimplementieren neuer Ideen zu erleichtern, wobei ich von der Devise ausgehe: **Wenn man einen Bug mehr als einen Tag erfolglos gesucht hat, sollte man das Programm neu schreiben.** So verwarf ich eine erste Version dieses Programms; und für die zweite Version entwarf ich vor allem die Konzepte der IMR–Datenstruktur und eines ausführlichen Info–Systems.

6.1 Was habe ich implementiert?

6.1.1 modulares Verfahren

Ich habe das modulare Verfahren in der angegebenen Form implementiert, dabei die schon erwähnten Verbesserungen eingebracht. Für die Rang– und Determinantenberechnung habe ich sowohl das sichere als auch des schnelle Verfahren implementiert. Optional kann man für die modulare Elementarteilerberechnung die reduzierte Primzahlpotenz wählen lassen. Die Begrenzung für die Anzahl der Rangminoren kann vom Nutzer beliebig eingestellt werden. Man kann zwischen der verbesserten oder der einfachen HADAMARD–Schranken–Berechnung wählen. Alle diese Alternativen sind mit einem einheitlichen Muster schaltbar.

6.1.2 nichtmodulare Verfahren

Ich habe einen Preview–Algorithmus implementiert. Dabei betrachte ich einfach isolierbare Pivots der Typen 1 bis 3. Die angesprochene Verbesserung des Isolierens habe ich für die Typen 1 bis 3 (Typen der Eliminierung) realisiert. Die 3 Möglichkeiten der begrenzten Pivotsuche wurden realisiert. Übergroße Schaden werden sowohl vor dem Isolieren (mit dem vorhergesehen Schaden), als auch danach (mit einem Backtracking–Mechanismus) verhindert. Ergänzt habe ich mit einer GAUSS–Reduktionsroutine, die in Abhängigkeit von der Matrixgröße an Durchläufen begrenzt wurde. Sie wird eingesetzt, nachdem ein übergroßer vorhergesehener Schaden verhindert wurde und nach einem erfolgten Backtracking.

Ich implementierte die Best–Remainder–Strategy; sie wird im Preview–Programm benutzt, nachdem das vierte Mal Backtracking ausgeführt wurde (was als Zeichen dafür angesehen wird, daß der reine Preview–Algorithmus nun eine nicht mehr vertretbare Zeit zur Suche von Pivots aufwendet. Die Implementation der Best–Remainder–Strategy kann aber auch von Beginn an genutzt werden.

6.2 Das Zusammenspiel der Prozeduren

Zunächst möchte ich einen Überblick über die Prozeduren des Programms geben; als Wörterbuch für die Übersetzung des Programms in die Formulierung der Algorithmen.

6.2.1 im modularen Verfahren

ElDivModIMR Das ist die Hauptprozedur des modularen Verfahrens. Sie ruft zunächst `DetIMR` auf, die ein Vielfaches des Rangminoren–ggT zurückgibt. Dieses wird faktorisiert und mit den entsprechend erhöhten Primzahlpotenzen `ModElDivIMR` aufgerufen. Am Ende werden die modularen Elementarteiler elementweise multipliziert, um die ganzzahligen Elementarteiler zu erhalten.

ModElDivIMR Elementarteiler modulo einer Primzahlpotenz.

DetIMR berechnet den ggT von Rangminoren. Zunächst wird `FiveSelectIMR` gerufen, das die Grundlage für den wiederholten Aufruf von `ModFiveStepIMR` liefert. Dessen Ergebnisse werden mit dem `ListChineseRem` kombiniert.

ModFiveStepIMR [1] Determinantenberechnung modulo einer Primzahl für einen Satz von Rangteilmatrizen.

FiveSelectIMR Wählt einen Satz von Rangteilmatrizen. Ruft `RankIMR` auf, um den Rang, vor allem aber verschieden Hypothesen für mögliche Rangteilmatrizensätze. Aus diesen wählt sie die beste aus.

RankIMR Universelle Prozedur, die nicht nur den Rang, sondern auch Teilmatrizensätze von diesem Rang und deren Determinanten modulo verschiedener Primzahlen liefert, die man als Hypothesen für den letztendlichen Rangminorensatz ansieht.

ModRankIMR Rang und Determinantenbestimmung modulo einer Primzahl bei gleichzeitiger Ausgabe des Teilmatrizensatzes.

HadamardIMR berechnet eine Hadamard–Schranke.

FiveHadamardIMR Maximum von Hadamard–Schranken von einem Satz Teilmatrizen.

BiggerPrimes liefert eine Menge von Primzahlen, deren Produkt gerade größer ist, als die übergebene Zahl.

MultTrafo,SubTrafo sind die elementaren modularen Matrixoperationen.

6.2.2 in den nichtmodularen Verfahren

ElDivIMR Das ist die Hauptprozedur. Sie ruft zunächst `DiagIMR` auf, die die Diagonalelemente einer äquivalenten Diagonalmatrix zurückgibt, die die Teilbarkeitsbedingung noch nicht erfüllen. Diese wird dann im folgenden hergestellt.

DiagIMR Hauptschleife des Programms. Hier werden nacheinander die Prozeduren aufgerufen, die Normen und ggTs berechnen, den Pivot auswählen, ihn isolieren und dann seine Zeile und Spalte aus der matrix entfernen. Nebenbei wird die Teilbarkeit aller Matrixelemente geprüft. In den sogenannten Notfällen (kein akzeptabler Pivot gefunden, Schaden war zu hoch) werden die entsprechenden Schritte eingeleitet (Mischen und neuen Pivot suchen, Backtracking). Hier wird auch das regelmäßige Sichern von Zwischenmatrizen aufgerufen.

ActualiseNorms Hier werden alle Normen und ggTs berechnet, gleichzeitig wird der wirkliche Schaden ermittelt und ggf. Backtracking eingeleitet.

ResActualiseNorms eingeschränkte Version, nur für `BRem`.

DivisorReductionIMR Die von `ActualiseNorm` berechneten ggTs werden benutzt, um zu ermitteln, ob die gesamte Matrix eionen Teile besitzt. Evtl. werden alle Einträge durch ihn geteilt und Normen sowie ggTs aktualisiert.

---

[1] Der Name FiveStep rührt von einer Implementation dieses Algorithmus von George Havas her, wo der Satz von Rangminoren genau aus fünf Rangminoren bestand — und in Ermangelung eines besseren Namens wurde aus *fünf Determinanten gleichzeitig in einem Schritt* eben FiveStep. Weil er lustig klingt, behielt ich ihn bis zum Schluß bei, obwohl nun die Anzahl der Minoren beliebig steuerbar ist.

TemporaryStoreIMR Die aktuelle Matrix `amat` wird in `oldmat` gesichert, um Backtracking zu ermöglichen. Gleichzeitig werden einige andere notwendige Daten gesichert.

BackTrackIMR realisiert das Backtracking.

PivotIMR Gibt ein akzeptiertes Pivotelement mit dem vorhergesehenen Abstieg.

ValidDescentIMR Gibt alle Abstiege für eine bestimmte Pivotposition und eine Zuordnung.

ZeroDescentsIMR Gibt den Abstieg vom Typ 1 bezüglich einer Pivotposition und einer Zuordnung.

DescentsIMR Berechnet alle Abstiege der Typen 2 und 3.

DamageIMR Berechnet den Schaden eines Abstiegs.

Alpha Die normtypische Summenerwartung.

ParamUpdate Die Parameter werden in Abhängigkeit vom Verlauf geändert.

IsolateIMR Das Isolieren mithilfe eine vorhergesehenen Abstiegs.

TryIsolateIMR Ein Schritt nach der Best Remainder Strategy.

BRemIMR Best Remainder Algorithm, falls man den Preview Algorithm aufgegeben hat.

GcdTrafoIMR, SubTrafoIMR, SwapTrafoIMR, MultTrafoIMR sind die elementaren Matrixoperationen.

6.3 Die IMR–Datenstruktur

In GAP ist eine Matrix eine Liste von gleichlangen Listen von Elementen eines Typs, oder anders formuliert eine Liste von Vektoren gleicher Länge. Eine ganzahlige Matrix ist eine solche Matrix, deren sämtliche Elemente ganzzahlig sind. Ich bette diese Matrix in eine Record–Datenstruktur ein die ich *IMR* (wie integer matrix record) nenne, im folgenden ist von einem Objekt dieses Typs als IMR die Rede.

6.3.1 Warum diese Datenstruktur?

Matrix als logische Einheit Mein Programm führt eine ganzzahlige Matrix auf verschiedenartigen Zwischenstadien zu ihrer SMITH–Normalform. Darum macht es Sinn, diese Zwischenstadien an der logischen Einheit Matrix selbst zuzuordnen. Ich fasse die Matrix damit als **Objekt**.

Zwischenergebnisse Es können Zwischenergebnisse in diese Datenstruktur ergänzt werden, was die Kommunikation zwischen den einzelnen Prozeduren erleichtert und außerdem eine interaktive Arbeitsweise ermöglicht. Wenn die Berechnung unterbrochen wird, kann man bereits errechnete Ergebnisse wiederverwenden.

Parametrisierung Es gibt mehrere Möglichkeiten und unzählige Varianten, Elementarteiler zu berechnen; etliche sind in meinem Programm vorhanden. Doch wie soll man sie steuern? Von den verschiedenen Möglichkeiten, Parameter mitzuteilen, gefallen mir die am besten, die die Parameter der *Matrix* logisch zuordnen.

Anschaubarkeit Schon mal versehentlich eine 1000*1000–Matrix anzeigen lassen? Ein IMR wird so angezeigt:

```
gap> IMRIMat([[2,1],[0,2]]);
IntegerMatrix(
  matrix  := ... ,
  transposed  := false,
  rows  := 2,
  cols  := 2,
  minrank  := 0,
```

```
maxrank  := 2,
ElementaryDivisorsOptions  := rec(
algorithm := "modular",
maxfivers := 20,
safeexp := false,
tolerate := 1/50,
tolbot := 0,
toltop := 1/3,
tolup := 1/25,
toldown := 1/100,
intolerate := 1/2,
intolbot := 2/5,
intoltop := 4,
intolup := 1/10,
intoldown := 1/50,
avoidrate := 5/2,
pivotbound := 30,
pivbot := 24,
pivtop := 46,
pivup := 3,
pivdown := 1,
descentbound := 90,
desbot := 74,
destop := 100,
desup := 6,
desdown := 2,
stepbound := 4,
stepbot := 2,
steptop := 6,
stepup := 1,
stepdown := 1,
shuffles := [ 1, 2 ],
redurate := 1,
isom := false ),
RankOptions  := rec(
fastrank := false,
fasthadamard := false ),
DeterminantOptions  := rec(
fastdet := false,
fasthadamard := false )  )
```

6.3.2 Was ist ein IMR?

Ein Record, der

- eine ganzzahlige Matrix im gewöhnlichen GAP–Format,
- ein paar Informationen zur Matrix wie die Zeilen/Spaltenzahl,
- viele Steuerparameter, die angeben, wie sich die Matrix zu bearbeiten wünscht und
- Zwischenergebnisse, wie weit die Matrix schon auf dem Weg zur SMITH–Normalform ist,

enthält.

6.3.3 Wie bekommt man einen IMR?

Die Erzeugung eines IMRs geschieht mit einem Aufruf der Prozedur `IMRIMat`, die eine ganzahlige Matrix in einen IMR verwandelt, das erste Funktionsargument. Das optionale zweite Funktionsargument ist ein sogenannter *Options–Record*, der Parameter des Algorithmus setzt. Fehlt er, werden Default–Werte benutzt; genauso, wenn bestimmte Felder nicht in dem Options–Record mit angegeben sind, die zur Berechnung benötigt werden. Jeder IMR enthält drei solcher Options–Records, nämlich:
`ElementaryDivisorsOptions`, `RankOptions` und `DeterminantOptions` — die Felder des eventuellen zweiten Parameters werden auf diese aufgeteilt.

Wenn `ElementaryDivisorsIMat`, `RankIMat` oder `DeterminantIMat` mit einer Matrix aufgerufen wird, dann wird der IMR implizit erzeugt.

6.3.4 Die Stärken des IMR-Konzepts

Anhalten und Weiterrechnen

Zu jedem Zeitpunkt soll es möglich sein, die Berechnung abzubrechen, Parameter zu ändern, auf einem anderen Rechner mit den selben Daten weiterarbeiten. Man kann eine halb berechnete Matrix abspeichern und später wieder laden.

Das geht über die Möglichkeit hinaus, mit `^C` zu unterbrechen und nach einigen Abfragen oder Änderungen mit `return;` fortzufahren. Man kann auch mit `^D` aus dem `break loop` herausgehen und mit einem erneuten Aufruf von z.B. `ElementaryDivisorsIMat` von vorn beginnen, ohne die meisten Zwischenergebnisse zu verlieren.

Das wird durch die Speicherung vieler Zwischenwerte im IMR ermöglicht. Dazu darf natürlich die Datenstruktur auch innerhalb der Prozeduren nicht inkonsistent werden. Das wird im wesentlichen durch die Zwischenspeicherung der Ergebnisse in lokalen Variablen erreicht, so daß nur fertige Ergebnisse in den IMR–Record eingehängt werden.

Das Sichern eines IMRs in ein File geschieht mit einem Aufruf von
`SaveIMRTo("filename","name",<imat>);`, das Wiedereinlesen erfolgt mit dem Einlesen des Files. Der IMR ist dann wieder unter dem Namen `"name"` abrufbar.

Probieren

Manchmal möchte man von der normalen Abfolge der Schritte abweichen. Man ruft dann einige Prozeduren auf, die gewisse Schritte nicht benötigen, die normalerweise am Anfang geschehen.

Z. B. könnte man sich zuerst für die 2^{14}–modularen Elementarteiler interessieren, bevor man den Rang und den Rangminoren–ggT ausrechnet.

Die Ergebnisse dieser Testaufrufe werden ebenfalls im IMR gespeichert. Es muß aber an allen Stellen gewährleistet sein, daß irrelevante Ergebnisse gelöscht oder ignoriert werden, nicht aber den Lauf des Programms behindern. Es soll nicht möglich sein, durch probeweise Aufrufe der Prozeduren falsche Ergebnisse hervorzubringen.

Manipulationen

Die Datenstruktur IMR macht es einfach, während des Laufs des Programms zu manipulieren. Dafür kann es viele vernünftige Gründe geben: Man weiß den Rang einer Matrix, und möchte das Programm den Rang nicht noch einmal bestimmen lassen. Oder man kennt den größten Elementarteiler schon vor Beginn, und möchte nur noch die anderen Elementarteiler berechnen. Man sieht, daß das Vielfache des Rangminoren–ggT viel zu groß geraten ist und vielleicht gar nicht faktorisiert werden kann, zumindest aber viele modulare Elementarteiler mit großen Moduln berechnet werden müssen. Man weiß aber z. B. aus mathematischen Überlegungen oder einfach nach Gefühl, daß die Elementarteiler klein sind.

Im Preview–Verfahren sind die Möglichkeiten der Manipulation ungleich vielfältiger. Zunächst ist es sinnvoll, die Parameter zur Begrenzung der Pivotsuche (oder die Parameter zu deren automatischer Änderung) zu ändern, wenn die automatische Anpassung dieser Parameter nicht ausreicht. Man kann ändern, auf wie viele Durchläufe die Reduktion begrenzt werden soll. Wie oft Zeilen– und Spalten–reduziert werden soll, bevor im Preview–Verfahren ein Best–Remainder–Schritt ausgführt wird. Nach wie vielen Backtrackings die Preview–Strategie aufgegeben werden soll. Ferner kann man sogar zwischen Preview und Best–Remainder hin– und herschalten.

Das Manipulieren erfolgt durch Zuweisung der Felder des IMRs.

6.4 Programmlisting

```
#############################################################################
##
#H  imat.g                 GAP program                Andreas Hoppe
##
##
##  This file contains implementations to compute the elementary divisors,
##    the rank, and the determinant of an integer matrix.
##  Furthermore it contains the definition of IMR, a data structure
##    especially designed for the elementary divisors computation.
##    the matrix and parameters used for the various elementary divisors
##    algorithms.
##
#############################################################################
##
#H  IMR data structure definition and utilities
##
#############################################################################
#F  InfoIMat1, InfoIMat2, InfoIMat3, InfoIMat4 . . . .  Intermediate results
if not IsBound(InfoIMat1)  then InfoIMat1 := Ignore;  fi;
# Procedure Calls and Exits
if not IsBound(InfoIMat2)  then InfoIMat2 := Ignore;  fi;
# Intermediate results Information
if not IsBound(InfoIMat3)  then InfoIMat3 := Ignore;  fi;
# The Moving Image Information
if not IsBound(InfoIMat4)  then InfoIMat4 := Ignore;  fi;
# Very Special Information

#############################################################################
#F  InfoIMat(true),InfoIMat(false) switches printing of information results
InfoIMat:=function(type)
  if type then
    InfoIMat3:=Print;
    InfoIMat2:=Print;
  else
    InfoIMat3:=Ignore;
    InfoIMat2:=Ignore;
  fi;
end;

#############################################################################
#F  PrintIMR( <imat> ) . . Print a IMR nicely
##  This procedure goes to the operations field of each IMR, so each
##    time an IMR is printed this procedure is used instead of printing
##    all fields. This avoids printing the large working data structures.
PrintIMR:=function(imat)
  local p;
  Print("IntegerMatrix(\n  matrix  := ... ");
  for p in RecFields(imat) do
    if p<>"operations" and p<>"matrix" then
      if p="amat" or p="rownorm" or
         p="colnorm" or p="oldmat" or p="left" or p="right" or
         p="oldleft" or p="oldright" then
        Print(",\n  ",p,"  := ... ");
      else
        Print(",\n  ",p,"  := ",imat.(p));
      fi;
    fi;
```

```
  od;
  Print("  )\n");
end;

#############################################################################
#V  ElementaryDivisorsDefaultOptions . . . record used in IMRIMat
ElementaryDivisorsDefaultOptions := rec(


    # type of algorithm, one of "modular", "preview", and  "bestrem"
    algorithm     := "modular",

    # parameters  for the modular method
    maxfivers     := 20,             # gcd of those many subdets
    safeexp       := false,          # ModElDiv only with big enough exp.

    # parameters for preview method
    tolerate      := 1/50,           # damage smaller than this proportion
    tolbot        := 0,              #   of matrixnorm is good enough to take
    toltop        := 1/3,
    tolup         := 1/25,
    toldown       := 1/100,
    intolerate    := 1/2,            # damage bigger than this proportion
    intolbot      := 2/5,            #   of matrixnorm is too bad to take though best
    intoltop      := 4,
    intolup       := 1/10,
    intoldown     := 1/50,
    avoidrate     := 5/2,            # matrixnorm increase bigger than this
                                     #   proportion causes backtracking
    pivotbound    := 30,             # so many non-zero matrix positions are
    pivbot        := 24,             #   considered as pivots
    pivtop        := 46,
    pivup         := 3,
    pivdown       := 1,
    descentbound  := 90,             # so many descents over all pivots are
    desbot        := 74,             #   considered
    destop        := 100,
    desup         := 6,
    desdown       := 2,
    stepbound     := 4,              # no descent is longer
    stepbot       := 2,
    steptop       := 6,
    stepup        := 1,
    stepdown      := 1,
    shuffles      := [1,2],          # order of the shuffle operations
    redurate      := 1,              # so many reduction cycles as proportion
                                     #   of rows/columns
    isom          := false           # calculate transformation matrices
  );
#V  RankDefaultOptions . . . record used in IMRIMat
RankDefaultOptions := rec(
    fastrank      := false,           # makes rank calculation fast and unsafe
    fasthadamard  := false           # gives a rougher bound while computing faster
  );
#V  DeterminantDefaultOptions . . . record used in IMRIMat
DeterminantDefaultOptions := rec(
    fastdet       := false,          # makes determinant calculation unsafe
    fasthadamard  := false           # gives a rougher bound while computing faster
  );

#############################################################################
```

```
#F  IMRIMat(<mat>[,optionrec] )  . . create an IMR
##  This is the basic function to create a integer matrix record. It
##    uses values from the above xxxDefaultOptions.
##  The fields of the optionrec are copied in the xxxOptions fields
##    depending in which of them it belongs. Fields which are components
##    of more than one xxxOptions field are copied in each of them (at
##    present only 'fasthadamard').
##  IMRs are displayed via PrintIMR.
##  Note that the field matrix can contain either a link to the original
##    matrix <mat> (in which case the changing of the original matrix can
##    cause serious inconsistencies) or to a unique copy.
IMRIMat := function( arg )
  local mat,imat,optchange,p;
  mat:=arg[1];
  if not IsMat(mat) then Error("matrix expected"); fi;

  imat:=rec();
  if Length(mat)<Length(mat[1]) then
    imat.matrix:=TransposedMat(mat);
    imat.transposed:=true;
  else
    imat.matrix:=mat;
    imat.transposed:=false;
  fi;
  imat.rows := Length(imat.matrix);
  imat.cols := Length(imat.matrix[1]);
  imat.operations    := rec(Print:=PrintIMR);
  imat.minrank       := 0;
  imat.maxrank       := imat.cols;

  imat.ElementaryDivisorsOptions:=rec();
  imat.RankOptions:=rec();
  imat.DeterminantOptions:=rec();
  if Length(arg)=2 then
    optchange:=arg[2];
    for p in RecFields(optchange) do
      if p="fastrank" then
        imat.RankOptions.(p):=optchange.(p);
      elif p="fastdet" then
        imat.DeterminantOptions.(p):=optchange.(p);
      elif p="fasthadamard" then
        imat.DeterminantOptions.(p):=optchange.(p);
        imat.RankOptions.(p):=optchange.(p);
      else
        imat.ElementaryDivisorsOptions.(p):=optchange.(p);
      fi;
    od;
  fi;
  for p in RecFields(ElementaryDivisorsDefaultOptions) do
    if not IsBound(imat.ElementaryDivisorsOptions.(p)) then
      imat.ElementaryDivisorsOptions.(p):=
        ElementaryDivisorsDefaultOptions.(p);
    fi;
  od;
  for p in RecFields(RankDefaultOptions) do
    if not IsBound(imat.RankOptions.(p)) then
      imat.RankOptions.(p):=RankDefaultOptions.(p);
    fi;
  od;
  for p in RecFields(DeterminantDefaultOptions) do
    if not IsBound(imat.DeterminantOptions.(p)) then
```

```
      imat.DeterminantOptions.(p):=DeterminantDefaultOptions.(p);
    fi;
  od;

  return imat;
end;


#############################################################################
#F  SaveIMRTo( "file","name",<imat> )  . save an IMR
##  Saved as gap-file in the variable "name". This is invaluable
##    when using more than one GAP-session (even on different computers)
##    for one calculation. Note that it saves the IMR-record as an
##    assignment to the variable "name".
##  Each IMR begins with an #V line with information about the matrix
##    size so that You can save more than one IMR to a file.
##  Be sure to have imat.g loaded when reloading files stored by this
##    procedure.
SaveIMRTo := function(file,name,imat)
  local p;
  AppendTo(file,"\n#V  ",imat.rows,"*",imat.cols,
    "-matrix imat record\nPrint(\"",name," loading\\n\");\n",
    name,":=rec(\n  matrix:= ",Copy(imat.matrix),
    ",\n  operations:=rec(Print:=PrintIMR)");
  for p in RecFields(imat) do
    if p<>"matrix" and p<>"operations" then
      AppendTo(file,",\n  ",p," := ",Copy(imat.(p)));
    fi;
  od;
  AppendTo(file," );\n\n");
end;

#############################################################################
##
#H  Simple subprocedures for the modular method
##
#############################################################################
#V  _primes . . . . . a list of 200 primes
##  They are numbers whose square is a short GAP integer.
##  Their product is about 4.5*10^837.
##  They are sorted that way that the product of the first n is just bigger
##   than the n-th power of 15425
##     (which is RootInt(Product(_primes),Length(_primes)) )
_primes:=
[ 15439, 15443, 15427, 15451, 15413, 15461, 15401, 15467, 15391, 15473,
  15383, 15493, 15377, 15497, 15373, 15511, 15361, 15527, 15359, 15541,
  15349, 15551, 15331, 15559, 15329, 15569, 15319, 15581, 15313, 15583,
  15307, 15601, 15299, 15607, 15289, 15619, 15287, 15629, 15277, 15641,
  15271, 15643, 15269, 15647, 15263, 15649, 15259, 15661, 15241, 15667,
  15233, 15671, 15227, 15679, 15217, 15683, 15199, 15727, 15193, 15731,
  15187, 15733, 15173, 15737, 15161, 15739, 15149, 15749, 15139, 15761,
  15137, 15767, 15131, 15773, 15121, 15787, 15107, 15791, 15101, 15797,
  15091, 15803, 15083, 15809, 15077, 15817, 15073, 15823, 15061, 15859,
  15053, 15877, 15031, 15881, 15017, 15887, 15013, 15889, 14983, 15901,
  14969, 15907, 14957, 15913, 14951, 15919, 14947, 15923, 14939, 15937,
  14929, 15959, 14923, 15971, 14897, 15973, 14891, 15991, 14887, 16001,
  14879, 16007, 14869, 16033, 14867, 16057, 14851, 16061, 14843, 16063,
  14831, 16067, 14827, 16069, 14821, 16073, 14813, 16087, 14797, 16091,
  14783, 16097, 14779, 16103, 14771, 16111, 14767, 16127, 14759, 16139,
  14753, 16141, 14747, 16183, 14741, 16187, 14737, 16189, 14731, 16193,
  14723, 16217, 14717, 16223, 14713, 16229, 14699, 16231, 14683, 16249,
```

```
  14669, 16253, 14657, 16267, 14653, 16273, 14639, 16301, 14633, 16319,
  14629, 16333, 14627, 16339, 14621, 16349, 14593, 16361, 14591, 16363,
  14563, 16369, 14561, 16381, 14557, 16411, 14551, 14549, 14543, 14537 ];
_primesprod:=Product(_primes);
#F  BiggerPrimes( <n> )  a number of primes whose product exeeds number n
BiggerPrimes:=function(n)
  local result,prime;

  n:=n*2;   # Note that we have to catch up negative values
  if n<_primesprod then
    result:=Sublist(_primes,[1..LogInt(n,15425)+1]);
  else
    result:=_primes;
    n:=n/_primesprod;
    prime:=16417;
    while n>1 do
      Add(result,prime);
      prime:=NextPrimeInt(prime);
      n:=n/prime;
    od;
  fi;
  return result;
end;
#############################################################################
#F  ElementProduct(vector1,vector2) . . . . elementwise product of vectors
ElementProduct:=function(vector1,vector2)
  local i,result;
  result:=[];
  for i in [1..Length(vector1)] do
    result[i]:=vector1[i]*vector2[i];
  od;
  return result;
end;
#############################################################################
#F  MultTrafo(<skalar>,<vector>,<col>,<modul>) .. modular matrix operation
##    multiply skalar to the vector in the ring of residues modulo modul
##  the vector is expected to be zero up to position col
##  returns nothing but changes vector directly
##  note that this is a invertible matrix operation if skalar is invertible
##    in the residues modulo modul and the determinant of the result is
##    multiplied by skalar
MultTrafo:=function (skalar,vector,col,modul)
  local i;
  for i in [col..Length(vector)] do
    vector[i]:=skalar*vector[i] mod modul;
  od;
end;
#############################################################################
#F  SubTrafo(<vector1>,<skalar>,<vector2>,<col>,<modul>) .. matrix operation
##    subtracts a multiple of vector2 from vector1
##  the vector is expected to be zero up to position col
##  returns nothing but changes vector1 directly
##  note that this is a invertible matrix operation and leaves the
##    determinant unchanged
SubTrafo:=function (vector1,skalar,vector2,col,modul)
  local i;
  for i in [col..Length(vector1)] do
    vector1[i]:=(vector1[i]-skalar*vector2[i]) mod modul;
  od;
end;
#############################################################################
```

```
#F  ListChineseRem(modullist,remslist) .. multiple chinese remainders
##  remslist is a matrix, its columns define different numbers, the
##    elements of each column are the remainders to the moduls in
##    modullist.
##  Returns the list of those numbers computed by multiple application of
##    the chinese Remainder procedure.
ListChineseRem:=function(modullist,remslist)
  local remlist,prod,chin,i,j;

  if modullist=[] then return []; fi;
  chin:=[];
  prod:=Product(modullist);
  remlist:=[];

  for i in [1..Length(remslist[1])] do
    for j in [1..Length(remslist)] do
      remlist[j]:=remslist[j][i];
    od;
    chin[i]:=ChineseRem(modullist,remlist);
    if 2*chin[i]>=prod then chin[i]:=chin[i]-prod; fi;
  od;
  return chin;
end;

#############################################################################
##
#H  Main procedures for the modular method
##
#############################################################################
#F  HadamardIMR(<imat>,[<maxrank>]) . . Hadamard bound of an IMR
##  Returns the a bound on all minors of the matrix.
##  If fasthadamard is set to false the maxrank-Lield of the IMR is used to
##    compute an upper bound on any rank minor.
##  The second argument replaces the actual maxrank of the imat as the
##    upper bound on the rank. In case of two arguments nothing is stored
##    in the IMR-Record.
HadamardIMR:=function(arg)
  local row,rows,col,cols,
    sum,
    normlist,entrylist,
    rowstart,colstart,
    ranked,    # reducing of hadamard bound via rank consideration
    rooted,    # root of product is taken rather than product of roots
    rproduct,cproduct, # product of row/column norms depending on 'rooted'
  # We prefer root of product if the largest factor of the product is smaller
  # than 2^1000. This figure is based on empirical results with gap.
    maxrank,   # dimension of considered submatrices
    imat,mat,result,
    fasthadamard;

  imat:=arg[1];
  mat:=imat.matrix;
  rows:=Length(mat);
  cols:=Length(mat[1]);

  fasthadamard:=imat.RankOptions.fasthadamard;
  if Length(arg)=2 then
    maxrank:=arg[2];
    if maxrank=cols then ranked:=true; else ranked:=false; fi;
  else
    maxrank:=imat.maxrank;
```

```
  ranked:=false;
fi;

if IsBound(imat.bound) and (fasthadamard or not ranked) then
  InfoIMat2("#I  Remembered bound:10^",LogInt(imat.bound,10),"\n");
  return imat.bound;
fi;

InfoIMat3("#I    Hadamarding rows\c");
if fasthadamard or rows=cols and maxrank=cols then

  # this is the simple way to compute hadamard bounds
  #   just the product of row (resp. column) norms of the whole matrix
  rproduct:=1;
  for row in [1 .. rows] do
    sum:=0;
    for col in [1 .. cols] do
      sum:=sum+mat[row][col]^2;
    od;
    if sum>0 then rproduct:=rproduct*sum; fi;
  od;
  InfoIMat3("\b\b\b\bcolumns\c");
  cproduct:=1;
  for col in [1 .. cols] do
    sum:=0;
    for row in [1 .. rows] do
      sum:=sum+mat[row][col]^2;
    od;
    if sum>0 then cproduct:=cproduct*sum; fi;
  od;
  result:=RootInt(Minimum(rproduct,cproduct))+1;

  if not ranked then imat.bound:=result; fi;
  # we do not want to remember a bound which is based on a possibly wrong
  # rank
  InfoIMat3("\b\b\b\b\b\b\b\b...done.\n");
  InfoIMat2("#I  Hadamard bound:10^",LogInt(result,10),"\n");
  return result;
fi;

rowstart:=rows-maxrank+1;
colstart:=cols-maxrank+1;

# we start with row norms
normlist:=[];
for row in [1 .. rows] do
  entrylist:=[];
  for col in [1 .. cols] do
    Add(entrylist,mat[row][col]^2);
  od;
  if cols=maxrank then
    Add(normlist,Sum(entrylist));
  else

    # take only the 'maxrank' biggest entries
    Sort(entrylist);
    Add(normlist,Sum(entrylist{[colstart..cols]}));
  fi;
od;

if rows=maxrank then
```

```
  rooted:=false;
  rproduct:=Product(normlist);
else

  # take only the 'maxrank' biggest row norms
  Sort(normlist);

  # if the rownorms are big it is better to take root just here
  if normlist[rows]<2^1000 then
    rooted:=false;
    rproduct:=Product(normlist{[rowstart..rows]});
  else
    rooted:=true;
    rproduct:=Product(normlist{[rowstart..rows]},
                     x->RootInt(x)+1);
  fi;
fi;

normlist:=[];
InfoIMat3("\b\b\b\bcolumns\c");
for col in [1 .. cols] do
  entrylist:=[];
  for row in [1 .. rows] do
    Add(entrylist,mat[row][col]^2);
  od;
  if rows=maxrank then
    Add(normlist,Sum(entrylist));
  else

    # only the 'maxrank' biggest entries
    Sort(entrylist);
    Add(normlist,Sum(entrylist{[rowstart..rows]}));
  fi;
od;
if rooted then

  # if the row norm values are rooted we do that here as well
  if cols=maxrank then
    cproduct:=Product(normlist,x->RootInt(x)+1);
  else
    Sort(normlist);

    # only the 'maxrank' biggest values
    if normlist[cols]<2^1000 then
      cproduct:=RootInt(Product(normlist{[colstart..cols]}));
    else
      cproduct:=Product(normlist{[colstart..cols]},
                       x->RootInt(x)+1);
    fi;
  fi;
else
  if cols=maxrank then
    cproduct:=Product(normlist);
  else
    Sort(normlist);
    if normlist[cols]<2^1000 then
      cproduct:=Product(normlist{[colstart..cols]});
    else
      rooted:=true;
      cproduct:=Product(normlist{[colstart..cols]},
                       x->RootInt(x)+1);
```

```
        rproduct:=RootInt(rproduct);
      fi;
    fi;
  fi;

  if rooted then result:=Minimum(rproduct,cproduct);
            else result:= RootInt(Minimum(rproduct,cproduct))+1; fi;
  imat.bound:=result;
  InfoIMat3("\b\b\b\b\b\b\b\b...done.\n");
  InfoIMat2("#I  Improved Hadamard bound:10^",LogInt(result,10),"\n");
  return result;
end;


#############################################################################
#F  FiveHadamardIMR(<imat>) . . special Hadamard bound
##  This is the maximum of hadamard bounds for submatrices defined by the
##    fivemat data structure.

##  All those submatrices a full rank square, we take minimum

##    of product of row norms or of product column norms.
FiveHadamardIMR:=function(imat)
  local row,col,rowindeces,colindeces,lastrowindeces,
    rproduct, # product of row norms
    cproduct, # product of column norms
    sum,      # sum of squared entries
    maxsum,   # maximal row norm of the lastrows
    maxentry, # maximal squared entry of the lastrows at certain position
    mat,result;

  if imat.DeterminantOptions.fasthadamard and IsBound(imat.bound) then
    InfoIMat2("#I  Remembered bound:10^",LogInt(imat.bound,10),"\n");
    return imat.bound;
  fi;

  InfoIMat3("#I    Hadamarding rows\c");
  mat:=imat.matrix;
  rowindeces:=imat.fivemat.rows;
  colindeces:=imat.fivemat.cols;
  lastrowindeces:=imat.fivemat.lastrows;

  rproduct:=1;
  for row in rowindeces do
    sum:=0;
    for col in colindeces do
      sum:=sum+mat[row][col]^2;
    od;
    if sum>0 then rproduct:=rproduct*sum; fi;
  od;
  maxsum:=0;
  for row in lastrowindeces do
    sum:=0;
    for col in colindeces do
      sum:=sum+mat[row][col]^2;
    od;
    if sum>maxsum then maxsum:=sum; fi;
  od;
  rproduct:=rproduct*maxsum;

  InfoIMat3("\b\b\b\bcolumns\c");
```

```
  cproduct:=1;
  for col in colindeces do
    sum:=0;
    for row in rowindeces do
      sum:=sum+mat[row][col]^2;
    od;
    maxentry:=0;
    for row in lastrowindeces do
      if mat[row][col]>maxentry then maxentry:=mat[row][col]^2; fi;
    od;
    sum:=sum+maxentry;
    if sum>0 then cproduct:=cproduct*sum; fi;
  od;
  result:=RootInt(Minimum(rproduct,cproduct))+1;
  InfoIMat3("\b\b\b\b\b\b\b...done.\n");
  InfoIMat2("#I  New Hadamard bound:10^",LogInt(result,10),"\n");
  return result;
end;

#############################################################################
#F  ModRankIMR( <imat>,<modul> ) . . . . . . modular rank calculation
##    Performs a Gauss-Jordan in the field of residues modulo the prime modul.
##    Returns the rank and does some changes in the imat record if
##        applicable. This includes adjusting
##      minrank: if the rank modulo this prime is higher
##      rank: if the new minrank reached maxrank
##      fivehyp: this is the data structure to give hypothesises for the
##        'fivemat'
##          It is a record of five lists which contain
##        rows: indeces of rows
##        cols: indeces of columns which define the submatrix
##        lastrows: indeces of rows that will alternatively be last rows
##          which define a submatrix each
##        dets: list of determinants of each
##        modules: the prime modules
##        if those lists already exist the values are added to it
##      zeroprimes: a list of primes modulo which the matrix has not the
##          so far minimal rank
ModRankIMR:=function(imat,modul)
  local matrow,  # row of the matrix just considered
    matrownr,    # its index in the matrix
    tria,        # triangulized independant rows of the matrix
    triasize,    # number of rows in 'tria'
    triarow,     # one of 'tria'
    rowlist,     # matrix row indeces of the rows in 'tria'
    det,sgn,     # det*sgn is the so far determinant
    olddet,oldsgn, # old=the actual determinant one step before
    cdet,        # copy of det
    hyprows,hypcols,hyplastrows,hypdets, # entries for fivehyp field of imat
    colperm,     # column permutation. The rows in 'tria' are permuted.
    pos,tmp,
    count,       # so many lastrows are found so far
    maxcount,    # bound on number of lastrows
    nozero,      # entry of a transformed row - if really not zero, its row
      # contributes to the rank
    butlast;     # values of rows at the position where the last row which
      # contributed to the rank had its nonzero value which formed the
      # diagonal element in 'tria'

  if IsBound(imat.fivehyp) and modul in imat.fivehyp.modules then
    if imat.minrank=imat.maxrank then imat.rank:=imat.minrank; fi;
```

```
    InfoIMat3("R\c");
    return imat.minrank;
  fi;
  if IsBound(imat.zeroprimes) and modul in imat.zeroprimes then return 0; fi;

  InfoIMat3("    ");
  triasize:=0;
  tria:=[];
  rowlist:=[];
  colperm:=();
  matrownr:=1;
  det:=1;
  olddet:=1;
  sgn:=false;
  oldsgn:=false;
  butlast:=[];

  while matrownr<=imat.rows and triasize<imat.maxrank do

    InfoIMat3("\b\b\b\b",String(QuoInt(matrownr*100,imat.rows),3),"%\c");
    if imat.rows-matrownr<imat.minrank-triasize-1 then
      if IsBound(imat.zeroprimes) then
        Add(imat.zeroprimes,modul);
      else
        imat.zeroprimes:=[modul];
      fi;
      InfoIMat3("\b\b\b\bG\c");
      return 0;
    fi;
    ## This is the mechanism of skipping the rest of the GJA if it can
    ##   be forseen that the maxrank cannot be reached.

    # copy, column permutation, and canonisation
    matrow:=[];
    for pos in [1..imat.cols] do
      matrow[pos]:=imat.matrix[matrownr][pos^colperm] mod modul;
    od;

    # reduction with the so far triangle
    if triasize>0 then
      for pos in [1..triasize-1] do
        if matrow[pos]<>0 then
          SubTrafo(matrow,matrow[pos],tria[pos],pos+1,modul);
        fi;
      od;
      butlast[matrownr]:=matrow[triasize];
      if matrow[triasize]<>0 then
        SubTrafo(matrow,matrow[triasize],tria[triasize],triasize+1,modul);
      fi;
    fi;

    # look for the first non-zero entry
    pos:=triasize+1;
    while pos<=imat.cols and matrow[pos]=0 do pos:=pos+1; od;

    if pos<=imat.cols then
      # reduce the head element to 1 and adjust the determinant
      olddet:=det;
      if matrow[pos]<>1 then
        det:=det*matrow[pos] mod modul;
        MultTrafo(1/matrow[pos] mod modul,matrow,pos+1,modul);
```

```
    fi;
    # independent row found, add them to our triangle
    triasize:=triasize+1;
    tria[triasize]:=matrow;
    rowlist[triasize]:=matrownr;
    # permutate the columns to get the head element in the diagonal
    if pos <> triasize then
      colperm:=(pos,triasize)*colperm;
      for triarow in tria do
        tmp:=triarow[pos];
        triarow[pos]:=triarow[triasize];
        triarow[triasize]:=tmp;
      od;
    fi;
    # Note that the triangle is permuted the columns in the original matrix
    #   not yet. We permute them each time we got a new row to consider.
  fi;
  matrownr:=matrownr+1;
od;

if triasize<imat.minrank or triasize=0 then
  if IsBound(imat.zeroprimes) then
    Add(imat.zeroprimes,modul);
  else
    imat.zeroprimes:=[modul];
  fi;
  InfoIMat3("\b\b\b\bG\c");
  return 0;
fi;

InfoIMat3("\b\b\b\b 99%\c");

# Now either we considered all rows or we confirmed rank to be maxrank
# In either case we go back to the last independent row, take this as
# the first "fivestep row" including its determiant.
hyplastrows:=[rowlist[triasize]];

# We go back to 1 row after this fivesteprow and restore values as if
#   that row does not exist;
# We continue with the restricted set of columns which suffice to the rank.
matrownr:=rowlist[triasize]+1;
hypcols:=List([1..triasize],x->x^colperm);
Unbind(tria[triasize]);
Unbind(rowlist[triasize]);
triasize:=triasize-1;
hyprows:=rowlist;

# the last swapping does not effect the resulting part of our matrix
#   because we cut out all columns in between
if 2*det>modul then det:=det-modul; fi;
hypdets:=[det];
det:=olddet;

# Now we have a triangular form in the first cols-1 rows and columns.
# We check if they are independent form the (now) first col-1
#   rows. We collect those in the resultlist.

count:=2;
while count<=imat.ElementaryDivisorsOptions.maxfivers and
   matrownr<=imat.rows do
```

```
    if IsBound(butlast[matrownr]) then
      nozero:=butlast[matrownr];
    else
      # copy, column permutation, and canonisation
      matrow:=[];
      for pos in [1..triasize+1] do
        matrow[pos]:=imat.matrix[matrownr][pos^colperm] mod modul;
      od;

      # reduction with the triangle
      for pos in [1..triasize] do
        if matrow[pos]<>0 then
          SubTrafo(matrow,matrow[pos],tria[pos],pos+1,modul);
        fi;
      od;
      nozero:=matrow[triasize+1];
    fi;

    # collection of replacing rows
    if nozero<>0 then
      hyplastrows[count]:=matrownr;
      cdet:=det*nozero mod modul;
      if 2*cdet>modul then cdet:=cdet-modul; fi;
      hypdets[count]:=cdet;
      count:=count+1;
    fi;
    matrownr:=matrownr+1;
  od;

  # Assemble the results in the fivehyp record
  if IsBound(imat.fivehyp) and triasize+1=imat.minrank then
    Add(imat.fivehyp.rows,hyprows);
    Add(imat.fivehyp.cols,hypcols);
    Add(imat.fivehyp.lastrows,hyplastrows);
    Add(imat.fivehyp.dets,hypdets);
    Add(imat.fivehyp.modules,modul);
  else
    imat.minrank:=triasize+1;
    if imat.maxrank=triasize+1 then imat.rank:=triasize+1; fi;
    imat.fivehyp:= rec(
        rows:=    [hyprows],
        cols:=    [hypcols],
        lastrows:=[hyplastrows],
        dets:=    [hypdets],
        modules:= [modul]   );
  fi;
  InfoIMat3("\b\b\b\bG\c");
  return imat.minrank;
end;

#############################################################################
#F  RankIMR( <imat> )  . . . rank of an Integer Matrix
##    Returns the rank and updates the imat record
##      rank: as number
##      fivehyp: record of hypothesises for fivemats
##    When fastdet is set to true it will compute the rank only modulo
##    one prime and believe it if it is at least the minrank.
##    Otherwise it will confirm this rank by taking so many primes that
##    its product will exceed the hadamard bound on any minor which is
##    one larger than the rank modulo the first prime.
RankIMR:=function(imat)
```

```
local safeprimes, modul, index, tmprank;

if IsBound(imat.rank) and IsBound(imat.fivehyp) then
  InfoIMat2("#I  Remembered Rank:",imat.rank,"\n");
  return imat.rank;
fi;

if  imat.RankOptions.fastrank then
  InfoIMat3("#I    Rank guessing...\c");
  index:=1;
  # Take so many prime modules if a modular rank is not smaller then
  #   'minrank'. This is rarely not exactly one.
  while index<=Length(_primes) and
    ModRankIMR(imat,_primes[index])<imat.minrank do
    index:=index+1;
  od;
  if index>Length(_primes) then Error("\nminrank value too high"); fi;
  imat.rank:=imat.minrank;
  if imat.maxrank=imat.rank then InfoIMat3("...fullrank\n");
                            else InfoIMat3("...done\n"); fi;
else
  repeat
    # first guess the rank as above
    InfoIMat3("#I    Rank guessing...\c");
    index:=1;
    while index<=Length(_primes) and
      ModRankIMR(imat,_primes[index])<imat.minrank do
      index:=index+1;
    od;
    if index>Length(_primes) then Error("\nminrank value too high"); fi;


    if IsBound(imat.rank) then
      InfoIMat3("...fullrank\n");
      InfoIMat2("#I  Rank:",imat.rank,"\n");
      return imat.rank;
    else
      InfoIMat3("...done\n");
    fi;

    # Now confirm the rank by checking, that there is no non-zero minor of
    # a dimension higher than 'minrank'.
    tmprank:=imat.minrank;
    safeprimes:=BiggerPrimes(HadamardIMR(imat,tmprank+1));
    InfoIMat2("#I  Maximal ",Length(safeprimes)," primes considered.\n");
    InfoIMat3("#I    Rank confirming...\c");
    for modul in safeprimes do
      ModRankIMR(imat,modul);
      if IsBound(imat.rank) then
        InfoIMat3("...fullrank\n");
        InfoIMat2("#I  Rank:",imat.rank,"\n");
        return imat.rank;
      fi;
    od;
    InfoIMat3("...done\n");

    # if we did find a higher rank, repeat
  until imat.minrank=tmprank;
  imat.rank:=imat.minrank;
  imat.maxrank:=imat.rank;
fi;
```

```
  InfoIMat2("#I  Rank:",imat.rank,"\n");
  return imat.rank;
end;

#############################################################################
#F  FiveSelectIMR( <imat> ) . selects submatrix for determinant calculation
##  This procedure selects one of the hypothesises (given in fivehyp)
##    and puts it to the field 'fivemat'.
##  If the field 'fivehyp' is not defined yet it will call RankIMR which
##    will create it.
##  The fields 'fivedets' and 'fiveprimes' are initialized with the
##    corresponding fields in 'fivehyp'. It is at least the first
##    element of those lists but it could be more if different primes
##    give the same fivemat hypothesis.
FiveSelectIMR:=function(imat)
  local collection,result,
    recfivers,actfivers,recdets,actdets,
    i,j,k,collected;

  if IsBound(imat.fivemat) then return true; fi;

  # computing the rank which produces a fivehyp (unless rank 0)
  if (not IsBound(imat.fivehyp) or
      not IsBound(imat.rank)) and RankIMR(imat)=0
  then return false;
  fi;

  # Collect all those hypothesis with maximal number of restrows which
  #   have the same fivemat (given by rows, cols, and lastrows).
  collection:=[];
  recfivers:=0;
  for i in [1..Length(imat.fivehyp.modules)] do
    collected:=false;
    actfivers:=Length(imat.fivehyp.lastrows[i]);
    if actfivers>recfivers then
      recfivers:=actfivers;
      collection:=[[i]];
    elif actfivers=recfivers then
      j:=1;
      while j<=Length(collection) and not collected do
        k:=collection[j][1];
        if imat.fivehyp.rows[k]=imat.fivehyp.rows[i] and
           imat.fivehyp.cols[k]=imat.fivehyp.cols[i] and
           imat.fivehyp.lastrows[k]=imat.fivehyp.lastrows[i] then
          Add(collection[j],i);
          collected:=true;
        fi;
        j:=j+1;
      od;
      if not collected then Add(collection,[i]); fi;
    fi;
  od;

  # We take the fivemat where we have the determinants modulo the most
  #   primes. We don't distinguish between the maximal ones.
  # Each row 'res' contains now indeces of the original lists where the
  #   entries for rows, cols, and lastrows are the same.
  recdets:=0;
  for i in [1..Length(collection)] do
    actdets:=Length(collection[i]);
    if actdets>recdets then
```

```
      recdets:=actdets;
      result:=collection[i];
    fi;
  od;

  imat.fivemat:=rec(rows    :=imat.fivehyp.rows[result[1]],
                    cols    :=imat.fivehyp.cols[result[1]],
                    lastrows:=imat.fivehyp.lastrows[result[1]]);
  imat.fiveprimes:=List(result,x->imat.fivehyp.modules[x]);
  imat.fivedets:=List(result,x->imat.fivehyp.dets[x]);

  InfoIMat2("#I  Considering ",Length(imat.fivemat.lastrows),
           " submatrices.\n");
  return true;
end;


#############################################################################
#F  ModFiveStepIMR( <imat>,<modul> ) . modular determinant calculation
##    It expects the fields
##      fivemat: data structure which define matrices which are equal but
##        one row
##      fivers: number of those matrices
##    Adjusts the following fields in the imat record
##      fivedets: list of determinants corresponding to the fiverows field
##      fiveprimes: modul is appendend
##    The return value is the list of modular determinants
ModFiveStepIMR:=function(imat,modul)
  local matrow,
    tria,triarow,
    rank,
    det,sgn,cdet,fivedets,
    colperm,
    step,pos,tmp,count;

  if IsBound(imat.fiveprimes) and modul in imat.fiveprimes then
    InfoIMat3("R\c");
    return imat.fivedets[Position(imat.fiveprimes,modul)];
  fi;

  InfoIMat3("    ");
  rank:=Length(imat.fivemat.cols);
  tria:=[];
  colperm:=();
  det:=1;
  sgn:=false;
  fivedets:=[];

  for step in [1..rank-1] do

    InfoIMat3("\b\b\b\b",String(QuoInt(step*100,rank),3),"%\c");

    # copy, column permutation, and canonisation
    matrow:=[];
    for pos in [1..rank] do
      matrow[pos]:=imat.matrix[imat.fivemat.rows[step]]
                     [imat.fivemat.cols[pos^colperm]] mod modul;
    od;

    # reduction with the so far triangle
    for pos in [1..step-1] do
```

```
    if matrow[pos]<>0 then
      SubTrafo(matrow,matrow[pos],tria[pos],pos+1,modul);
    fi;
  od;

  # look for the first non-zero entry
  pos:=step;
  while pos<=rank and matrow[pos]=0 do pos:=pos+1; od;

  if pos<=rank then
    # reduce the head element to 1 and adjust the determinant
    if matrow[pos]<>1 then
      det:=det*matrow[pos] mod modul;
      MultTrafo(1/matrow[pos] mod modul,matrow,pos+1,modul);
    fi;
    # independent row found, add them to our triangle
    tria[step]:=matrow;

    # permutate the columns to get the head element in the diagonal
    if pos <> step then
      colperm:=(pos,step)*colperm;
      sgn:=not sgn;
      for triarow in tria do
        tmp:=triarow[pos];
        triarow[pos]:=triarow[step];
        triarow[step]:=tmp;
      od;
    fi;
    # Note that the triangle is permuted the columns in the original matrix
    #   not yet. We permute them each time we got a new row to consider.
  else
    for count in [1..Length(imat.fivemat.lastrows)] do
      fivedets[count]:=0;
    od;
    Add(imat.fiveprimes,modul);
    Add(imat.fivedets,fivedets);
    InfoIMat3("\b\b\b\bG");
    return fivedets;
  fi;
od;

InfoIMat3("\b\b\b\b 99%\c");

# Now we have a triangular form in the first rank-1 rows and columns.
# We add each of the fiverows as the last row and collect the determinants.
for count in [1..Length(imat.fivemat.lastrows)] do

  # copy, column permutation, and canonisation
  matrow:=[];
  for pos in [1..rank] do
    matrow[pos]:=imat.matrix[imat.fivemat.lastrows[count]]
                  [imat.fivemat.cols[pos^colperm]] mod modul;
  od;

  # reduction with the triangle
  for pos in [1..rank-1] do
    if matrow[pos]<>0 then
      SubTrafo(matrow,matrow[pos],tria[pos],pos+1,modul);
    fi;
  od;
```

```
    # collection of the determinants
    cdet:=det*matrow[rank] mod modul;
    if 2*cdet>modul then cdet:=cdet-modul; fi;
    if sgn then cdet:=-cdet; fi;
    fivedets[count]:=cdet;
  od;

  Add(imat.fiveprimes,modul);
  Add(imat.fivedets,fivedets);

  InfoIMat3("\b\b\b\bG");
  return fivedets;
end;

#############################################################################
#F  DetIMR( <imat> ) . . gives multiple of gcd of determinats of rankminors
##    Returns the gcd of rank minors
##    If the 'fivemat' field is not bound yet it will call FiveSelectIMR.
##    It uses the 'zeroprimes' field if present.
##    If the flag fastdet is set true then it will use the following
##      algorithm: After each prime it assembles the modular
##      determinants to a (minmal absolute value) result - for each
##      subdeterminant. If none of those changes during one step they will
##      be considered as final. The propability of this to fail is for one
##      submatrix 1/(the prime of this step) - under certain circumstances.
##      It decreases considerably for more submatrices.
##    Otherwise it will use the Hadamard bound.
##    It apppends the field
##      det: the gcd of those subdeterminants which are defined by the
##        'fivemat' fields
##    which is also the return value.
DetIMR:=function(imat)
  local product,prime,dets,olddets,primelist,index;

  if IsBound(imat.det) then
    InfoIMat2("#I  Remembered rankminor gcd:",imat.det,"\n");
    return imat.det;
  fi;

  # computing a fivemat if possible
  if not IsBound(imat.fivemat) and not FiveSelectIMR(imat) then
    return 0;
  fi;

  # if there are primes which have a lower modular rank on the whole matrix
  # the determinants are obviously zero
  if IsBound(imat.zeroprimes) then
    for prime in imat.zeroprimes do
      if not prime in imat.fiveprimes then
        Add(imat.fiveprimes,prime);
        Add(imat.fivedets,List([1..Length(imat.fivemat.lastrows)],x->0));
      fi;
    od;
  fi;

  if imat.DeterminantOptions.fastdet then

    # take so many modular determinants until combined determinants don't
    # change
    InfoIMat3("#I    Determinants calculating...\c");
    product:=Product(imat.fiveprimes);
```

```
    dets:=ListChineseRem(imat.fiveprimes,imat.fivedets);
    for index in [1..Length(_primes)] do
      prime:=_primes[index];
      if not prime in imat.fiveprimes then
        olddets:=dets;
        if dets=[] then dets:=ModFiveStepIMR(imat,prime);
        else dets:=ListChineseRem([product,prime],
                         [dets,ModFiveStepIMR(imat,prime)]);
        fi;
        if olddets=dets then

          # Determinants haven't chanbged in the last step, consider them
          # correct
          InfoIMat3("...done\n");
          imat.det:=Iterated(dets,GcdInt);
          InfoIMat2("#I  Rankminor gcd:",imat.det,"\n");
          return imat.det;
        fi;
        product:=product*prime;
      fi;
    od;
    imat.det:=Iterated(dets,GcdInt);
    InfoIMat3("...done\n");
  else

    # take so many modular determinants as suggested by the hadamard bound
    primelist:=Difference(BiggerPrimes(FiveHadamardIMR(imat)),
                      imat.fiveprimes);
    InfoIMat2("#I  Considering ",Length(primelist)," primes.\n");
    InfoIMat3("#I    Determinants calculating...\c");
    for prime in primelist do
      ModFiveStepIMR(imat,prime);
    od;
    imat.det:=Iterated(ListChineseRem(imat.fiveprimes,imat.fivedets),GcdInt);
    InfoIMat3("...done\n");
  fi;
  InfoIMat2("#I  Rankminor gcd:",imat.det,"\n");
  return imat.det;
end;

#############################################################################
#F  ModElDivIMR( <imat>,<prime>,<exponent> ) . modular elementary divisors
##    modulo prime power. If the matrix has modulo <prime>^<exponent> the same
##    rank as the 'rank'-field of <imat> then the modular elementary divisors
##    are returned. If it is lower then the exponent was too low and a field
##    'eldivfail' is inserted in <imat>. If it is higher then the rank
##    calculation failed. All values dependant of the rank are deleted and a
##    recursive call 'ElDivModIMR' is performed telling that the rank is at
##    least one higher than previously accepted. After the recursive call
##    of 'ElDivModIMR' returned the higher instance of 'ElDIvModIMR'
##    recognizes the return value "ready" and returns itself.
ElDivModIMR:=Ignore;  # recursion
ModElDivIMR := function (imat,prime,maxexp)
  local exp,count,modul,tmp,pivot,nozero,
    eldivisors,mat,
    found,       # boolean to leave the loop if the pivot is found
    row,col;     # indexes when running through the matrix

  if IsBound(imat.eldivconst) and
      prime in List(imat.eldivconst,x->x.prime) then
    return imat.eldivconst[Position(
```

```
       List(imat.eldivconst,x->x.prime),prime)].list;
  fi;
  if IsBound(imat.eldivfail) and
      prime in List(imat.eldivfail,x->x.prime) and
      imat.eldivfail[Position(
         List(imat.eldivfail,x->x.prime),prime)].exp>=exp then
    return [];
  fi;
  if IsBound(imat.noteldiv) and prime in imat.noteldiv then
    return List([1..imat.rank],x->1);
  fi;


  eldivisors:=[];        # That will be the result.
  mat:=Copy(imat.matrix);
  exp:=0;        # the exponent that the actual elementary divisor (count)
                 # would get if it is found
  modul:=prime^maxexp;   # the actual module for calculations
                 # note that the power decreases when the actual rest of
                 # the matrix is divided by prime
  count:=1;      # this is the count for the elamantary factors but for
                 # matrix positions as well; it ranges from 1 to rank
  InfoIMat3("    ");


  # The loop to look for a good pivot which is one that is invertible.
  # Note that we do not consider any heuristics (row-column-sum) whatsoever
  #   because there is no fear of large entries - they cover the whole
  #   residue ring anyway and small values are not cheaper than big ones.
  #   The only good features could be
  #     1) One as pivot to save a vector multiplication
  #     2) Zeros in the column to save a SubTrafo for each
  #   Both mean a trade-off between search and calculation.
  repeat         # loop quite like "for count in [1..rank]"
    InfoIMat3("\b\b\b\b",String(QuoInt(count*100,imat.rank),3),"%\c");
    found:=false;
    row:=count;
    repeat       # loop of row
      col:=count;
      repeat     # loop of col
        if mat[row][col] mod prime <>0 then  # invertible in the ring
          found:=true;
        else col:=col+1;
        fi;
      until found or col>imat.cols;
    row:=row+1;
    until found or row>imat.rows;

    if found then
      # We have found an elemantary divisor.

      # swap row and column that the invertible gets to the left top
      # note that variable names row and tmp are recycled
      row:=row-1;  # col is corrrect value because increment is in else branch
      tmp:=mat[count];
      mat[count]:=mat[row];
      mat[row]:=tmp;
      for row in [count .. imat.rows] do
        tmp:=mat[row][count];
        mat[row][count]:=mat[row][col];
        mat[row][col]:=tmp;
```

```
    od;

    # Make the pivot 1
    pivot:=mat[count][count] mod modul;
    if pivot<>1 then
      MultTrafo(1/pivot mod modul,mat[count],count+1,modul);
    fi;

    for row in [count+1..imat.rows] do
      nozero:=mat[row][count] mod modul;
      if nozero<>0 then
        SubTrafo(mat[row],nozero,mat[count],count+1,modul);
      fi;
    od;

    # grasp the elemantary divisor
    eldivisors[count]:=prime^exp;
    count:=count+1;

  # No invertible element has been found. So prime divides all entries
  #   in the matrix. We divide and readjust the module. Actually we
  #   consider a matrix with a factor (prime^exp) in front.
  else
    exp:=exp+1;
    if exp=maxexp then

      # The exponent was too low, information about this is stored
      if IsBound(imat.eldivfail) then
        Add(imat.eldivfail,rec(prime:=prime,exp:=exp));
      else
        imat.eldivfail:=[rec(prime:=prime,exp:=exp)];
      fi;
      InfoIMat3("\b\b\b\b");
      return [];
    fi;
    modul:=prime^(maxexp-exp);
    for row in [count..imat.rows] do
      for col in [count..imat.cols] do
        mat[row][col]:=mat[row][col]/prime;
      od;
    od;
  fi;
until count>imat.rank;

# We now have the chance to check if the rank is wrong
for row in [count..imat.rows] do
  for col in [count..imat.cols] do
    if mat[row][col]<>0 then
      InfoIMat3("\b\b\b\b\n");
      InfoIMat2("Fatal: Rang was wrong, start all over\n");
      imat.maxrank:=imat.cols;
      imat.minrank:=imat.rank+1;
      Unbind(imat.rank);
      if IsBound(imat.fivehyp) then
        if IsBound(imat.zeroprimes) then
          Append(imat.zeroprimes,imat.fivehyp.modules);
        else
          imat.zeroprimes:=imat.fivehyp.modules;
        fi;
        Unbind(imat.fivehyp);
      fi;
```

```
        if IsBound(imat.fivemat) then Unbind(imat.fivemat); fi;
        if IsBound(imat.fivedets) then Unbind(imat.fivedets); fi;
        if IsBound(imat.fiveprimes) then Unbind(imat.fiveprimes); fi;
        if IsBound(imat.det) then Unbind(imat.det); fi;
        if IsBound(imat.factors) then Unbind(imat.factors); fi;
        if IsBound(imat.noteldiv) then Unbind(imat.noteldiv); fi;
        if IsBound(imat.eldivconst) then Unbind(imat.eldivconst); fi;
        if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
        ElDivModIMR(imat);
        return "ready";
      fi;
    od;
  od;

  if exp>0 then

    # There are non-trivial modular elementary divisors - store them.
    if IsBound(imat.eldivconst) then
      Add(imat.eldivconst,rec(prime:=prime,
                              exp:=exp+1,
                             list:=eldivisors));
    else
      imat.eldivconst:=[rec(prime:=prime,
                              exp:=exp+1,
                             list:=eldivisors)];
    fi;
    InfoIMat3("\b\b\b\b");
    return eldivisors;
  else

    # There are no non-trivial modular elementary divisors - remember that.
    if IsBound(imat.noteldiv) then
      Add(imat.noteldiv,prime);
    else
      imat.noteldiv:=[prime];
    fi;
    InfoIMat3("\b\b\b\b");
    return eldivisors;
  fi;
end;

#############################################################################
#F  ElDivModIMR( <imat> ) . elementary divisors modular method
##  Main procedure modular method
ElDivModIMR := function (imat)
  local result,tupel,count,det,
    fail,fails,
    eldiv,exp;

  InfoIMat1("#I  ElementaryDivisorsIMat(modular) called\n");

  if DetIMR(imat)=0 then
    imat.eldivisors:=[];
    return [];
  fi;

  result:=[];
  for count in [1..imat.rank] do result[count]:=1; od;
  if not AbsInt(imat.det)=1 then
    InfoIMat3("#I    Factorizing...\c");
    if not IsBound(imat.factors) then
```

```
      imat.factors:=Collected(FactorsInt(AbsInt(imat.det))); fi;
    InfoIMat3("\b\b\b\b\b\b\b\bs:",Concatenation(List(imat.factors,
      x->Concatenation(String(x[1]),['^'],String(x[2]),['*']))),"\b    \n");
    if imat.ElementaryDivisorsOptions.safeexp then
      InfoIMat3("#I    Taking factor...\c");
      for tupel in imat.factors do
        InfoIMat3(tupel[1],"^",tupel[2]+1," \c");
        if ModElDivIMR( imat,tupel[1],tupel[2]+1)="ready" then
          return imat.eldivisors; fi;
      od;
    else
      InfoIMat3("#I    Taking factor...\c");
      for tupel in imat.factors do
        exp:=Minimum(Maximum(2,LogInt(2^14,tupel[1])),tupel[2]+1);
        InfoIMat3(tupel[1],"^",exp," \c");
        if ModElDivIMR(imat,tupel[1],exp)="ready" then
          return imat.eldivisors; fi;
      od;
      if IsBound(imat.eldivfail) then

        # the 'eldivfail' field is a agenda-like data structure: while

        #   working through it new entries can be introduced
        while imat.eldivfail<>[] do
          fails:=imat.eldivfail;
          imat.eldivfail:=[];
          for fail in fails do
            InfoIMat3(fail.prime,"^",fail.exp+1," \c");
            if ModElDivIMR(imat,fail.prime,fail.exp+1)="ready" then
          return imat.eldivisors; fi;
          od;
        od;
      fi;
    fi;
    InfoIMat3("...done\n");
    if IsBound(imat.eldivconst) then
      for eldiv in imat.eldivconst do
        result:=ElementProduct(result,eldiv.list);
      od;
    fi;
  fi;
  imat.eldivisors:=result;
  InfoIMat2("#I  Elementary Divisors:",result,"\n");
  InfoIMat1("#I  ElementaryDivisorsIMat(modular) returns\n");
  return result;
end;

#############################################################################
##
#H  Simple subprocedures for the non-modular methods
##
#############################################################################
#F  Gcdex2( <m>, <n> ) . . . . . . . . . greatest common divisor of integers
##    gives a det=1 coeff-matrix in differnece to the standard GAP-function
##      'Gcdex', otherwise the same
Gcdex2 := function ( m, n )
    local   f, g, h, fm, gm, hm, q, s;
    if m>=0  then f:=m; fm:=1; else f:=-m; fm:=-1; fi;
    if n>=0  then g:=n; gm:=0; else g:=-n; gm:=0;  fi;
    while g <> 0  do
        q := QuoInt( f, g );
```

```
        h := g;          hm := gm;
        g := f - q * g;  gm := fm - q * gm;
        f := h;          fm := hm;
    od;
    if n = 0  then
        return rec( gcd := f, coeff1 := fm,coeff2 := 0,
                              coeff3 := 0 ,coeff4 := fm );
    else
        return rec( gcd := f, coeff1 := fm, coeff2 :=  (f - fm * m) / n,
                              coeff3 := -n/f,coeff4 := m / f );
    fi;
end;

#############################################################################
##
#H  Main procedures for the non-modular methods
##
#############################################################################
#F  DescentsIMR(imat,pivotrow,pivotcol,poslist,rowfirst)
##  Gives all possible descents of type 2 and 3 of a pivot position
##    (pivotrow,pivotcol)
##    and the direction rowfirst only using the indeces in poslist as
##    candidates for a gcd-transformation. Only in case of type 2 it can
##    happen that no descent (as []) is returned if the candidates for a
##    gcd-transformation haven't got a gcd which is equal to the gcd of the
##    whole primary vector.
##  The result is a list of descents which is a list of records with the
##    integer field index: (which vector to clear) and the boolean file
##    gcd: (if a gcd-transformation is required to clear).
##  It is a graph algorithm with hypothetical pivot values as knots and
##    candidates for gcd-transformations as edges. A valid descent is an
##    edge which runs from the present pivot value to the vector divisor.
DescentsIMR:=function(imat,pivotrow,pivotcol,poslist,rowfirst)
  local row,rows,col,cols,
    mat,pivot,
    Edges,SEdges,Edge,SEdge,Knots,Knot,
    Match,
    actual,pos,entry,gcd,i,
    divisor,norm,
    allindeces,indeces,indexlist,index,restindeces,
    bound,steps,
    descent,descentlist,descentstep,
    minalt,min;
  rows:=Length(imat.amat);
  cols:=Length(imat.amat[1]);
  mat:=imat.amat;

  pivot:=mat[pivotrow][pivotcol];

  # Make the elementary edges
  Edges:=[];
  Knots:=[pivot];
  actual:=1;
  if rowfirst then
    while actual<=Length(Knots) do
      Knot:=Knots[actual];
      for pos in poslist do
        if pos<>pivotcol then
          entry:=mat[pivotrow][pos];
          gcd:=GcdInt(entry,Knot);
          if gcd<>Knot and
```

```
            not ForAny(Edges,x->x.start=Knot and x.aim=gcd) then
          Add(Edges,rec(start:=Knot,
                        aim  :=gcd,
                        index:=[pos],
                        steps:=[Knot]));
          if not gcd in Knots then Add(Knots,gcd); fi;
        fi;
      fi;
    od;
    actual:=actual+1;
  od;
else
  while actual<=Length(Knots) do
    Knot:=Knots[actual];
    for pos in poslist do
      if pos<>pivotrow then
        entry:=mat[pos][pivotcol];
        gcd:=GcdInt(entry,Knot);
        if gcd<>Knot and
            not ForAny(Edges,x->x.start=Knot and x.aim=gcd) then
          Add(Edges,rec(start:=Knot,
                        aim  :=gcd,
                        index:=[pos],
                        steps:=[Knot]));
          if not gcd in Knots then Add(Knots,gcd); fi;
        fi;
      fi;
    od;
    actual:=actual+1;
  od;
fi;

if rowfirst then divisor:=imat.rowdiv[pivotrow];
                 norm:=imat.colnorm;
                 allindeces:=Set(Concatenation(
                      [1..pivotcol-1],[pivotcol+1..cols]));
            else divisor:=imat.coldiv[pivotcol];
                 norm:=imat.rownorm;
                 allindeces:=Set(Concatenation(
                      [1..pivotrow-1],[pivotrow+1..rows]));
fi;

if not divisor in Knots then return []; fi;

# Connect edges to form edges running through several knots
bound:=imat.ElementaryDivisorsOptions.stepbound;
if bound>1 then
  actual:=1;
  while actual<=Length(Edges) do
    Edge:=Edges[actual];
    steps:=Length(Edge.steps);
    if steps<bound then
      for i in [1..Length(Edges)] do
        Match:=Edges[i];
        if Match.start=Edge.aim and
            Length(Match.steps)+steps<=bound then
          Add(Edges,rec(start:=Edge.start,
                      aim  :=Match.aim,
                      index:=Concatenation(Edge.index,Match.index),
                      steps:=Concatenation(Edge.steps,Match.steps)));
        fi;
```

```
        od;
      fi;
      actual:=actual+1;
    od;
  fi;

  # Fill in the not pivot reducing indeces as sub-trafos
  descentlist:=[];
  for descent in Edges do
    if  descent.start=pivot and
        descent.aim=divisor then
      indexlist:=[];
      restindeces:=Difference(allindeces,descent.index);
      for i in [1..Length(descent.steps)] do
        descentstep:=descent.steps[i];
        for pos in Copy(restindeces) do
          if rowfirst then entry:=mat[pivotrow][pos];
                      else entry:=mat[pos][pivotcol]; fi;
          if entry mod descentstep =0 then
            if entry<>0 then
              Add(indexlist,rec(index:=pos,gcd:=false));
            fi;
            RemoveSet(restindeces,pos);
          fi;
        od;
        Add(indexlist,rec(index:=descent.index[i],gcd:=true));
      od;
      Append(indexlist,List(restindeces,x->rec(index:=x,gcd:=false)));
      Add(descentlist,indexlist);
    fi;
  od;
  return descentlist;
end;

#############################################################################
#F  ZeroDescentIMR(imat,pivotrow,pivotcol,rowfirst)
##  This produces a descent of type 1 compatible to the function
##    DescentsIMR.
ZeroDescentIMR:=function(imat,pivotrow,pivotcol,rowfirst)
  local indeces,mat;

  mat:=imat.amat;
  if rowfirst then indeces:=
      Concatenation([1..pivotcol-1],[pivotcol+1..Length(imat.amat[1])]);
  else indeces:=
      Concatenation([1..pivotrow-1],[pivotrow+1..Length(imat.amat)]);
  fi;
  return [List(Filtered(indeces,y->rowfirst and mat[pivotrow][y]<>0 or
                             not rowfirst and mat[y][pivotcol]<>0),
               x->rec(index:=x,gcd:=false))];
end;

#############################################################################
#F  ValidDescentIMR(imat,pivotrow,pivotcol,rowfirst)
##  This procedure checks which descent type is applicable and prepares
##    the input for the two above procedures whose output it returns.
ValidDescentsIMR:=function(imat,row,col,rowfirst)
  local mat,pivot,descents,indeces;
  mat:=imat.amat;

  descents:=[];
```

```
  pivot:=mat[row][col];if pivot<0 then pivot:=-pivot; fi;
  if rowfirst then
    if pivot=imat.rowdiv[row] and pivot=imat.coldiv[col] then
      descents:=ZeroDescentIMR(imat,row,col,true);
    else
      if not IsBound(imat.stepbound) or imat.stepbound<>0 then
        if imat.rowdiv[row]=1 then
          descents:=DescentsIMR(imat,row,col,
           Filtered(imat.colsort,x->x<>col),true);
        elif imat.coldiv[col] mod imat.rowdiv[row] =0 then
          descents:=DescentsIMR(imat,row,col,
           Filtered(imat.colsort,x->x<>col and
                    imat.coldiv[x] mod imat.rowdiv[row]=0),true);
        fi;
      fi;
    fi;
  else
    if pivot=imat.rowdiv[row] and pivot=imat.coldiv[col] then
      descents:=ZeroDescentIMR(imat,row,col,false);
    else
      if not IsBound(imat.stepbound) or imat.stepbound<>0 then
        if imat.coldiv[col]=1 then
          descents:=DescentsIMR(imat,row,col,
            Filtered(imat.rowsort,x->x<>row),false);
        elif imat.rowdiv[row] mod imat.coldiv[col] =0 then
          descents:=DescentsIMR(imat,row,col,
           Filtered(imat.rowsort,x->x<>row and
                    imat.rowdiv[x] mod imat.coldiv[col]=0),false);
        fi;
      fi;
    fi;
  fi;
  return descents;
end;
#F  Alpha(n1,n2)
##  This is the typical expectation of the 1-norm of the sum of two vectors
##    with the 1-norms n1 and n2.
Alpha:=function(n1,n2)
  if n1>n2 then
    return Maximum(n1,4/3*n2);
  else
    return Maximum(4/3*n1,n2);
  fi;
end;

#############################################################################
#F  DamageIMR(imat,pivotrow,pivotcol,rowfirst,descent)
##  This procedure computes the predicted damage of a descent.
DamageIMR:=function(imat,pivotrow,pivotcol,rowfirst,descent)
  local norm,pivot,pivotpos,mat,N,damage,g,c1,c2,c3,c4,
    t1,t2,mult,step;

  if rowfirst then norm:=imat.colnorm;
             pivotpos:=pivotcol;
  else           norm:=imat.rownorm;
             pivotpos:=pivotrow;
  fi;

  mat:=imat.amat;
  pivot:=mat[pivotrow][pivotcol];if pivot<0 then pivot:=-pivot; fi;
  N:=norm[pivotpos]-pivot;
```

```
  damage:=0;
  for step in descent do
    if step.gcd then
      if rowfirst then g:=Gcdex2(pivot,mat[pivotrow][step.index]);
                  else g:=Gcdex2(pivot,mat[step.index][pivotcol]); fi;
      c1:=g.coeff1;if c1<0 then c1:=-c1; fi;
      c2:=g.coeff2;if c2<0 then c2:=-c2; fi;
      c3:=g.coeff3;if c3<0 then c3:=-c3; fi;
      c4:=g.coeff4;if c4<0 then c4:=-c4; fi;

      t1:=c3*N;t2:=c4*norm[step.index];
      if t1>t2 then t2:=QuoInt(4*t2,3); else t1:=QuoInt(4*t1,3); fi;
      if t1>t2 then damage:=damage+t1-norm[step.index];
               else damage:=damage+t2-norm[step.index]; fi;

      t1:=c1*N;t2:=c2*norm[step.index];
      if t1>t2 then t2:=QuoInt(4*t2,3); else t1:=QuoInt(4*t1,3); fi;
      if t1>t2 then N:=t1; else N:=t2; fi;

      pivot:=g.gcd;
    else
      if rowfirst then mult:=mat[pivotrow][step.index]/pivot;
                  else mult:=mat[step.index][pivotcol]/pivot; fi;
      if mult<0 then mult:=-mult; fi;
      t1:=norm[step.index];t2:=mult*N;
      if t1>t2 then t2:=QuoInt(4*t2,3); else t1:=QuoInt(4*t1,3); fi;
      if t1>t2 then damage:=damage+t1-norm[step.index];
               else damage:=damage+t2-norm[step.index]; fi;
    fi;
  od;
  return damage;
end;

##############################################################################
#F  ParamUpdate(imat,type)
##  Each time a pivot is selected or abandoned this procedure is calkled
##    to update the parameters tolerate, intolerate, pivotbound,
##    descentbound, stepbound. It uses the set of adjustment parameters
##    (up, down, top, bot) for each.
##  It uses the scheme: if a restriction parameter was actually used then
##    its further application is made more inpropable.
ParamUpdate:=function(imat,type)
  local opts;
  opts:=imat.ElementaryDivisorsOptions;

  if type="tol" then
        opts.tolerate:=Maximum(opts.tolerate-opts.toldown,opts.tolbot);
        opts.intolerate:=
          Maximum(opts.intolerate-opts.intoldown,opts.intolbot);
        opts.pivotbound:=Maximum(opts.pivotbound-opts.pivdown,opts.pivbot);
        opts.descentbound:=
          Maximum(opts.descentbound-opts.desdown,opts.desbot);
  elif type="piv" or type="pfail" then
        opts.tolerate:=Minimum(opts.tolerate+opts.tolup,opts.toltop);
        opts.pivotbound:=Minimum(opts.pivotbound+opts.pivup,opts.pivtop);
        opts.descentbound:=
          Maximum(opts.descentbound-opts.desdown,opts.desbot);
        opts.stepbound:=Minimum(opts.stepbound+opts.stepup,opts.steptop);
  elif type="des" or type="dfail" then
        opts.tolerate:=Minimum(opts.tolerate+opts.tolup,opts.toltop);
        opts.pivotbound:=Maximum(opts.pivotbound-opts.pivdown,opts.pivbot);
```

```
        opts.descentbound:=Minimum(opts.descentbound+opts.desup,opts.destop);
        opts.stepbound:=Maximum(opts.stepbound-opts.stepdown,opts.stepbot);
  elif type="none" or type="nfail" then
        opts.tolerate:=Minimum(opts.tolerate+opts.tolup,opts.toltop);
        opts.pivotbound:=Maximum(opts.pivotbound-opts.pivdown,opts.pivbot);
        opts.descentbound:=
          Maximum(opts.descentbound-opts.desdown,opts.desbot);
  else Error("fatal internal");
  fi;
  if type="pfail" or type="dfail" or type="nfail" then
        opts.intolerate:=Minimum(opts.intolerate+opts.intolup,opts.intoltop);
  fi;
  InfoIMat4("\n#I   Bounds: tolerate:",Int(100*opts.tolerate),
                 "% intolerate:",Int(100*opts.intolerate),
                     "% pivots:",opts.pivotbound,
                    " descents:",opts.descentbound,
                       " steps:",opts.stepbound," \c");

end;

#############################################################################
#F  PivotIMR(imat)
##  This procedure selects a pivot with a descent. It gets the possible
##    pivot postions from the sorted list of row/column indeces from
##    which it selects a good comprimise between low row and column norm.
##  It tries both first row and first column elimination.
##  All possible descents for these values are computed and their damages
##    are compared.
##  It stops comparing if the damage was low enough (tolerate), if enough
##    non-zero positions are used (pivotbound) if enough descents were
##    calculated (descentbound).
##  It returns false if no or no acceptable descent (intolerate) was found.
PivotIMR:=function(imat)
  local min,best,mat,row,rows,col,cols,sum,pos,
    pcount,dcount,
    pivot,descent,descents,damage,rowfirst,
    tolerate,intolerate,pivotbound,descentbound;
  mat:=imat.amat;
  rows:=Length(imat.rowsort);   # number of nonzero rows
  if rows=0 then return false; fi;
  cols:=Length(imat.colsort);   # number of nonzero columns
  if cols=0 then return false; fi;

  pcount:=0;
  dcount:=0;
  tolerate:=Int(imat.ElementaryDivisorsOptions.tolerate*imat.matnorm);
  intolerate:=Int(imat.ElementaryDivisorsOptions.intolerate*imat.matnorm);
  pivotbound:=imat.ElementaryDivisorsOptions.pivotbound;
  descentbound:=imat.ElementaryDivisorsOptions.descentbound;

  min:=[];      # larger than every number

  for sum in [2..rows+cols] do
    for pos in [Maximum(1,sum-cols)..Minimum(sum-1,rows)] do
      row:=imat.rowsort[pos];       # indeces analog CANTOR diagonal method
      col:=imat.colsort[sum-pos];
      pivot:=mat[row][col];
      if pivot<>0 then
############## for one pivot position ########################################
        pcount:=pcount+1;
        # row first
```

```
descents:=ValidDescentsIMR(imat,row,col,true);
dcount:=dcount+Length(descents);
for descent in descents do
  damage:=DamageIMR(imat,row,col,true,descent);
  if damage<min then
    if damage<=tolerate then
      InfoIMat3("\n#I Tolerate ",Ordinal(imat.step)," descent, length "
         ,Number(descent,x->x.gcd),", from ",dcount,
        ": m[",row,"][",col,"]:=",pivot,
        " (row)\n#I  predicted damage: ",Int(damage)," \c");
      ParamUpdate(imat,"tol");
      return rec(row:=row,col:=col,rowfirst:=true,isolist:=descent);
    fi;
    best:=rec(row:=row,col:=col,rowfirst:=true,isolist:=descent);
    min:=damage;
  fi;
od;
if dcount>descentbound and min<>[] then
  InfoIMat3("\n#I Convinced of ",Ordinal(imat.step)," descent, length "
     ,Number(best.isolist,x->x.gcd),", from ",dcount,": m[",best.row,
    "][",best.col,"]:=",mat[best.row][best.col]," (");
  if best.rowfirst then InfoIMat3("row)");
                   else InfoIMat3("col)"); fi;
  InfoIMat3("\n#I  predicted damage: ",Int(min)," \c");
  if min>intolerate then
    InfoIMat3("\n#I But not taking it, intolerable damage! \c");
    ParamUpdate(imat,"dfail");
    return false;
  fi;
  ParamUpdate(imat,"des");
  return best;
fi;

# column first
descents:=ValidDescentsIMR(imat,row,col,false);
dcount:=dcount+Length(descents);
for descent in descents do
  damage:=DamageIMR(imat,row,col,false,descent);
  if damage<min then
    if damage<=tolerate then
      InfoIMat3("\n#I Tolerate ",Ordinal(imat.step)," descent, length "
         ,Number(descent,x->x.gcd),", from ",dcount,": m[",row,
        "][",col,"]:=",pivot," (col)\n#I  predicted damage: ",
         Int(damage)," \c");
      ParamUpdate(imat,"tol");
      return rec(row:=row,col:=col,rowfirst:=false,isolist:=descent);
    fi;
    best:=rec(row:=row,col:=col,rowfirst:=false,isolist:=descent);
    min:=damage;
  fi;
od;
if dcount>descentbound and min<>[] then
  InfoIMat3("\n#I Convinced of ",Ordinal(imat.step)," descent, length "
     ,Number(best.isolist,x->x.gcd),", from ",dcount,": m[",best.row,
    "][",best.col,"]:=",mat[best.row][best.col]," (");
  if best.rowfirst then InfoIMat3("row)");
                   else InfoIMat3("col)"); fi;
  InfoIMat3("\n#I  predicted damage: ",Int(min)," \c");
  if min>intolerate then
    InfoIMat3("\n#I But not taking it, intolerable damage! \c");
    ParamUpdate(imat,"dfail");
```

```
          return false;
        fi;
        ParamUpdate(imat,"des");
        return best;
      fi;

      if pcount>pivotbound and min<>[] then
        InfoIMat3("\n#I Persuaded of ",Ordinal(imat.step)," descent, length "
           ,Number(best.isolist,x->x.gcd),", from ",dcount,": m[",best.row,
          "][",best.col,"]:=",mat[best.row][best.col]," (");
        if best.rowfirst then InfoIMat3("row)");
                         else InfoIMat3("col)"); fi;
        InfoIMat3("\n#I  predicted damage: ",Int(min)," \c");
        if min>intolerate then
          InfoIMat3("\n#I But not taking it, intolerable damage! \c");
          ParamUpdate(imat,"pfail");
          return false;
        fi;
        ParamUpdate(imat,"piv");
        return best;
      fi;
############################################################################
     fi;
    od;
  od;
  if min=[] then
    InfoIMat3("\n#I No Pivot found! ");
    ParamUpdate(imat,"fail");
    return false;
  fi;
  InfoIMat3("\n#I Select ",Ordinal(imat.step)," descent, length ",
    Number(best.isolist,x->x.gcd),", from ",dcount,": m[",best.row,
    "][",best.col,"]:=",mat[best.row][best.col]," (");
  if best.rowfirst then InfoIMat3("row)");
                   else InfoIMat3("col)"); fi;
  InfoIMat3("\n#I  predicted damage: ",Int(min)," \c");
  if min>intolerate then
    InfoIMat3("\n#I But not taking it, intolerable damage! \c");
    ParamUpdate(imat,"nfail");
    return false;
  fi;
  ParamUpdate(imat,"none");
  return best;
end;

############################################################################
#F  SwapTrafoIMR(imat,pos1,pos2,rowop) . . . elementary matrix operation
##  Swaps two vektors and adjusts transformation matrices if applicable.
SwapTrafoIMR:=function(imat,pos1,pos2,rowop)
  local row,tmp;
  if rowop then
    tmp:=imat.amat[pos1];
    imat.amat[pos1]:=imat.amat[pos2];
    imat.amat[pos2]:=tmp;
    if imat.ElementaryDivisorsOptions.isom then
      tmp:=imat.left[pos1];
      imat.left[pos1]:=imat.left[pos2];
      imat.left[pos2]:=tmp;
    fi;
  else
    for row in [1..Length(imat.amat)] do
```

```
      tmp:=imat.amat[row][pos1];
      imat.amat[row][pos1]:=imat.amat[row][pos2];
      imat.amat[row][pos2]:=tmp;
    od;
    if imat.ElementaryDivisorsOptions.isom then
      for row in [1..imat.cols] do
        tmp:=imat.right[row][pos1];
        imat.right[row][pos1]:=imat.right[row][pos2];
        imat.right[row][pos2]:=tmp;
      od;
    fi;
  fi;
end;
##

#############################################################################
#F  MultTrafoIMR(imat,pos,rowop) . . . elementary matrix operation
##  Multiplies a vector with -1 and adjusts the transformation matrices
##    if applicable.
MultTrafoIMR:=function(imat,pos,rowop)
  local row,tmp;
  if rowop then
    imat.amat[pos]:=-imat.amat[pos];
    if imat.ElementaryDivisorsOptions.isom then
      imat.left[pos]:=-imat.left[pos];
    fi;
  else
    for row in [1..Length(imat.amat)] do
      imat.amat[row][pos]:=-imat.amat[row][pos];
    od;
    if imat.ElementaryDivisorsOptions.isom then
      for row in [1..imat.cols] do
        imat.right[row][pos]:=-imat.right[row][pos];
      od;
    fi;
  fi;
end;
##

#############################################################################
#F  SubTrafoIMR(imat,pos1,pos2,mult,rowop) . . . elementary matrix operation
##  Subtracts a multiple of vector (pos1) from a vector (pos2) and
##    adjusts the transformation matrices if applicable.
SubTrafoIMR:=function(imat,pos1,pos2,mult,rowop)
  local row;
  if mult=0 then return; fi;
  if rowop then
    imat.amat[pos2]:=imat.amat[pos2]-mult*imat.amat[pos1];
    if imat.ElementaryDivisorsOptions.isom then
      imat.left[pos2]:=imat.left[pos2]-mult*imat.left[pos1];
    fi;
  else
    for row in [1..Length(imat.amat)] do
      imat.amat[row][pos2]:=imat.amat[row][pos2]-
                            mult*imat.amat[row][pos1];
    od;
    if imat.ElementaryDivisorsOptions.isom then
      for row in [1..imat.cols] do
        imat.right[row][pos2]:=imat.right[row][pos2]-
                              mult*imat.right[row][pos1];
      od;
```

```
    fi;
  fi;
end;
##

#############################################################################
#F  GcdTrafoIMR(imat,pos1,pos2,h1,h2,rowop) . . . elementary matrix operation
##  Performs a gcd-transformation with h1 and h2 thought as entries at a
##    certain position of vectors pos1 and pos2 resp. so that the gcd
##    comes to the place of h1 and 0 comes to the position of h2.
GcdTrafoIMR:=function(imat,pos1,pos2,h1,h2,rowop)
  local row,g,tmp,mat,left,right;
  mat:=imat.amat;
  if imat.ElementaryDivisorsOptions.isom then
      left:=imat.left;right:=imat.right; fi;
  if h2=0 then return; fi;
  g:=Gcdex2(h1,h2);
  if g.coeff2=0 then
    SubTrafoIMR(imat,pos1,pos2,h2/h1,rowop);
  elif g.coeff1=0 then
    SubTrafoIMR(imat,pos2,pos1,h1/h2,rowop);
    SwapTrafoIMR(imat,pos1,pos2,rowop);
  else
    if rowop then
      tmp:=g.coeff1*mat[pos1]+g.coeff2*mat[pos2];
      mat[pos2]:=g.coeff3*mat[pos1]+g.coeff4*mat[pos2];
      mat[pos1]:=tmp;
      if imat.ElementaryDivisorsOptions.isom then
        tmp:=g.coeff1*left[pos1]+g.coeff2*left[pos2];
        left[pos2]:=g.coeff3*left[pos1]+g.coeff4*left[pos2];
        left[pos1]:=tmp;
      fi;
    else
      for row in [1..Length(mat)] do
        tmp:=g.coeff1*mat[row][pos1]+g.coeff2*mat[row][pos2];
        mat[row][pos2]:=
          g.coeff3*mat[row][pos1]+g.coeff4*mat[row][pos2];
        mat[row][pos1]:=tmp;
      od;
      if imat.ElementaryDivisorsOptions.isom then
        for row in [1..imat.cols] do
          tmp:=g.coeff1*right[row][pos1]+g.coeff2*right[row][pos2];
          right[row][pos2]:=
            g.coeff3*right[row][pos1]+g.coeff4*right[row][pos2];
          right[row][pos1]:=tmp;
        od;
      fi;
    fi;
  fi;
end;
##

#############################################################################
#F  IsolateIMR(imat,agenda) . . . isolate a pivot using an agenda
##  Performs the isolation of a pivot as previewed in agenda. A better
##    elimanation of type 1 to 3 is tried to improve the previewed damage.
IsolateIMR:=function(imat,agenda)
  local mat,row,rows,col,cols,el,
    i,j,minj,min,damage,
    index,index1,d1,d2,
    pnorm,pivot,isolist,len;
```

```
mat:=imat.amat;
rows:=Length(mat);
cols:=Length(mat[1]);
isolist:=agenda.isolist;
len:=Length(isolist);


if agenda.row<>rows then
  SwapTrafoIMR(imat,agenda.row,rows,true);
fi;
if agenda.col<>cols then
  SwapTrafoIMR(imat,agenda.col,cols,false);
fi;
pivot:=mat[rows][cols];
InfoIMat4("\n#I   Isolate:\c");


if agenda.rowfirst then

  # swap indeces in isolist
  for i in [1..len] do
    index:=isolist[i].index;
    if index=cols then
      isolist[i].index:=agenda.col;
    fi;
  od;
  pnorm:=imat.colnorm[agenda.col];
  imat.colnorm[agenda.col]:=imat.colnorm[cols];
  imat.colnorm[cols]:=pnorm;

  for i in [1..len] do

    index:=isolist[i].index;
    if isolist[i].gcd then
      GcdTrafoIMR(imat,cols,index,pivot,mat[rows][index],false);
      pivot:=mat[rows][cols];
      pnorm:=0;
      for col in [1..cols] do
        el:=mat[rows][col]; if el<0 then el:=-el; fi;
        pnorm:=pnorm+el;
      od;
    else

      # search best reducing col
      d2:=mat[rows][index];
      min:=[];
      for j in [i+1..len] do
        index1:=isolist[j].index;
        d1:=mat[rows][index1];
        if d2 mod d1=0 then
          if d1<0 then d1:=-d1; fi;
          damage:=QuoInt(imat.colnorm[index1],d1);
          if damage<min then min:=damage; minj:=j; fi;
        fi;
      od;
      if min=[] or min>=QuoInt(pnorm,AbsInt(pivot)) then
        # do what was forseen in agenda
        SubTrafoIMR(imat,cols,index,QuoInt(d2,pivot),false);
        InfoIMat4("=\c");
      else
        # something better has been found
```

```
        index1:=isolist[minj].index;
        d1:=mat[rows][index1];
        SubTrafoIMR(imat,index1,index,QuoInt(d2,d1),false);
        InfoIMat4("+\c");
      fi;
    fi;
  od;
  for row in [1..rows-1] do
    if mat[row][cols]<>0 then
      SubTrafoIMR(imat,rows,row,QuoInt(mat[row][cols],pivot),true);
    fi;
  od;
else

  # swap indeces in isolist
  for i in [1..len] do
    index:=isolist[i].index;
    if index=rows then
      isolist[i].index:=agenda.row;
    fi;
  od;
  pnorm:=imat.rownorm[agenda.row];
  imat.rownorm[agenda.row]:=imat.rownorm[rows];
  imat.rownorm[rows]:=pnorm;

  for i in [1..len] do

    index:=isolist[i].index;
    if isolist[i].gcd then
      GcdTrafoIMR(imat,rows,index,pivot,mat[index][cols],true);
      pivot:=mat[rows][cols];
      pnorm:=0;
      for row in [1..rows] do
        el:=mat[row][cols]; if el<0 then el:=-el; fi;
        pnorm:=pnorm+el;
      od;
    else

      # search best reducing row
      d2:=mat[index][cols];
      min:=[];
      for j in [i+1..len] do
        index1:=isolist[j].index;
        d1:=mat[index1][cols];
        if d2 mod d1=0 then
          if d1<0 then d1:=-d1; fi;
          damage:=QuoInt(imat.rownorm[index1],d1);
          if damage<min then min:=damage; minj:=j; fi;
        fi;
      od;
      if min=[] or min>=QuoInt(pnorm,AbsInt(pivot)) then
        # do what was forseen in agenda
        SubTrafoIMR(imat,rows,index,QuoInt(d2,pivot),true);
        InfoIMat4("=\c");
      else
        # something better has been found
        index1:=isolist[minj].index;
        d1:=mat[index1][cols];
        SubTrafoIMR(imat,index1,index,QuoInt(d2,d1),true);
        InfoIMat4("+\c");
      fi;
```

```
      fi;
    od;
    for col in [1..cols-1] do
      if mat[rows][col]<>0 then
        SubTrafoIMR(imat,cols,col,QuoInt(mat[rows][col],pivot),false);
      fi;
    od;

  fi;
  InfoIMat4(" ");
end;

#############################################################################
#F  TryIsolateIMR( <imat> ) . . . isolate a pivot without an agenda
##  A step of the best remainder strategy is performed. If it leads to
##    an actual isolation true is returned, otherwise false.
TryIsolateIMR := function ( imat )
  local mat,rows,cols,prod,clear,
    row, col,         # row and column loop variables
    prow, pcol,       # row and column index of minimal element
    entry, pivot;     # entries of the matrix
  mat:=imat.amat;
  rows:=Length(mat);
  cols:=Length(mat[1]);

  # find the element with the smallest absolute value in the matrix
  pivot:=0;
  for row  in [1..rows]  do
    for col  in [1..cols]  do
      entry:=mat[row][col];
      if 0<entry and (pivot=0 or entry<pivot or entry=pivot and
          imat.rownorm[row]*imat.colnorm[col]<prod) then
        pivot:=entry;
        prow:=row;
        pcol:=col;
        prod:=imat.rownorm[row]*imat.colnorm[col];
      elif entry<0 and (pivot=0 or -entry<pivot or -entry=pivot and
          imat.rownorm[row]*imat.colnorm[col]<prod) then
        pivot:=-entry;
        prow:=row;
        pcol:=col;
        prod:=imat.rownorm[row]*imat.colnorm[col];
      fi;
    od;
  od;

  # if there is no nonzero entry we are done
  if pivot=0 then
    imat.done:=true;
    return false;
  else
    pivot:=mat[prow][pcol];
    InfoIMat3("\n#I Try to isolate m[",prow,"][",pcol,"]:=",pivot," \c");

    if prow<>rows then
      SwapTrafoIMR(imat,prow,rows,true);
    fi;
    if pcol<>cols then
      SwapTrafoIMR(imat,pcol,cols,false);
    fi;
    clear:=true;
```

```
    # try to clear the column
    for row in [1..rows-1] do
      entry:=mat[row][cols];
      if entry>0 then
        if pivot>0 then
          SubTrafoIMR(imat,rows,row,QuoInt(2*entry+pivot,2*pivot),true);
        elif pivot<0 then
          SubTrafoIMR(imat,rows,row,QuoInt(2*entry-pivot,2*pivot),true);
        fi;
      elif entry < 0  then
        if pivot>0 then
          SubTrafoIMR(imat,rows,row,QuoInt(2*entry-pivot,2*pivot),true);
        elif pivot<0 then
          SubTrafoIMR(imat,rows,row,QuoInt(2*entry+pivot,2*pivot),true);
        fi;
      fi;
      clear:=clear and mat[row][cols]=0;
    od;
    # Note that I always use the quotient which leaves a rest with minimal
    #   absolute value. I use this form of QuoInt to avoid the formation
    #   of a rational number which includes a gcd calculation.

    # try to clear the row
    for col in [1..cols-1] do
      entry:=mat[rows][col];
      if entry>0 then
        if pivot>0 then
          SubTrafoIMR(imat,cols,col,QuoInt(2*entry+pivot,2*pivot),false);
        elif pivot<0 then
          SubTrafoIMR(imat,cols,col,QuoInt(2*entry-pivot,2*pivot),false);
        fi;
      elif entry < 0  then
        if pivot>0 then
          SubTrafoIMR(imat,cols,col,QuoInt(2*entry-pivot,2*pivot),false);
        elif pivot<0 then
          SubTrafoIMR(imat,cols,col,QuoInt(2*entry+pivot,2*pivot),false);
        fi;
      fi;
      clear:=clear and mat[rows][col]=0;
    od;
  fi;
  return clear;
end;
#############################################################################
#F  RowColReduction(imat,limit,rowop) . . . limited reduction routine
##  A restricted vector reduction is performed until the spanned module
##    is minimal or the number of loops exeeded limit.
GaussianReduce:=function(imat,i1,i2,rowop)
  # computes vectors for the same module with minimized 2-norm
  local mat,
        len,     # length of both vectors
        n1,n2,   # 2-norms of the vectors
        n,       # Euklidian product of the vectors
        t,       # norm of the next step vector
        r,i,changed;
  mat:=imat.amat;
  changed:=false;

  if rowop then
    len:=Length(mat[1]);
```

```
    n1:=Sum([1..len],i->mat[i1][i]^2);
    if n1=0 then return false; fi;
    n2:=Sum([1..len],i->mat[i2][i]^2);
    if n2=0 then return false; fi;
    n:=Sum([1..len],i->mat[i1][i]*mat[i2][i]);
  else
    len:=Length(mat);
    n1:=Sum([1..len],i->mat[i][i1]^2);
    if n1=0 then return false; fi;
    n2:=Sum([1..len],i->mat[i][i2]^2);
    if n2=0 then return false; fi;
    n:=Sum([1..len],i->mat[i][i1]*mat[i][i2]);
  fi;
  if n1<n2 then SwapTrafoIMR(imat,i1,i2,rowop);
                t:=n1;n1:=n2;n2:=t;
  fi;
  if n>0 then r:=QuoInt(2*n+n2,2*n2);
         else r:=QuoInt(2*n-n2,2*n2); fi;
  t:=n1-2*r*n+r^2*n2;

  while t>0 and t<n2 do
    changed:=true;
    SubTrafoIMR(imat,i2,i1,r,rowop);
    SwapTrafoIMR(imat,i1,i2,rowop);
    n1:=n2;
    n2:=t;
    if rowop then
      n:=Sum([1..len],i->mat[i1][i]*mat[i2][i]);
    else
      n:=Sum([1..len],i->mat[i][i1]*mat[i][i2]);
    fi;
    if n>0 then r:=QuoInt(2*n+n2,2*n2);
           else r:=QuoInt(2*n-n2,2*n2); fi;
    t:=n1-2*r*n+r^2*n2;
  od;

  if t<n1 then
    SubTrafoIMR(imat,i2,i1,r,rowop);
  elif not changed then
    return false;
  fi;
  return true;
end;

RowColReduction:=function(imat,limit,rowop)
  local mat,rows,pos1,pos2,len,changed,count;
  if rowop then
    len:=Length(imat.amat);
    InfoIMat2("\n#I Row reduction \c");
  else
    len:=Length(imat.amat[1]);
    InfoIMat2("\n#I Column reduction \c");
  fi;
  count:=0;

  repeat
    changed:=false;
    count:=count+1;
    for pos1 in [1..len-1] do
      for pos2 in [pos1+1..len] do
        changed:=changed or GaussianReduce(imat,pos1,pos2,rowop);
```

```
      od;
    od;
    InfoIMat4(".\c");
  until not changed or count>limit;
  InfoIMat2(" ");
end;


#############################################################################
#F  InitIMR(imat) . . . initialize for non-modular method
##  This procedure is to be called to initialize the working matrix for
##    the non-modular method and to begin with identity matrices as
##    transformation matrices.
InitIMR:=function(imat)
  imat.amat:=Copy(imat.matrix);
  imat.step:=1;
  imat.age:=0;
  imat.div:=1;
  if IsBound(imat.matnorm) then Unbind(imat.matnorm); fi;
  imat.done:=false;
  imat.diags:=[];
  if imat.ElementaryDivisorsOptions.isom then
    imat.left:=IdentityMat(Length(imat.amat));
    imat.right:=IdentityMat(Length(imat.amat[1]));
  fi;
end;

#############################################################################
#F  ClearIMR(imat) . . . clear (unusable) working data structures
##  Clear the working data structures.
ClearIMR:=function(imat)
  Unbind(imat.amat);
  Unbind(imat.step);
  Unbind(imat.div);
  if IsBound(imat.matnorm) then Unbind(imat.matnorm); fi;
  if IsBound(imat.rownorm) then Unbind(imat.rownorm); fi;
  if IsBound(imat.colnorm) then Unbind(imat.colnorm); fi;
  if IsBound(imat.rowdiv) then Unbind(imat.rowdiv); fi;
  if IsBound(imat.coldiv) then Unbind(imat.coldiv); fi;
  if IsBound(imat.rowsort) then Unbind(imat.rowsort); fi;
  if IsBound(imat.colsort) then Unbind(imat.colsort); fi;
  if IsBound(imat.oldmat) then Unbind(imat.oldmat); fi;
  if IsBound(imat.oldleft) then Unbind(imat.oldleft); fi;
  if IsBound(imat.oldright) then Unbind(imat.oldright); fi;
  if IsBound(imat.oldstep) then Unbind(imat.oldstep); fi;
  if IsBound(imat.olddiags) then Unbind(imat.olddiags); fi;
  if IsBound(imat.olddiv) then Unbind(imat.olddiv); fi;
  if IsBound(imat.diags) then Unbind(imat.diags); fi;
  Unbind(imat.done);
end;

#############################################################################
#F  ShrinkIMR(imat) . . remove row and columnn of an isolated pivot
##  This is to remove an isolated pivot with ist row and column from the
##    working matrix. The isolated element must be remembered as a
##    diagonal element. It is made positive at this point, including
##    changes to the transformation matrix if applicable.
##  If the matrix is shrinked to nothing, IMR field 'done' is set.
ShrinkIMR:=function(imat)
  local pos,mat,rows,cols;
  mat:=imat.amat;
```

```
  rows:=Length(mat);
  cols:=Length(mat[1]);

  # redundant consistency check

# for pos in [1..cols-1] do if mat[rows][pos]<>0 then
#   Error("IMR bugged!!"); fi; od;
# for pos in [1..rows-1] do if mat[pos][cols]<>0 then
#   Error("IMR bugged!!"); fi; od;

  if mat[rows][cols]<0 then
    Add(imat.diags,-imat.div*mat[rows][cols]);
    if imat.ElementaryDivisorsOptions.isom then
      imat.left[rows]:=-imat.left[rows];
    fi;
  else
    Add(imat.diags,imat.div*mat[rows][cols]);
  fi;
  imat.step:=imat.step+1;

  if rows=1 or cols=1 then
    mat:=[];
    imat.done:=true;
  else
    Unbind(mat[rows]);
    for pos in [1..rows-1] do Unbind(mat[pos][cols]); od;
  fi;
end;
##

#############################################################################
#F  ShuffleIMR(imat) . . . shuffle operation
##  If no acceptable pivot is found the matrix is reshuffled.
##  This procedure controls the shuffle possibilities which are
##    row/column reduction and a best remainder step.
ShuffleIMR:=function(imat)
  local shuffles,redurate;
  shuffles:=imat.ElementaryDivisorsOptions.shuffles;
  redurate:=imat.ElementaryDivisorsOptions.redurate;

  if imat.shufflecount<shuffles[1] then
    RowColReduction(imat,Int(redurate*Length(imat.amat)),true);
  elif imat.shufflecount<shuffles[2] then
    RowColReduction(imat,Int(redurate*Length(imat.amat[1])),false);
  else
    if TryIsolateIMR(imat) then
      InfoIMat3("#I ",Ordinal(imat.step)," element ");
      InfoIMat2("i\c");
      InfoIMat3("solated \c");
      ShrinkIMR(imat);
      imat.shufflecount:=0;
      return;
    fi;
  fi;
  imat.shufflecount:=imat.shufflecount+1;
end;
#############################################################################
#F  ResActualiseNorms(imat)
##  ActualiseNorms restricted for the use of TryIsolateIMR which does
##    not require vector divisors and sorted indeces.
ResActualiseNorms:=function(imat)
```

```
  local row,rows,col,cols,
    rownorm,colnorm,matnorm,
    entry,max,diff,quot,spars;
  rows:=Length(imat.amat);
  cols:=Length(imat.amat[1]);
  rownorm:=0*[1..rows];
  colnorm:=0*[1..cols];
  matnorm:=0;
  max:=0;
  spars:=0;

  for row in [1..rows] do
    for col in [1..cols] do
      entry:=imat.amat[row][col];
      if entry<0 then entry:=-entry; fi;
      matnorm:=matnorm+entry;
      if entry>max then max:=entry; fi;
      if entry=0 then spars:=spars+1; fi;
      rownorm[row]:=rownorm[row]+entry;
      colnorm[col]:=colnorm[col]+entry;
    od;
  od;
  if IsBound(imat.matnorm) then
    diff:=matnorm-imat.matnorm;
    InfoIMat3("\n#I  realised damage:  ",diff," (",
      QuoInt(diff*100,imat.matnorm+1),"%) ");
  fi;

  imat.rownorm:=rownorm;
  imat.colnorm:=colnorm;
  InfoIMat3("\n#I   maximal entry:   ",max,", matrix norm: ",matnorm,
    ", sparsity: ",QuoInt(spars*100,rows*cols),"% \c");
  imat.matnorm:=matnorm;
end;

#############################################################################
#F  BRemIMR(imat) . . diagonalize a matrix with best remainder strategy
##  The best remainder strategy main program.
BRemIMR:=function(imat)
  repeat
    if TryIsolateIMR(imat) then
      InfoIMat3("\n#I ",Ordinal(imat.step)," element ");
      InfoIMat2("i\c");
      InfoIMat3("solated \c");
      ShrinkIMR(imat);
    fi;
    if not imat.done then ResActualiseNorms(imat); fi;
  until imat.done;
  return imat.diags;
end;
#############################################################################
#F  TemporaryStoreIMR(imat) . . . prepare for backtracking
##  Stores the necessary information for the backtracking.
TemporaryStoreIMR:=function(imat)
  imat.oldmat:=Copy(imat.amat);
  if imat.ElementaryDivisorsOptions.isom then
    imat.oldleft:=Copy(imat.left);
    imat.oldright:=Copy(imat.right);
  fi;
  imat.olddiags:=Copy(imat.diags);
  imat.oldstep:=imat.step;
```

```
  imat.olddiv:=imat.div;
end;
###############################################################################
#F  BacktrackIMR(imat) . . . invoke backtracking
##  The copy of the working matrix and some more information is restored.
BacktrackIMR:=function(imat)
  if IsBound(imat.oldmat) then
    InfoIMat2("\n#I Taking the last step back \c");
    imat.amat:=imat.oldmat;
    if imat.ElementaryDivisorsOptions.isom then
      imat.left:=imat.oldleft;
      imat.right:=imat.oldright;
    fi;
    imat.diags:=imat.olddiags;
    imat.step:=imat.oldstep;
    imat.div:=imat.olddiv;
  fi;
end;
###############################################################################
#F  ActualiseNorms(imat)
##  Computes norms and vector divisors, sorts the vectors and gives
##    information about the matrix.
ActualiseNorms:=function(imat)
  local row,rows,col,cols,
    rownorm,colnorm,matnorm,rowdiv,coldiv,rowsort,colsort,
    entry,max,diff,quot,spars, avoidrate, intolerate, redurate;
  rows:=Length(imat.amat);
  cols:=Length(imat.amat[1]);
  avoidrate:=imat.ElementaryDivisorsOptions.avoidrate;
  intolerate:=imat.ElementaryDivisorsOptions.intolerate;
  redurate:=imat.ElementaryDivisorsOptions.redurate;

  rownorm:=0*[1..rows];
  colnorm:=0*[1..cols];
  rowdiv:=0*[1..rows];
  coldiv:=0*[1..cols];
  rowsort:=[1..rows];
  colsort:=[1..cols];
  matnorm:=0;
  max:=0;
  spars:=0;

  for row in [1..rows] do
    for col in [1..cols] do
      entry:=imat.amat[row][col];
      if entry<0 then entry:=-entry; fi;
      matnorm:=matnorm+entry;
      if entry>max then max:=entry; fi;
      if entry=0 then spars:=spars+1; fi;
      rownorm[row]:=rownorm[row]+entry;
      colnorm[col]:=colnorm[col]+entry;
      rowdiv[row]:=GcdInt(rowdiv[row],entry);
      coldiv[col]:=GcdInt(coldiv[col],entry);
    od;
  od;
  if IsBound(imat.matnorm) then
    diff:=matnorm-imat.matnorm;
    InfoIMat3("\n#I  realised damage:  ",diff," (",
      QuoInt(diff*100,imat.matnorm+1),"%) ");
    if diff>avoidrate*intolerate*imat.matnorm then
      BacktrackIMR(imat);
```

```
      imat.age:=imat.age+1;
      RowColReduction(imat,redurate*Length(imat.amat),true);
      ActualiseNorms(imat);
      return;
    fi;
  fi;
  SortParallel(Copy(rownorm),rowsort);
  while rownorm[rowsort[1]]=0 do rowsort:=rowsort{[2..Length(rowsort)]}; od;
  SortParallel(Copy(colnorm),colsort);
  while colnorm[colsort[1]]=0 do colsort:=colsort{[2..Length(colsort)]}; od;

  imat.rownorm:=rownorm;
  imat.colnorm:=colnorm;
  imat.rowdiv:=rowdiv;
  imat.coldiv:=coldiv;
  imat.rowsort:=rowsort;
  imat.colsort:=colsort;
  InfoIMat3("\n#I   maximal entry:   ",max,", matrix norm: ",matnorm,
    ", spars");
  InfoIMat2("i\c");
  InfoIMat3("ty: ",QuoInt(spars*100,rows*cols),"% \c");
  imat.matnorm:=matnorm;
end;

#############################################################################
#F  DivisorReductionIMR(imat) . . tries to find divisor of the whole matrix
##  If all entries are divisible by the same number, the whole matrix can
##    be divided by it.
DivisorReductionIMR:=function(imat)
  local el,div;

  div:=0;
  for el in imat.rowdiv do
    div:=GcdInt(div,el);
    if div=1 then return; fi;
  od;
  for el in imat.coldiv do
    div:=GcdInt(div,el);
    if div=1 then return; fi;
  od;

  InfoIMat2("\n#I whole rest of matrix divided by ",div," ");
  imat.amat:=imat.amat/div;
  imat.rownorm:=imat.rownorm/div;
  imat.colnorm:=imat.colnorm/div;
  imat.rowdiv:=imat.rowdiv/div;
  imat.coldiv:=imat.coldiv/div;
  imat.matnorm:=imat.matnorm/div;
  imat.div:=imat.div*div;
end;

#############################################################################
#F  DiagIMR(imat) . . diagonalize matrix with preview method
##  Main procedure for the preview strategy. Note that the divisibility
##    property is not yet established.
DiagIMR:=function(imat)
  local agenda;

  repeat
    ActualiseNorms(imat);
    if imat.age=3 then
```

```
      InfoIMat2("\n#I Giving up Preview Strategy. ");
      BRemIMR(imat);
      return imat.diags;
    fi;
    DivisorReductionIMR(imat);
    imat.shufflecount:=0;
    TemporaryStoreIMR(imat);

    agenda:=PivotIMR(imat);
    while agenda=false do
      ShuffleIMR(imat);
      ActualiseNorms(imat);
      if imat.age=3 then
        InfoIMat2("\n#I Giving up Preview Strategy. ");
        BRemIMR(imat);
        return imat.diags;
      fi;
      DivisorReductionIMR(imat);
      TemporaryStoreIMR(imat);
      agenda:=PivotIMR(imat);
    od;

    IsolateIMR(imat,agenda);

    ShrinkIMR(imat);
    if ForAll(imat.amat,x->ForAll(x,y->y=0)) then imat.done:=true; fi;
  until imat.done;

  return imat.diags;
end;

#############################################################################
#F  ElDivIMR( <imat> ) . . elementary divisors non-modular method
##  Main procedure of the non-modular algorithm. After calling the
##    diagonalisation the divisibility property is established including
##    the adjustments to the transformation matrices if applicable.
ElDivIMR := function ( imat )
  local  a, b, divs, g,
    rows, cols, row, col, tmp,
    d1,d2,rpos1, rpos2,lpos1, lpos2,
    left, right, mult, tmp;

  InfoIMat1("#I  ElementaryDivisorsIMat(non-modular) called\n");

  if not IsBound(imat.amat) then InitIMR(imat);
  elif not IsBound(imat.left) and imat.ElementaryDivisorsOptions.isom then
    Print("Warning: Your IMR was not initialized to output diagonalisation",
          "\n   unsetting 'isom' parameter\n");
    imat.ElementaryDivisorsOptions.isom:=false;
  fi;
  InfoIMat2("#I  ");

  # diagonalize the matrix
  if imat.ElementaryDivisorsOptions.algorithm="preview" then
    divs:=DiagIMR( imat );
  elif imat.ElementaryDivisorsOptions.algorithm="bestrem" then
    ResActualiseNorms(imat);
    divs:=BRemIMR( imat );
  fi;
  rows:=Length(imat.matrix);
  cols:=Length(imat.matrix[1]);
```

```
# the matrix is now diagonalized, we
# transform the diagonal elements so that every one divides the next
for d1  in [1..Length(divs)-1]  do
  for d2  in [d1+1..Length(divs)]  do
    a:=divs[d1];b:=divs[d2];
    if a <> 0  and b mod a <> 0  then
      g     := Gcdex2(a,b);

      divs[d1] := g.gcd;
      divs[d2] := b * g.coeff4;

      if imat.ElementaryDivisorsOptions.isom then
        rpos1:=cols-d1+1;
        rpos2:=cols-d2+1;
        lpos1:=rows-d1+1;
        lpos2:=rows-d2+1;

        for row in [1..cols] do
          imat.right[row][rpos1]:=imat.right[row][rpos1]+imat.right[row][rpos2];
        od;

        tmp:=g.coeff1*imat.left[lpos1]+g.coeff2*imat.left[lpos2];
        imat.left[lpos2]:=g.coeff3*imat.left[lpos1]+g.coeff4*imat.left[lpos2];
        imat.left[lpos1]:=tmp;

        mult:=g.coeff3*g.coeff2;
        for row in [1..imat.cols] do
          imat.right[row][rpos2]:=imat.right[row][rpos2]+mult*imat.right[row][rpos1];
        od;
      fi;
    fi;
  od;
od;

# Our matrix is not quite in Smith normal form due to our shrinking technique:
#   the entry bottom right shoud bo the entry top left etc. We change that only if we
#   output the transformation matrices.
if imat.ElementaryDivisorsOptions.isom then
  for row in [1..QuoInt(rows,2)] do
    tmp:=imat.left[row];
    imat.left[row]:=imat.left[rows-row+1];
    imat.left[rows-row+1]:=tmp;
  od;
  for row in [1..cols] do
    for col in [1..QuoInt(cols,2)] do
      tmp:=imat.right[row][col];
      imat.right[row][col]:=imat.right[row][cols-col+1];
      imat.right[row][cols-col+1]:=tmp;
    od;
  od;
fi;

imat.eldivisors:=divs;
ClearIMR( imat );

InfoIMat2("\n");
InfoIMat1("#I  ElementaryDivisorsIMat(non-modular) returns\n");
if imat.ElementaryDivisorsOptions.isom then
  return rec(divisors:=divs,left:=imat.left,right:=imat.right);
else
```

```
    return divs;
  fi;
end;


###########################################################################
##
#H  Main procedures to be called by the user
##
###########################################################################
#F  ElementaryDivisorsIMat(<mat|imat>[,optionrec]) . elementary divisors
##  of an integer matrix. The details of the algorithm are determined by the
##    record fields of the IMR <imat>, the optionrec or the xxxDefaultOptions
##    records.
ElementaryDivisorsIMat:=function(arg)
  local imat,p,optchange, opts;
  if Length(arg)<2 then
    optchange:=rec();
  else
    optchange:=arg[2];
  fi;
  if not IsRec(arg[1]) then
    imat:=IMRIMat(arg[1],optchange);
    opts:=imat.ElementaryDivisorsOptions;
  else
    imat:=arg[1];
    opts:=imat.ElementaryDivisorsOptions;
    for p in RecFields(optchange) do
      if p="fastrank" then
        if optchange.(p)=false and
           imat.RankOptions.(p)=true and
           IsBound(imat.rank) and imat.rank<imat.cols
        then
          Unbind(imat.rank);
          if IsBound(imat.fivemat) then Unbind(imat.fivemat); fi;
          if IsBound(imat.fiveprimes) then Unbind(imat.fiveprimes); fi;
          if IsBound(imat.fivedets) then Unbind(imat.fivedets); fi;
          if IsBound(imat.det) then Unbind(imat.det); fi;
          if IsBound(imat.factors) then Unbind(imat.factors); fi;
          if IsBound(imat.noteldiv) then Unbind(imat.noteldiv); fi;
          if IsBound(imat.eldivconst) then Unbind(imat.eldivconst); fi;
          if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
        fi;
        imat.RankOptions.(p):=optchange.(p);
      elif p="fastdet" then
        if optchange.(p)=false and
           imat.DeterminantOptions.(p)=true
        then
          if IsBound(imat.det) then Unbind(imat.det); fi;
          if IsBound(imat.factors) then Unbind(imat.factors); fi;
          if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
        fi;
        imat.DeterminantOptions.(p):=optchange.(p);
      elif p="fasthadamard" then
        imat.DeterminantOptions.(p):=optchange.(p);
        imat.RankOptions.(p):=optchange.(p);
      else
        if p="maxfivers" and optchange.(p)>opts.maxfivers then
          if IsBound(imat.det) then Unbind(imat.det); fi;
          if IsBound(imat.factors) then Unbind(imat.factors); fi;
          if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
```

```
        fi;
        opts.(p):=optchange.(p);
      fi;
    od;
  fi;

  if IsBound(imat.eldivisors) then
    InfoIMat2("#I  Remembered Elementary Divisors:",imat.eldivisors,"\n");
    return imat.eldivisors;
  fi;

  if opts.algorithm="modular" then
    return ElDivModIMR(imat);
  elif opts.algorithm="preview" or opts.algorithm="bestrem" then
    return ElDivIMR(imat);
  else Error("Algorithm for elementary divisors calculation must be one of\n\
    \"modular\", \"bestrem\", or \"preview\"");
  fi;
end;

#############################################################################
#F  RankIMat(<mat|imat>[,optionrec]) . . rank of an integer matrix
##  computed by a modular method.
##  The details of the algorithm are determined by the
##    record fields of the IMR <imat>, the optionrec or
##    the RankDefaultOptions records.
RankIMat:=function(arg)
  local imat,p,optchange, opts, rank;

  InfoIMat1("#I  RankIMat called\n");
  if Length(arg)<2 then
    optchange:=rec();
  else
    optchange:=arg[2];
  fi;
  if not IsRec(arg[1]) then
    imat:=IMRIMat(arg[1],optchange);
    opts:=imat.RankOptions;
  else
    imat:=arg[1];
    opts:=imat.RankOptions;
    for p in RecFields(optchange) do
      if p="fastrank" then
        if optchange.(p)=false and
           imat.RankOptions.(p)=true and
           IsBound(imat.rank) and imat.rank<imat.cols
        then
          Unbind(imat.rank);
          if IsBound(imat.fivemat) then Unbind(imat.fivemat); fi;
          if IsBound(imat.fiveprimes) then Unbind(imat.fiveprimes); fi;
          if IsBound(imat.fivedets) then Unbind(imat.fivedets); fi;
          if IsBound(imat.det) then Unbind(imat.det); fi;
          if IsBound(imat.factors) then Unbind(imat.factors); fi;
          if IsBound(imat.noteldiv) then Unbind(imat.noteldiv); fi;
          if IsBound(imat.eldivconst) then Unbind(imat.eldivconst); fi;
          if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
        fi;
        opts.(p):=optchange.(p);
      elif p="fasthadamard" then
        opts.(p):=optchange.(p);
      else
```

```
        opts.(p):=optchange.(p);
      fi;
    od;
  fi;

  rank:=RankIMR(imat);
  InfoIMat1("#I  RankIMat returns\n");
  return rank;
end;

#############################################################################
#F  DeterminantIMat(<mat|imat>[,optionrec]) . determinant of a integer matrix
##  which must be square, computed by a modular method.
##    The details of the algorithm are determined by the
##    record fields of the IMR <imat>, the optionrec or the xxxDefaultOptions
##    records.
DeterminantIMat:=function(arg)
  local imat,p,optchange, opts, det;
  InfoIMat1("#I  DeterminantIMat called\n");
  if Length(arg)<2 then
    optchange:=rec();
  else
    optchange:=arg[2];
  fi;
  if not IsRec(arg[1]) then
    imat:=IMRIMat(arg[1],optchange);
    opts:=imat.DeterminantOptions;
  else
    imat:=arg[1];
    opts:=imat.DeterminantOptions;
    for p in RecFields(optchange) do
      if p="fastdet" then
        if optchange.(p)=false and
           imat.DeterminantOptions.(p)=true and
           IsBound(imat.det)
        then
          Unbind(imat.det);
          if IsBound(imat.factors) then Unbind(imat.factors); fi;
          if IsBound(imat.noteldiv) then Unbind(imat.noteldiv); fi;
          if IsBound(imat.eldivconst) then Unbind(imat.eldivconst); fi;
          if IsBound(imat.eldivisors) then Unbind(imat.eldivisors); fi;
        fi;
        opts.(p):=optchange.(p);
      elif p="fasthadamard" then
        opts.(p):=optchange.(p);
      else
        opts.(p):=optchange.(p);
      fi;
    od;
  fi;

  if imat.cols<imat.rows then Error("square matrix expected"); fi;
  imat.fivemat:=rec(rows:=[1..imat.cols-1],
                    cols:=[1..imat.cols],
                    lastrows:=[imat.cols] );
  imat.fivedets:=[];
  imat.fiveprimes:=[];
  det:=DetIMR(imat);
  InfoIMat1("#I  DeterminantIMat returns\n");
  return det;
end;
```

```
#E  Emacs . . . . . . . . . . . . . . . . . . . . . . . local emacs variables
##
##  Local Variables:
##  mode:               outline
##  outline-regexp:     "#H\\|#F\\|#V\\|#E\\|##F"
##  fill-column:        73
##  fill-prefix:        "##    "
##  eval:               (hide-body)
##
##  End:
```

Kapitel 7

Fazit

Der Preview-Algorithmus geht gemächlich seinen Weg, und ist trotzdem nicht vor Koeffizentenexplosion gefeit. Im Vergleich mit der eleganten Implementierung des KANNAN–BACHEM und des Best–Remainder, die beide auf ein, zwei Bildschirmseiten passen wirkt er wie ein aufgeblasener schwerfälliger Riese, vollgepropft mit lauter Schaltern und Hebeln.

Hoffnungslos im Zeitvergleich hinterherhinkend kann er nur darauf hoffen, die anderen beiden in der Koeffizentenexplosion versanden zu sehen. Es ruft ihnen hinterher, um wieviel weniger Matrixnorm es an einer Wegmarke hatte, nur Gelächter erntend. Doch nun beginnen die Gefilde der Koeffizentenexplosion, und plötzlich, KANNAN–BACHEM, am schnellsten gestartet, gibt als erstes auf, und rechnet mit <<an integer too large to be printed>>. Preview hofft, daß auch Best Remainder bald steckenbleibt, aber es rappelt sich immer wieder auf, die Matrixnorm verhundertfacht sich bald, dann hundertstelt sie sich wieder, hoch und runter, aber es geht noch weiter. Und Preview selbst wird immer schwerfälliger, die vielen ggT–Berechnungen, am Anfang noch ein Kinderspiel, machen schwer zu schaffen.

Und dann kommt der Punkt, wo Preview keine guten Ideen mehr für einfach isolierbare Pivot hat, es beginnt, die Matrix durchzumischen, um noch ein paar gute Abstiege hervorzuzaubern — das kostet Zeit und hat nicht allzu viel Erfolg. Irgendwann bleibt dann Preview stecken; nicht, weil die Rechenoperationen wegen der Koeffizientenexplosion zu lange dauern, sondern weil keine Abstiege mit tolerierbaren Schäden mehr existieren. Das kombinierte Programm aus Preview–Strategie und Best–Remainder kommt wirklich ein Stück weiter, als Best–Remainder von Beginn an — mit erheblich längerer Rechenzeit in den kritischen Phasen.

Was ist also überhaupt gekonnt? Zunächst muß gesagt werden, daß Preview ein Verfahren ist, um die Koeffizentenexplosion des KANNAN–BACHEM–Verfahrens abzumildern, und das tut es außerordentlich gut; bei einer ganzen Reihe von Beispielen ist es zu einem Ergebnis gekommen, wo die ursprünglich Variante versagte.

Es muß sich aber dem Vergleich mit Best–Remainder stellen — und es wird sich schwerlich ein Beispiel finden lassen, wo nur Preview zu einem Ergebnis kommt, und Best–Remainder an der Koeffizientenexplosion scheitert, so klein ist der Unterschied im Koeffizentenwachstum. Auch die Chancen, daß Preview schneller zu einem Ergebnis kommt, sind nicht allzu gut, wenn man sich verdeutlicht, daß Preview bei beginnender Koeffizientenexplosion z. B. achtmal mehr Rechenzeit benötigt.

Die Gründe dafür, daß ich versucht habe, das Preview–Verfahren auszufeilen, war nicht nur, daß möglicherweise ein überlegenes Programm entstehe, sondern auch

- um tieferes Verständnis des Wesens der Koeffizientenexplosion des GAUSS–Verfahrens zu erlangen. Intuitiv wurde bisher angenommen, daß die Koeffizientenexplosion nicht zu vermeiden ist. Das kann aber nur definitiv behauptet werden, wenn die Koeffizientenexplosion auch bei den bestmöglichen der exponentiell vielen Wahlentscheidungen eintritt. Und bei den bisherigen Verfahren beschränkte sich die Approximation der besten Möglichkeit des Isolierens auf die sehr grobe Heuristik des Auswählens eines einzigen Pivots mithilfe von Vektorennormen. Die Approximation des Preview–Verfahrens ist viel feiner, somit kann davon ausgegengen werden, daß die vom Preview–Verfahren ausgewählten Alternativen viel dichter an den optimalen liegen, wenn auch weniger mögliche Operationen in Betracht gezogen werden, als beim Best–Remainder–Verfahren.

- um Wege aufzuzeigen, die Informatik stärker mit der Mathematik zu verknüpfen. Mein implementiertes Backtracking ist z. B. ein rein informatisches Verfahren — ein einfacher evolutionärer Algorithmus. Aber auch die Preview–Grundidee ist eher mit den Trial–and–Error–Methoden der Informatik verwandt als mit den strengen Methoden der Computeralgebra.

- Vielleicht kann die Preview–Idee noch ausgebaut werden. Schließlich erreicht das Preview–Verfahren die Begrenzung der Koeffizientenexplosion auf einem ganz anderen Weg als das Best–Remainder–Verfahren. Ich schließe natürlich auch nicht aus, daß der Preview–Algorithmus, so wie ich ihn hier beschrieben habe, noch Denkfehler enthält, deren Ausmerzung das Verfahren entscheidend verbessern könnte.

Literaturverzeichnis

[1]

```
                 ########           Lehrstuhl D fuer Mathematik
               ###    ####          RWTH Aachen
              ##         ##
             ##          #            #######            #########
            ##                       #      ##          ## #     ##
            ##           #           #       ##            #      ##
            ####        ##           ##       #            #      ##
             #####     ###           ##      ##            ##    ##
               ######### #            #########            #######
                         #                                 #
                        ##          Version 3              #
                       ###          Release 3              #
                      ## #          09 Nov 93              #
                     ##  #
                    ##   #  Alice Niemeyer, Werner Nickel,  Martin Schoenert
                   ##    #  Johannes Meier, Alex Wegner,    Thomas Bischops
                  ##     #  Frank Celler,   Juergen Mnich,  Udo Polis
                  ###   ##  Thomas Breuer,  Goetz Pfeiffer, Hans U. Besche
                   ######   Volkmar Felsch, Heiko Theissen, Alexander Hulpke
                            Ansgar Kaup,    Akos Seress
```

[2]

```
    |\^/|     Maple V Release 2 (Rhein-Westf Technische Hochschule)
._|\|   |/|_. Copyright (c) 1981-1993 by the University of Waterloo.
 \  MAPLE  /  All rights reserved. Maple and Maple V are registered
 <____ ____>  trademarks of Waterloo Maple Software.
      |
```

[3]

```
integer matrix diagonalisation program v2.0
c copyright
c
c   this program drives a collection of subroutines which compute
c   a canonical form for an integer matrix.
c   descriptions of the program and results obtained with it appear in
c   g.havas and l.s.sterling, 'integer matrices and abelian groups',
c   proc. 1979 eurosam conference
```

[4] George Havas and Leon S. Sterling. Integer Matrices and abelian groups, Symbolic and Algebraic Computation, Lecture Notes in Computer Science 72, Springer-Verlag, 1979, pp. 431-451

[5] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. Linear Algebra and its Applications, Vol. 192 (1993), pp. 137-163.

[6] J. L. Hafner and K. S. McCurley. Asymptotically fast triangularization of matrices over rings. SIAM J. Comput. 20 (1991), pp. 1068-1083.

[7] R. Kannan and R. Bachem. Polynomial Time algorithms for computing Smith and Hermite normal forms of an integer matrix. SIAM Journal for Computing 8 (1979), pp. 499-507.

[8] Henri Cohen. A Course in Computational Algebraic Number Theory. Graduate Texts in Mathematics 138. Springer-Verlag, 1993

[9] S. Lang, Algebra, Addison-Wesley (2nd edition 1984)

[10] D. J. Rose and R. E. Tarjan. Algorithmis aspects of vertex elimination directed graphs. SIAM Journal for Applied Mathematics 34 (1978). S. 176-197.