Universität Kaiserslautern
Fachbereich Mathematik

Der FGLM-Algorithmus: verallgemeinert und implementiert in SINGULAR

Diplomarbeit von
Tim Wichmann
November 1997

Inhaltsverzeichnis

Einleitung

Die vorliegende Arbeit beschäftigt sich mit dem FGLM-Algorithmus. Neben der Behandlung der Theorie zu dieser Methode war das Ziel der Arbeit, eine effiziente Implementierung für das Computeralgebrasystem SINGULAR zu erstellen. In anschließenden Versuchen sollte die Nutzbarkeit dieser Methode sowie die Qualität der Implementierung untersucht werden.

Eines der wohl bedeutendsten Ereignisse der Computeralgebra ist die Entdeckung des sogenannten Buchberger-Algorithmus, der die Berechnung einer Standardbasis erlaubt. Es ist erstaunlich, dass dieser Algorithmus erst 1965 gefunden wurde (B. Buchberger [5]), obwohl der Begriff der Standardbasis schon seit langem bekannt war. Seit seiner Entdeckung hält der Siegeszug des Buchberger-Algorithmus unvermindert an, und er ist heutzutage einer der zentralen Algorithmen der meisten Computeralgebrasysteme. Es hat zahlreiche Verbessrungen gegeben, die seine Leistungsfähigkeit steigen ließen, doch auch mit modernen, schnellen Rechnern kann die Berechnung einer Standardbasis mit Hilfe des Buchberger-Algorithmus sehr aufwendig, wenn nicht unmöglich werden.

Der FGLM-Algorithmus:

Im Jahre 1989 schlugen J. C. Faugère, P. Gianni, D. Lazard und T. Mora in ihrem Paper *Efficient computation of zero-dimensional Gröbner bases by change of ordering* [11] eine alternative Möglichkeit zur Berechnung einer Standardbasis vor. Der Algorithmus hat den Namen **FGLM-Algorithmus** bekommen, entsprechend den Initialien ihrer Entdecker. Sie nutzten die Tatsache aus, dass der Buchberger-Algorithmus sehr empfindlich auf verschiedene Ordnungen reagiert: Die Berechnungszeiten können drastische Differenzen aufweisen, arbeitet man bezüglich zweier unterschiedlicher Ordnungen. Im Allgemeinen kann man einem Erzeugendensystem a priori nicht ansehen, ob und bezüglich welcher Ordnung sich die Berechnung einfach oder aufwendig gestaltet. Die Idee von Faugère et al. besteht darin, eine Basis nicht wie der Buchberger-Algorithmus auf direktem Wege zu erstellen, sondern vielmehr eine Gröbnerbasen*transformation* vorzunehmen: Man berechnet eine Basis bezüglich einer „Ausweichsordnung", für die sich die Berechnung einfach erweist, und transformiert diese dann in eine Gröbnerbasis bezüglich der gewünschten Ordnung. Dabei ist von Vorteil, dass man zur Transformation nicht ein beliebiges Erzeugendensystem, sondern gerade eine Standardbasis zur Verfügung hat, deren Strukturen und Eigenschaften man gut kennt. Diese Informationen können in den Algorithmus mit einfließen; so kann z.B. das Erstellen von Normalformen gänzlich umgangen werden. Leider genießt der von Faugère et al. verfolgte Ansatz nicht die volle Allgemeinheit: Man kann ihn nur auf die Klasse der nulldimensionalen Ideale anwenden.

In ihrer Veröffentlichung zeigten Faugère et al. mit der Implementierung ihres Algorithmus, dass dieser Weg durchaus eine kostengünstige Alternative zum Buchberger-Algorithmus sein kann. Das galt es im Rahmen dieser Arbeit zu bestätigen. Da in SINGULAR einer der schnellsten Buchberger-Algorithmen implementiert ist, war eine faire Umgebung gegeben, in der die beiden Methoden miteinander verglichen werden konnten.

In jüngster Zeit sind neue Algorithmen zur Gröbnerbasen-Transformation entstanden, so z.B. der sogenannte *Gröbner-Walk-Algorithmus* (S. Collart et al. [7]). Solche Algorithmen gilt es in Zukunft zu implementieren und auf ihre Effizienz zu überprüfen.

verallgemeinert ...

Die Einschränkung des FGLM-Algorithmus auf nulldimensionale Ideale stört ein wenig, und es hat Ansätze gegeben, diese aufzulockern (z.B. M. Niermann [15]). Im Rahmen der vorliegenden Arbeit wurden die FGLM-Methoden unter Zuhilfenahme der Hilbertfunktion auf die Klasse der homogenen Ideale erweitert. Der daraus resultierende Algorithmus zeigt leider in der Praxis eine geringere Effizienz als der Algorithmus für nulldimensionale Ideale. Es steckt aber sicher noch ein gewisses Potential an Verbesserungsmöglichkeiten in ihm, die es in Zukunft noch zu entdecken gilt.

Der FGLM-Algorithmus arbeitet mit dem Begriff der linearen Abhängigkeit modulo eines Ideals. Diese Sichtweise führte im Laufe dieser Arbeit noch zu zwei kleinen, netten Algorithmen, die sich als nützlich bei der Primärzerlegung und der Elimination erweisen.

und implementiert ...

Nicht zuletzt aus der praktischen Erfahrung mit Beispielen, die sich mit dem Buchberger-Algorithmus nur noch schwer berechnen lassen, kam der Wunsch nach einer Ausweichmöglichkeit auf. Daher wurde eine Implementierung für SINGULAR angestrebt.

Von Beginn an lautete das Ziel, eine effiziente Implementierung zu erreichen. Die Verwendung der SINGULAR-eigenen Interpretersprache schied damit von vorne herein aus, stattdessen wurde der Algorithmus in C/C++ geschrieben und direkt in den Kern von SINGULAR eingebaut.

in SINGULAR.

SINGULAR ist ein Computeralgebrasystem, das speziell für die Anforderungen der kommutativen Algebra, der algebraischen Geometrie und der Singularitätentheorie entwickelt worden ist. Die wichtigsten Objekte, mit denen SINGULAR arbeitet, sind Polynome, Ringe und Moduln, es gibt eine interaktive Shell sowie eine eigene, C-ähnliche Interpretersprache. SINGULAR bietet eine der schnellsten und allgemeinsten Implementierungen zur Gröbnerbasen-Berechnung, sowie Algorithmen zur polynomialen Faktorisierung, Resultanten, charakteristische Mengen, Syzygien-Berechnungen und vieles mehr. Außerdem besitzt SINGULAR eine eigene Homepage, diese findet sich unter

`http://www.mathematik.uni-kl.de/~zca/Singular` .

Dort steht SINGULAR auch als ausführbares Programm für verschiedene Plattformen zur Verfügung.

Zu dieser Arbeit

Die vorliegende Arbeit gliedert sich in drei große Abschnitte: In den Kapiteln 1 bis 4 wird der theoretische Hintergrund zusammengetragen, der zum Verständnis der Algorithmen notwendig ist. Diese werden in den Kapiteln 5 und 6 hergeleitet, formuliert und bewiesen. In den verbleibenden beiden Kapiteln 7 und 8 geht es dann in die Praxis: Hier finden sich die Zeitmessungen und der Quelltext der Implementierung in SINGULAR.

In dieser Arbeit werden die folgenden Notationen verwendet:

- Das zentrale Objekt ist der Polynomring über einem Körper K in mehreren Veränderlichen. Dieser wird mit $K[x_1,\ldots,x_n]$ bezeichnet, so dass $x_1,\ldots,x_n$ die Unbestimmten des Polynomringes sind. Aus typografischen Gründen wird in dieser Arbeit der Polynomring alternativ auch mit P_n bezeichnet, hier tauchen die Unbestimmten in der Notation nicht mehr auf.

- Wie üblich bezeichnen $\mathbb{Q}$, $\mathbb{R}$ bzw. $\mathbb{C}$ den Körper der rationalen, der rellen bzw. der komplexen Zahlen. Die ganzen Zahlen *ohne* Null werden als $\mathbb{N}$, die ganzen Zahlen *inklusive* der Null werden als ω notiert. Ist K ein Körper, so ist mit K^* die Menge $K \setminus \{0\}$ gemeint.

- Ist R ein Ring, $I \subseteq R$ ein Ideal und $X \in R/I$ eine Restklasse des Quotientenringes, so wird diese mit $X = r + I$ bezeichnet, wobei $r \in R$.

- Ist $v \in K^d$ ein Vektor eines Vektorraumes, so wird mit $v^{(i)}$ die i-te Komponente von v bezeichnet. Ist $M \in \mathrm{Mat}(d \times d, K)$ eine $d \times d$-Matrix über K, so bezeichnet $M^{(i)}$ die i-te Spalte von M.

- In dieser Arbeit wird an keiner Stelle die Differentiation benötigt, so dass der Strich $'$ immer nur typografische Bedeutung hat.

- Die Bezeichner werden, soweit dies möglich ist, wie folgt gewählt: Ideale heißen I, Monome m oder b, ihre Exponentenvektoren sind mit α, β oder γ benannt. Polynome heißen p oder q; handelt es sich um spezielle Polynome, etwa solche aus einer Gröbnerbasis, so werden diese meist f oder g genannt. Standardbasen tragen die Bezeichnung F oder G. Elemente eines Vektorraumes bekommen die Bezeichnung v, die Elemente der kanonischen Basis in diesem Vektorraum heißen wie üblich e_i. Die Hilbertfunktion schließlich trägt den Namen h.

- Die Algorithmen sind in einer Pseudosprache beschrieben. Diese arbeitet mit den üblichen Kontrollstrukturen wie **if**-Abfragen, **for**-, **while**- und **repeat**-Schleifen. Zuweisungen werden durch einen nach links gerichteten Pfeil angegeben. Der Gültigkeitsbereich eines Blockes, etwa einer **for**-Schleife, ist allein durch die Einrückung gekennzeichnet, im folgenden Beispiel z.B. die Zeilen 4 bis 6 für die äußere Schleife.

$(q_1,\ldots,q_n) = \textbf{beispiel}(f,g)$

1 $\quad p \leftarrow f \cdot g$
2 $\quad \textbf{let } p = \lambda_1 m_1 + \cdots + \lambda_k m_k \ , \textbf{ where } \lambda_i \in K, m_i \in M_n$
3 $\quad \textbf{for } (i = 1,\ldots,n)$
4 $\qquad \textbf{for } (j = 1,\ldots,k)$
5 $\qquad\quad \textbf{if } (x_i \mid m_k)$
6 $\qquad\qquad q_i \leftarrow q_i + m_j$
$\qquad\quad \bullet\ (1) \quad x_i$ teilt jedes Monom von q_i
7 $\quad \textbf{return } (q_1,\ldots,q_n)\ \bullet$

Die Algorithmen werden anhand von Invarianten bewiesen, die in verkleinerter Schrift erscheinen und durch einen vorangestellten Punkt gekennzeichnet sind. Solche Zeilen erhalten keine Zeilennummer. Ein besonderes Konstrukt ist die **let**-Anweisung, die nur Bezeichnungen ineinander überführt. Im Beispiel heißt sie etwa, dass die Monome des Polynoms p fortan mit $m_1,\ldots,m_k$ bezeichnet seien. Die Parameter und Ausgaben der Algorithmen sind jeweils zu Beginn genau spezifiziert.

Es war vielleicht etwas übereilt, aber die vorliegende Arbeit ist nach den Regeln der neuen Rechtschreibung geschrieben.

Ordnungen im Polynomring

Bevor man mit der Arbeit beginnt, sollte man zunächst einmal sein Handwerkszeug bereitlegen. Genau das geschieht in diesem Kapitel. Es werden die grundlegenden Begriffe wie etwa *Term*, *Wohlordnung* oder *Leitmonom* erklärt, die für den weiteren Verlauf unentbehrlich sind. Ausserdem erfährt man, was ein *Newton-Diagramm* ist, denn dieses erleichtert die Anschauung erheblich.

Definition. *Sei $t \in K[x_1, \ldots, x_n]$. Existieren $\alpha_1, \ldots, \alpha_n \in \omega$ und ein $c \in K$ mit*

$$t = c \cdot x_1^{\alpha_1} \cdots x_n^{\alpha_n} \ ,$$

so heißt t ein ***Term*** *und c der* ***Koeffizient*** *des Terms. Ein Term mit Koeffizient 1 heißt* ***Monom****.*

Monome in $K[x_1, \ldots, x_n]$ haben also die Gestalt $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ mit $\alpha_i \in \omega$. Man vereinbart für sie die abkürzende Schreibweise

$$x^\alpha := x_1^{\alpha_1} \cdots x_n^{\alpha_n} \quad \text{mit} \quad \alpha = (\alpha_1, \ldots, \alpha_n) \in \omega^n.$$

Die Monome sind von so großer Bedeutung, dass sie eine eigene Bezeichnung verdienen:

Notation. *Mit M_n sei die Menge aller Monome des Polynomrings P_n in n Variablen bezeichnet, d.h. es ist*

$$M_n = \{\, x^\alpha \mid \alpha \in \omega^n \,\} \subset P_n \ .$$

Ein Polynom $p \in P_n$ lässt sich dann immer eindeutig als Summe seiner Terme schreiben, d.h. es existieren ein eindeutig bestimmtes, endliches $\Phi \subset \omega^n$ und eindeutig bestimmte $c_\alpha \in K^*$ für alle $\alpha \in \Phi$, so dass gilt

$$p = \sum_{\alpha \in \Phi} c_\alpha x^\alpha \ .$$

Ist ein Polynom notiert als $p = \sum c_\alpha x^\alpha$, so soll fortan implizit immer davon ausgegangen werden, dass $c_\alpha \neq 0$ gilt, sofern nicht anders gesagt.

Die Menge der Monome ist offensichtlich bezüglich der Multiplikation eine Halbgruppe mit neutralem Element $1 = x^{(0,\ldots,0)} \in M_n \subset P_n$. Da die Multiplikation zweier Monome gerade der Addition ihrer Exponentenvektoren entspricht, ist diese Halbgruppe über

$$\sigma : \omega^n \to M_n \quad \text{mit} \quad \alpha \mapsto x^\alpha$$

isomorph zur Halbgruppe ω^n, versehen mit der elementweisen Addition. Diese Isomorphie bringt den großen Vorteil mit sich, dass man viele Aussagen über Monome (zumindest in bis zu drei Variablen) anhand der sogenannten *Newton-Diagramme* bildlich darstellen kann: Ein Monom wird mit seinem Exponentenvektor identifiziert, und dieser wird in ein entsprechendes

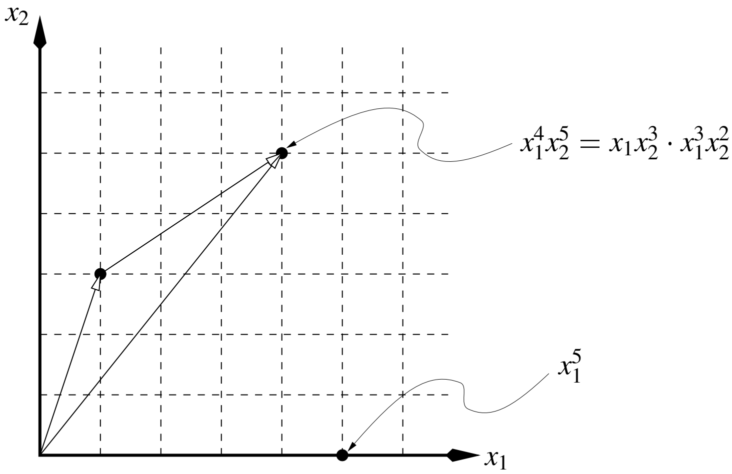


Bild 1: $M_2 \subset K[x_1, x_2]$ dargestellt als ω^2.

Koordinatensystem eingetragen. Bild 1 zeigt dies für die Monome in zwei Veränderlichen. Hier sieht man etwa, was es heißt, zwei Monome zu multiplizieren.

Die Menge aller Teiler bzw. aller Vielfachen eines Monoms x^α lassen sich z.B. dann in diese Diagramme direkt einzeichnen, ist doch

$$\begin{aligned}
&x^\beta \text{ Teiler von } x^\alpha \\
&\qquad \Leftrightarrow \quad x^\alpha = x^\beta x^\gamma \text{ mit } \gamma \in \omega^n \quad \Leftrightarrow \quad \alpha = \beta + \gamma \text{ mit } \gamma \in \omega^n , \\
&x^\beta \text{ Vielfaches von } x^\alpha \\
&\qquad \Leftrightarrow \quad x^\beta = x^\alpha x^\gamma \text{ mit } \gamma \in \omega^n \quad \Leftrightarrow \quad \beta = \alpha + \gamma \text{ mit } \gamma \in \omega^n .
\end{aligned}$$

Das entsprechende Newton-Diagramm zeigt Bild 2.

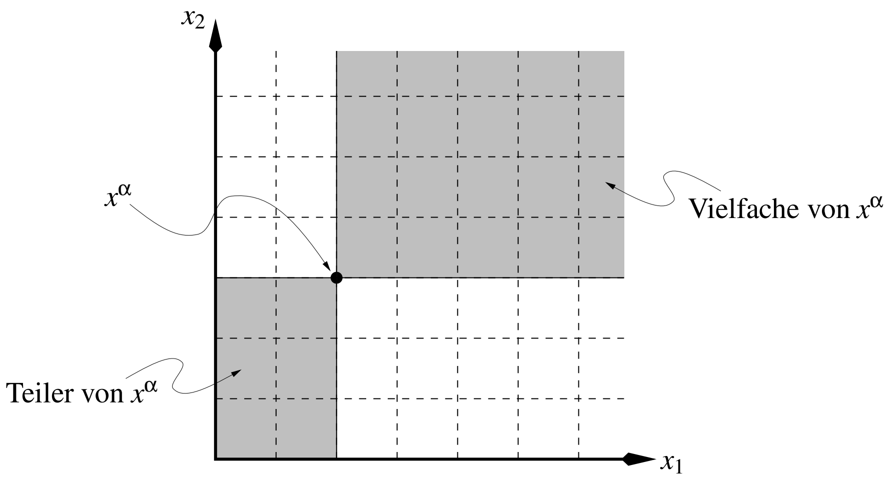


Bild 2: Alle Teiler und Vielfachen von $x^\alpha \in K[x_1, x_2]$.

Für das Verständnis der Begriffe, die später definiert werden, sind solche Diagramme äußerst hilfreich. Auch die Vorgehensweise der verschiedenen Algorithmen kann man sehr gut an ihnen ablesen. Am Ende dieser Arbeit findet sich eine Seite mit Blanko-Diagrammen zum Heraustrennen oder Kopieren.

Im Polynomring einer Veränderlichen ist durch die Gradfunktion auf kanonische Weise eine Anordnung der Monome gegeben. Hier ist $x^a < x^b$ genau dann, wenn $a < b$ (mit $a, b \in \omega$). Im Polynomring mehrerer Veränderlicher dagegen ist die Auswahl an Ordnungen, die mit der Halbgruppenstruktur verträglich sind, sehr viel größer, und keine dieser Ordnungen ist von vornherein als kanonisch prädestiniert. Nun hängen aber die meisten der Begriffe, die im Laufe dieser Arbeit eingeführt werden, in direkter Weise von der Ordnung ab, die man auf den Monomen verwendet. Und nicht zuletzt beruht der hier vorgestellte Algorithmus ja gerade darauf, sich den Wechsel zweier verschiedener Ordnungen zu Nutze zu machen. Es ist also ein gewissen Maß an Sorgfalt vonnöten, wenn Aussagen formuliert werden, die von der verwendeten Ordnung auf den Monomen abhängen. Dennoch werden die Notationen der Übersichtlichkeit halber in der Regel so sein, dass sie die Ordnung nicht explizit widerspiegeln. Falls jedoch Verwechselungen auftreten können, werden die Bezeichnungen so gewählt, dass aus ihnen die benutzte Ordnung hervorgeht.

Definition. *Sei $(G, \cdot)$ eine Halbgruppe. Eine Relation $<$ auf G heißt **Halbgruppenordnung**, wenn gilt:*

i) Für alle a, $b \in G$ gilt entweder $a = b$ oder $a < b$ oder $b < a$.
ii) Sind a, $b \in G$, mit $a < b$, so ist auch $ga < gb$ für alle $g \in G$.

*Besitzt jede nicht leere Teilmenge $H \subseteq G$ bezüglich der Halbgruppenordnung $<$ ein kleinstes Element $g \in H$, so heißt $<$ eine **Wohlordnung**. In diesem Fall schreibt man $g = \min_< H$.*

Bemerkung: Die Bedingung *i)* aus der Definition stellt sicher, dass überhaupt alle Elemente von G miteinander verglichen werden können.

Wohlordnungen auf der Menge der Monome nennt man auch ***Monomordnungen***. Zunächst wird es keine Rolle spielen, ob es sich bei den Ordnungen um Monomordnungen handelt oder nicht. Doch schon ab Kapitel 2 werden vornehmlich nur noch Monomordnungen zur Verwendung kommen. Unter anderem ist der FGLM-Algorithmus nur für Wohlordnungen definiert.

Doch nun erst einmal ein paar Beispiele von Ordnungen auf den Monomen. An dieser Stelle kann man die oben beschriebene Isomorphie von M_n und ω^n ausnutzen: Man definiert eine Ordnung $<_M$ auf M_n, indem man eine Ordnung $<_\omega$ auf ω^n definiert und vereinbart, dass $x^\alpha <_M x^\beta$ genau dann, wenn $\alpha <_\omega \beta$ ist. Fortan werden $<_M$ und $<_\omega$ in der Notation nicht mehr unterschieden.

Bei der Definition der Ordnungen wird außerdem noch folgende Schreibweise verwendet:

Definition. *Sei $w = (w_1, \ldots, w_n) \in \mathbb{R}^n$. Für $\alpha = (\alpha_1, \ldots, \alpha_n) \in \omega^n$ setze*

$$|\,\alpha\,|_w := \sum_{i=1}^{n} w_i \alpha_i \quad \textit{sowie} \quad |\,\alpha\,| := |\,\alpha\,|_{(1,\ldots,1)} = \sum_{i=1}^{n} \alpha_i \,.$$

*Ist $m = x^\alpha \in M_n$, so ist der **Grad** von m definiert als $\mathrm{grad}(m) := |\,\alpha\,|$.*
*Ist $p \in P_n$ mit $p = \sum_{\alpha\in\Phi} c_\alpha x^\alpha \neq 0$, wobei $\Phi \in \omega^n$ und $c_\alpha \in K^*$ für $\alpha \in \Phi$, so ist der **Grad** von p definiert als $\mathrm{grad}(p) := \max\{\mathrm{grad}(x^\alpha) \mid \alpha \in \Phi\}$.*

Wie später gezeigt wird, gibt es die verschiedensten Ordnungen, in der Praxis wird man jedoch hauptsächlich mit den hier aufgeführten Beispielen arbeiten. Vor allem die lexikografische Ordnung spielt eine entscheidende Rolle bei der Lösung polynomialer Gleichungssysteme (siehe hierzu z.B. Cox, Little, O'Shea [8], Kapitel 3).

1. **lexikografische Ordnung** $<_{lex}$

$$\alpha <_{lex} \beta \;\Leftrightarrow\; \exists\, i \in \{1, \ldots, n\} : \alpha_i < \beta_i \text{ und } \alpha_j = \beta_j \;\forall\, 1 \leq j < i$$

2. **grad-lexikografische Ordnung** $<_{deglex}$

$$\alpha <_{deglex} \beta \iff \begin{cases} |\,\alpha\,| < |\,\beta\,| \\ \quad \text{oder} \\ |\,\alpha\,| = |\,\beta\,| \text{ und } \alpha <_{lex} \beta \end{cases}$$

3. **grad-reverse-lexikografische Ordnung** $<_{degrevlex}$

$$\alpha <_{degrevlex} \beta \iff \begin{cases} |\,\alpha\,| < |\,\beta\,| \\ \quad \text{oder} \\ |\,\alpha\,| = |\,\beta\,| \text{ und } \exists\, i \in \{1,\ldots,n\}: \\ \qquad \alpha_i > \beta_i \text{ und } \alpha_j = \beta_j \;\forall\, i < j \leq n \end{cases}$$

4. **grad-lokale Ordnung** $<_{locdeglex}$

$$\alpha <_{locdeglex} \beta \iff \begin{cases} |\,\alpha\,| > |\,\beta\,| \\ \quad \text{oder} \\ |\,\alpha\,| = |\,\beta\,| \text{ und } \alpha <_{lex} \beta \end{cases}$$

Die Ordnungen, die zuerst nach dem Grad sortieren, nennt man ***grad-orientiert***. Indem man den gewöhnlichen Betrag $|\,\alpha\,|$ durch $|\,\alpha\,|_w$ ersetzt, kann man aus ihnen die gewichteten Versionen ableiten. In der Bezeichnung werden die gewichteten Ordnungen oft durch ein vorangestelltes „w“ (für *weighted*) gekennzeichnet.

5. **grad-gewichtete Ordnung** $<_{wdeglex}$ **zu** $w \in \mathbb{R}^n$

$$\alpha <_{wdeglex} \beta \iff \begin{cases} |\,\alpha\,|_w < |\,\beta\,|_w \\ \quad \text{oder} \\ |\,\alpha\,|_w = |\,\beta\,|_w \text{ und } \alpha <_{lex} \beta \end{cases}$$

6. **grad-lokal-gewichtete Ordnung** $<_{wlocdeglex}$ **zu** $w \in \mathbb{R}^n$

$$\alpha <_{wlocdeglex} \beta \iff \begin{cases} |\,\alpha\,|_w > |\,\beta\,|_w \\ \quad \text{oder} \\ |\,\alpha\,|_w = |\,\beta\,|_w \text{ und } \alpha <_{lex} \beta \end{cases}$$

Bemerkung
Die soeben definierten Ordnungen implizieren eine Anordnung der n Unbestimmten des Polynomrings. So gilt z.B. für die lexikografische Ordnung $x_1 > x_2 > \ldots > x_n$, wie man leicht anhand der Definition nachprüft. Dies ist aber völlig willkürlich, man kann eine lexikografische Ordnung ebenso definieren, so dass $x_{\sigma(1)} > \ldots > x_{\sigma(n)}$ für eine beliebige Permutation σ. Strenggenommen gibt es also nicht „die“ lexikografische Ordnung, sondern $n!$ verschiedene, je nach Anordnungen der x_i. Die Ergebnisse werden für verschiedene Permutationen auch wirklich unterschiedlich ausfallen. Ist im Folgenden der Polynomring etwa $K[x_1,\ldots,x_n]$ und ist von einer Ordnung die Rede, so soll implizit immer die Version gemeint sein, für die $x_1 > \ldots > x_n$ ist.

Die hier aufgeführten Beispiele sind aber nur die Spitze des Ordnungseisberges. Die wahre Bandbreite offenbart sich, wenn man anfängt, Ordnungen mit Hilfe von Matrizen zu definieren:

Definition. *Sei $A \in \mathrm{GL}_n(\mathbb{R})$. Dann ist die durch A **induzierte Ordnung** $<_A$ definiert durch*

$$\alpha <_A \beta \;\Leftrightarrow\; A\alpha <_{lex} A\beta \quad \textit{für } \alpha, \beta \in \omega^n \;.$$

Ist $<$ eine Ordnung auf ω^n, und existiert ein $A \in \mathrm{GL}_n(\mathbb{R})$, so dass

$$\alpha < \beta \;\Leftrightarrow\; \alpha <_A \beta \quad \textit{für alle } \alpha, \beta \in \omega^n \;,$$

*so heißt $<$ durch A **repräsentiert**.*

Dass damit dann aber auch alles erschlagen ist, zeigt folgender Satz:

Satz 1. *Sei $<$ eine Ordnung auf ω^n. Dann existiert ein $A \in \mathrm{GL}_n(\mathbb{R})$, so dass $<$ durch A repräsentiert wird.*
Beweis.
Der Beweis findet sich bei L. Robbiano [16].

Im Polynomring $K[x_1, x_2, x_3]$ stellen sich die oben beschriebenen Beispiele, wie man leicht nachprüft, in ihrer Matrizenrepräsentation wie folgt dar:

$$A_{lex} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad A_{deglex} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$A_{degrevlex} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix} \qquad A_{locdeglex} = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

$$A_{wdeglex} = \begin{pmatrix} w_1 & w_2 & w_3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad A_{wlocdeglex} = \begin{pmatrix} -w_1 & -w_2 & -w_3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Von zentraler Bedeutung für den weiteren Verlauf sind die folgenden Begriffe:

Definition. *Sei $p \in P_n$ mit $p = \sum_{\alpha\in\Phi} c_\alpha x^\alpha \neq 0$, wobei $\Phi \subset \omega^n$ und $c_\alpha \in K^*$ für $\alpha \in \Phi$. Sei eine Ordnung $<$ fixiert. Sei $\gamma = \max_< \Phi$ das bezüglich $<$ größte Element von Φ, d.h. x^γ ist das größte Monom von p.*
*a) Der **Multigrad** von p ist definiert als* $\mathrm{multideg}(p) := \gamma$.
*b) Das **Leitmonom** von p ist definiert als* $\mathrm{L}(p) := x^\gamma$.
*c) Der **Leitkoeffizient** von p ist definiert als* $\mathrm{c}(p) := c_\gamma$.
*d) Der **Leitterm** von p ist definiert als* $\mathrm{LT}(p) := c_\gamma x^\gamma = \mathrm{c}(p)\,\mathrm{L}(p)$.
Ferner setze $\mathrm{tail}(p) := p - \mathrm{LT}(p)$.

Beachte: Diese Begriffe sind nicht für das Nullpolynom erklärt!
Ist nun $f = \sum_{\alpha\in\Phi} c_\alpha x^\alpha \in P_n$ ein Polynom und ist eine Ordnung $<$ fixiert, so werden die Monome von f in absteigender Reihenfolge bezüglich der Ordngung $<$ notiert. Für

$$f = 3y^2z + x^5 + 7x^3z^3 + 4y^4z^2 \in \mathbb{Q}[x, y, z]$$

ergibt sich z.B. folgendes Bild:

	Darstellung von f	$\mathrm{L}(f)$	$\mathrm{c}(f)$	$\mathrm{multideg}(f)$	
$<_{lex}$	$x^5 + 7x^3z^3 + 4y^4z^2 + 3y^2z$	x^5	1	$(5,0,0)$	
$<_{deglex}$	$7x^3z^3 + 4y^4z^2 + x^5 + 3y^2z$	x^3z^3	7	$(3,0,3)$	
$<_{degrevlex}$	$4y^4z^2 + 7x^3z^3 + x^5 + 3y^2z$	y^4z^2	4	$(0,4,2)$	
$<_{locdeglex}$	$3y^2z + 7x^3z^3 + 4y^4z^2 + x^5$	y^2z	3	$(0,2,1)$	
$<_{wdeglex}$	$7x^3z^3 + 4y^4z^2 + 3y^2z + x^5$	x^3z^3	7	$(3,0,3)$	$w = (2,3,6)$
$<_{wlocdeglex}$	$4y^4z^2 + 7x^3z^3 + x^5 + 3y^2z$	y^4z^2	4	$(0,4,2)$	$w = (2,3,6)$

Es wird häufig nötig sein, die Leitmonome einer ganzen Menge von Polynomen zu bezeichnen, daher folgende Notation:

Notation. *Für $F \subseteq P_n$ setze*

$$\mathrm{L}(F) := \{\mathrm{L}(f) \mid f \in F, f \neq 0\}.$$

Wie erwähnt interessieren in dieser Arbeit vor allem Wohlordnungen. Sie werden dafür sorgen, dass der formulierte Algorithmus nach endlich vielen Schritten terminiert. Nun ist aber die Bedingung für „wohl“ einer Ordnung aus der Definition nicht gerade handlich: Jede nicht leere Teilmenge muss ein kleinstes Element besitzen. Der folgende Satz liefert glücklicherweise äquivalente Formulierungen dieser Bedingung, von denen man zumindest eine wesentlich besser im Griff hat. Er bildet auch gleichzeitig einen Brückenschlag zum nächsten Kapitel, in dem Ideale eines Polynomrings näher untersucht werden sollen, indem er eine erst dort formulierte Behauptung (Satz 3) in seinem Beweis verwendet. (Es handelt sich hierbei aber nicht um einen Zirkelschluss.)

Satz 2. *Sei $<$ eine Halbgruppenordnung auf ω^n. Dann sind äquivalent:*

(i) $<$ *ist Wohlordnung.*
(ii) $(0,\dots,0)$ *ist kleinstes Element bezüglich* $<$.
(iii) *Jede Kette* $\alpha_1 > \alpha_2 > \dots$ *mit* $\alpha_i \in \omega^n$ *bricht ab.*

Beweis.
$(iii) \Rightarrow (ii)$: Annahme: Es gebe ein $\alpha \in \omega^n$ mit $\alpha < 0$. Dann ist aber auch $\alpha + \alpha < \alpha$, also $n\alpha < (n-1)\alpha$ für alle $n \in \mathbb{N}$. Damit hat man aber eine nicht abbrechende Kette $\alpha > 2\alpha > 3\alpha > \dots \Rightarrow$ Widerspruch.
$(ii) \Rightarrow (i)$: (Diese Richtung verwendet Bezeichnungen und Ergebnisse aus dem nächsten Kapitel.) Annahme: Es gebe eine Menge $\Phi \subseteq \omega^n$, die kein kleinstes Element besitzt. Dann gibt es also eine nicht abbrechende Kette $\alpha_1 > \alpha_2 > \dots$ mit $\alpha_i \in \Phi$. Betrachte das Ideal $I = \langle x^{\alpha_i} \mid i \in \mathbb{N} \rangle$. Da I endlich erzeugt ist, existiert ein $k \in \mathbb{N}$, so dass $I = \langle x^{\alpha_1}, \dots, x^{\alpha_k} \rangle$. Sei nun $l > k$. Da $x^{\alpha_l} \in I$, existiert ein $x^{\beta} \in M_n$ mit $x^{\alpha_l} = x^{\beta}x^{\alpha_i}$ für ein $i \in \{1,\dots,k\}$. Dann ist $x^{\alpha_i} > x^{\alpha_l}$. Nach (ii) ist $1 < x^{\beta}$, also $x^{\alpha_i} < x^{\beta}x^{\alpha_i} = x^{\alpha_l}$. $\Rightarrow$ Widerspruch.
$(i) \Rightarrow (iii)$: Annahme: Die Kette $\alpha_1 > \alpha_2 > \dots$ mit $\alpha_i \in \omega^n$ bräche nicht ab. Dann hat die Menge $\{\alpha_i \mid i \in \mathbb{N}\}$ kein kleinstes Element. $\Rightarrow$ Widerspruch. □

Zum Abschluss dieses Kapitels sei noch auf folgende Tatsache hingewiesen: Ist $<$ ein Wohlordnung auf ω^n, und sind $m, m' \in M_n$ mit $m \neq m'$, so gilt:

$$m \text{ teilt } m' \quad \Rightarrow \quad m < m' .$$

Denn: ist m Teiler von m', so gibt es ein $x^{\gamma} \in M_n$ mit $m' = x^{\gamma} m$. Da $<$ aber eine Wohlordnung ist, ist $1 < x^{\gamma}$ und damit dann auch $m = 1 \cdot m < x^{\gamma} \cdot m = m'$. Dieser Sachverhalt wird oft sehr nützlich sein (und wird falsch, wenn die Ordnung nicht wohl ist).

Ideale im Polynomring

Der FGLM-Algorithmus hat die Aufgabe, eine Standardbasis eines nulldimensionalen Ideals zu berechnen. In diesem Kapitel erfährt man nun, was eine Standardbasis ist. Auf dem Weg dorthin stößt man auf die sogenannten monomialen Ideale mit ihren Eigenschaften. Neben der Möglichkeit, Ideale durch ihre Standardbasis zu repräsentieren, wird noch eine weitere Variante vorgestellt: die Darstellung anhand eines definierenden Systems. Und zu guter Letzt werden noch homogene Ideale eingeführt, da der FGLM-Algorithmus im Rahmen dieser Arbeit auf solche erweitert wurde.

Notation. *Sei $F \subseteq P_n$. Dann bezeichne $\langle F \rangle$ das kleinste Ideal von P_n, in dem F enthalten ist, d.h. $\langle F \rangle$ ist der Schnitt über alle Ideale, die F enthalten.*
Schreibe ferner abkürzend $I = \langle f_1, \ldots, f_n \rangle$ für $I = \langle \{f_1, \ldots, f_n\} \rangle$.

Ist $I = \langle F \rangle$, so sagt man, das Ideal I werde von F ***erzeugt*** und bezeichnet F als das ***Erzeugendensystem*** von I. Ist $F \subseteq P_n$, so hat ein $p \in \langle F \rangle$ die Gestalt

$$p = \sum_{i=1}^{m} p_i f_i \qquad \text{mit } f_i \in F \text{ und } p_i \in P_n \text{ für alle } i.$$

Sind $F, G \subseteq P_n$ und ist $F \subseteq G$, so ist natürlich $\langle F \rangle \subseteq \langle G \rangle$. Ist ferner $I = \langle F \rangle$ und $f \in I$, so ist $\langle F \rangle = \langle F \cup \{f\} \rangle$.

Der nächste Satz zeigt nun glücklicherweise, dass jedes Ideal im Polynomring endlich erzeugt ist, d.h für jedes Ideal existiert ein endliches Erzeugendensystem.

Satz 3. *Der Polynomring P_n ist noethersch, d.h. für jedes Ideal $I \subseteq P_n$ gibt es endlich viele Polynome $f_1, \ldots, f_s \in I$, so dass $I = \langle f_1, \ldots, f_s \rangle$.*

Beweis.
Der Beweis soll hier nur skizziert werden: Zunächst zeige, dass der Polynomring $K[x]$ in einer Unbestimmten noethersch ist. Dazu zeigt man: Ist R ein noetherscher Ring, so ist $R[x]$ ebenfalls noethersch. Da Körper aber automatisch noethersch sind (alle Ideale sind von 0 bzw. 1 erzeugt), folgt die Aussage dann per Induktion. □

Die Tatsache, dass der Polynomring noethersch ist, versetzt einen in die glückliche Lage, ein Ideal im Computer sehr einfach repräsentieren zu können: Man stellt es als die endliche Liste seiner erzeugenden Polynome dar. Nun ist das Erzeugendensystem eines Ideals natürlich alles andere als eindeutig bestimmt, und es stellt sich die Frage, ob alle Erzeugendensysteme gleich gut geeignet sind, das Ideal zu repräsentieren.

Eine besondere Klasse von Erzeugendensystemen bilden die sogenannten *Standardbasen*, die im weiteren Verlauf dieses Kapitels eingeführt werden. Diese haben spezielle Eigenschaften, die sie für die Computeralgebra unentbehrlich machen. Zur Definition müssen erst noch einige Begriffe bereitgestellt werden.

Definition. *Sei $I \subseteq P_n$ ein Ideal. Sei eine Ordnung fixiert. Dann heißt $\langle\, \mathrm{L}(I)\, \rangle$ das* ***Leitideal*** *von I.*

Das Leitideal wird also von einer Menge bestehend aus (unendlich vielen) Monomen erzeugt.

Monomiale Ideale

Solche speziellen Ideale nennt man *monomial*. Diese sollen jetzt kurz etwas näher betrachtet werden.

Definition. *Ein Ideal $I \subseteq P_n$ heißt* ***monomial****, wenn es ein $\Phi \subseteq M_n$ gibt, so dass $I = \langle\, \Phi\, \rangle$. Man nennt das Erzeugendensystem Φ dann ebenfalls* ***monomial****.*

Beachte: In der Definition ist nicht gefordert, dass ein monomiales Erzeugendensystem endlich ist. Dieser Definition nach ist das Leitideal also ein monomiales Ideal. Ein einfaches Beispiel eines Ideals, das nicht monomial ist, ist etwa das Ideal $I = \langle\, x+y\, \rangle \subset K[x,y]$. Denn jedes $f \in I$ hat die Form $f = p \cdot (x+y)$ mit $p \in K[x,y]$, kann also kein Monom sein, folglich kann I kein monomiales Erzeugendensystem haben.

Natürlich ist nicht jedes Erzeugendensystem eines monomialen Ideals monomial: Es ist ja z.B. $\langle\, x+y, x-y\, \rangle = \langle\, x,y\, \rangle \subset K[x,y]$.

Kriterien für die Zugehörigkeit eines Polynoms zu einem monomialen Ideal liefert folgendes Lemma:

Lemma 4. *Sei $I \subseteq P_n$ ein monomiales Ideal, sei $\Psi \subseteq M_n$ ein monomiales Erzeugendensystem von I. Dann gilt:*

a) Ist $m \in M_n$, so ist genau dann $m \in I$, wenn es ein $p \in M_n$ und ein $q \in \Psi$ gibt mit $m = pq$.
b) Ist $p = \sum_{\alpha \in \Phi} c_\alpha x^\alpha \in P_n$, so ist $p \in I$ genau dann, wenn $x^\alpha \in I$ für alle $\alpha \in \Phi$.

Beweis.
Einfach.

Ebenso offensichtlich ist folgende Feststellung.

Lemma 5. *Sei $I \subseteq P_n$ ein monomiales Ideal, sei $\Phi \subseteq M_n$ mit $I = \langle\, \Phi\, \rangle$. Dann existiert eine endliche Teilmenge $\Psi \subseteq \Phi$, so dass $I = \langle\, \Psi\, \rangle$.*

Beweis.
Da P_n noethersch ist, existieren $f_1, \ldots, f_s \in P_n$ mit $I = \langle\, f_1, \ldots, f_n\, \rangle$.
Ist nun $f \in \{f_1, \ldots, f_s\}$ und $f \notin M_n$, so zeigt folgende Überlegung, wie man f im Erzeugendensystem durch endlich viele Monome aus Φ ersetzen kann, ohne das Erzeugnis zu ändern: Sei etwa $f = \sum_{i=1}^{k} c_i x^{\alpha_i}$. Nach dem letzten Lemma ist dann $x^{\alpha_i} \in I$ für alle i. Dann ist aber $x^{\alpha_i} = p_i q_i$ mit $p_i \in M_n$ und $q_i \in \Phi$, d.h. $f = \sum_{i=1}^{k} (c_i p_i) q_i$. Dann kann aber f in $\{f_1, \ldots, f_s\}$ durch $q_1, \ldots, q_k$ ersetzt werden, und man landet schließlich induktiv bei einer endlichen Teilmenge von Φ, die I erzeugt. $\square$

Angewendet auf das Leitideal bekommt man damit direkt folgendes Korollar geschenkt:

Korollar. *Sei $I \subseteq P_n$ ein Ideal, $I \neq \{0\}$, sei eine Ordnung fixiert. Dann gibt es $f_1, \ldots, f_s \in I$, so dass $\langle\, \mathrm{L}(I)\, \rangle = \langle\, \mathrm{L}(f_1), \ldots, \mathrm{L}(f_s)\, \rangle$.*

Beweis.
Spezialfall des Lemmas.

Mit einer einfachen Definition kann man nun erreichen, ein eindeutig bestimmtes monomiales Erzeugendensystem auszuzeichnen.

Definition. *Sei $I \subseteq P_n$ ein monomiales Ideal und sei $\Phi \subset M_n$ ein endliches monomiales Erzeugendensystem von I. Dann heißt Φ* ***minimal****, wenn für jedes $m \in \Phi$ gilt: $m \notin \langle\, \Phi \setminus \{m\}\,\rangle$.*

Nach Lemma 4 heißt Φ also minimal, wenn kein Element von Φ Vielfaches eines anderen Elementes von Φ ist. Beachte: In der Definition ist explizit gefordert, dass das Erzeugendensystem endlich ist. Wie gesehen existiert ein solches aber ja immer.

Satz 6. *Sei $I \subseteq P_n$ ein monomiales Ideal. Dann gilt:*
a) Es existiert ein minimales monomiales Erzeugendensystem von I.
b) Ein minimales monomiales Erzeugendensystem von I ist eindeutig bestimmt.

Beweis.
zu *a*): Nach Lemma 5 existiert ein endliches monomiales Erzeugendensystem $\Phi \subset M_n$ von I. Streicht man aus Φ all die Elemente, die von anderen Elementen aus Φ geteilt werden, so erhält man ein minimales $\Phi' \subseteq \Phi$.
zu *b*): Seien $\Phi, \Phi' \subset M_n$ minimale monomiale Erzeugendensysteme. Sei nun $m \in \Phi$. Dann ist $m \in \langle\, \Phi\,\rangle = I = \langle\, \Phi'\,\rangle$, also existiert ein $m' \in \Phi'$ und ein $p \in M_n$, so dass $m = p\,m'$. Umgekehrt ist natürlich auch $m' \in \langle\, \Phi'\,\rangle = I = \langle\, \Phi\,\rangle$. Dann existiert ein $\overline{m} \in \Phi$ und ein $p' \in M_n$, so dass $m' = p'\,\overline{m}$, also $m = p\,m' = pp'\,\overline{m}$. Somit ist $\overline{m}$ ein Teiler von m, und da Φ minimal ist, folgt $m = \overline{m}$, also $p = p' = 1$, d.h. $m = m' \in \Phi'$. Also ist $\Phi \subseteq \Phi'$. Aus Symmetriegründen folgt dann $\Phi = \Phi'$. □

Standardbasen

Wie hängt das im Korollar zu Lemma 5 gefundene Erzeugendensystem des Leitideals mit dem Erzeugendensystem des Ideals selbst zusammen? Ist $I = \langle\, f_1, \ldots, f_s\,\rangle$, so ist natürlich $\mathrm{L}(\{f_1, \ldots, f_s\}) \subseteq \mathrm{L}(I)$, also $\langle\, \mathrm{L}(f_1), \ldots, \mathrm{L}(f_s)\,\rangle \subseteq \langle\, \mathrm{L}(I)\,\rangle$, aber die Gleichheit gilt im Allgemeinen nicht:

Beispiel
Betrachte etwa das Ideal $I = \langle\, f_1, f_2\,\rangle \subset K[x, y]$ mit $f_1 = x^3 - 2xy$ und $f_2 = x^2y - 2y^2 + x$. Die Monome seien grad-lexikografisch geordnet. Dann ist

$$-y \cdot f_1 + x \cdot f_2 = x^2 \in I\,,$$

also ist $x^2 = \mathrm{L}(x^2) \in \langle\, \mathrm{L}(I)\,\rangle$. Aber x^2 ist weder durch $\mathrm{L}(f_1)$ noch durch $\mathrm{L}(f_2)$ teilbar, nach Lemma 4 ist also $x^2 \notin \langle\, \mathrm{L}(f_1), \mathrm{L}(f_2)\,\rangle$. Fazit: Für diese Erzeuger des Ideals und diese Ordnung ist $\langle\, \mathrm{L}(f_1), \mathrm{L}(f_2)\,\rangle \neq \langle\, \mathrm{L}(I)\,\rangle$.

Nun zeichnet man gerade die Teilmengen eines Ideals aus, deren Leitmonome das Leitideal erzeugen; dies führt endlich zum schon lange angekündigten Begriff der *Standardbasis*.

Definition. *Sei $I \subseteq P_n$ ein Ideal, sei eine Ordnung $<$ fixiert und sei $F \subset I \setminus \{0\}$ endlich. Dann heißt F eine* ***Standardbasis*** *von I bezüglich der Ordnung $<$, wenn*

$$\langle\, \mathrm{L}(F)\,\rangle = \langle\, \mathrm{L}(I)\,\rangle.$$

Ist darüberhinaus $<$ eine Wohlordnung, so heißt F eine ***Gröbnerbasis****.*

Das Korollar zu Lemma 5 stellt sicher, dass zu jedem Ideal und zu jeder Ordnung eine Standardbasis existiert. Man beachte, dass in der Definition nicht gefordert ist, dass die Standardbasis ein Erzeugendensystem für das Ideal ist. Für Gröbnerbasen ist dies auch gar nicht notwendig, da es direkt aus der Definition folgt:

Satz 7. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei $F \subset I$ eine Gröbnerbasis von I bezüglich $<$. Dann ist F ein Erzeugendensystem von I.*

Beweis.
Es ist nur zu zeigen, dass $I \subseteq \langle F \rangle$. Sei also $p \in I$. Dann ist $\mathrm{L}(p) \in \langle \mathrm{L}(I) \rangle = \langle \mathrm{L}(F) \rangle$. Also existiert ein $f \in F$, so dass $\mathrm{L}(f) \mid \mathrm{L}(p)$. Dann setze

$$g_1 = \frac{\mathrm{c}(p)}{\mathrm{c}(f)} \frac{\mathrm{L}(p)}{\mathrm{L}(f)} f \in \langle F \rangle \quad \text{sowie} \quad p_1 = p - g_1 \in I\,.$$

Beachte: Es ist $\mathrm{L}(g_1) = (\mathrm{L}(p)/\mathrm{L}(f))\,\mathrm{L}(f) = \mathrm{L}(p)$, d.h. g_1 ist gerade so gebaut, dass es den Leitterm von p aufhebt. Dann ist also entweder $p_1 = 0$ oder $\mathrm{L}(p_1) < \mathrm{L}(p)$. Verfährt man mit p_1 genauso wie mit p, so erhält man eine Kette $\mathrm{L}(p) > \mathrm{L}(p_1) > \ldots$, die abbrechen muss, da $<$ eine Wohlordnung ist. Es existiert daher ein k, so dass $p_k = 0$. Dann ist

$$0 = p_k = p_{k-1} - g_k = p - \sum_{i=1}^{k} g_i\,.$$

Da aber nach Konstruktion $g_i \in \langle F \rangle$, folgt $p \in \langle F \rangle$. □

Beispiel
Betrachte nochmals das Ideal aus dem Beispiel von oben. Wiederum bezüglich der grad-lexikografischen Ordnung ist

$$F_{deglex} = \{2y^2 - x,\ xy,\ x^2\}$$

eine Gröbnerbasis von I. Dies sieht man an dieser Stelle noch nicht unmittelbar, aber immerhin ist schon einmal $x^2 \in \langle \mathrm{L}(F) \rangle$. Es gibt eine schöne Methode, wie man eine Basis als Gröbnerbasis enttarnt, doch dazu in Kapitel 4 mehr (vergleiche Satz 21).

Da eine Standardbasis über das Leitideal definiert ist, wird das Aussehen einer solchen von der Ordnung abhängen. So ist etwa bezüglich der lexikografischen Ordnung

$$F_{lex} = \{y^3,\ x - 2y^2\}$$

eine Gröbnerbasis von I, nicht aber bezüglich der grad-lexikografischen Ordnung.

Vereinbarung
Von nun an sollen nur noch Gröbnerbasen betrachtet werden, d.h. alle auftretenden Ordnungen sollen ab jetzt Wohlordnungen sein. Ist fortan von einer Gröbnerbasis bezüglich einer Ordnung die Rede, so soll diese automatisch immer eine Wohlordnung sein.

Eine Gröbnerbasis $F \subset P_n$ zu einem Ideal $I \subseteq P_n$ bezüglich einer festen Ordnung ist natürlich ebenfalls bei weitem nicht eindeutig bestimmt. Ist doch z.B. auch $F \cup \{g\}$ für jedes $g \in I$ wiederum eine Gröbnerbasis für I. Es ist aber leicht möglich, eine spezielle Gröbnerbasis auszuzeichnen, die dann zumindest bezüglich einer festen Ordnung eindeutig ist.

Zunächst einmal beobachtet man, dass die Elemente einer Gröbnerbasis beliebig mit Faktoren aus K multipliziert werden können. Im ersten Schritt schränkt man also die Leitkoeffizienten ein:

Definition. *Sei $I \subseteq P_n$ Ideal, sei $F \subset I$ Gröbnerbasis von I bezüglich der Wohlordnung $<$. Dann heißt F* ***normiert****, wenn für jedes $f \in F$ gilt: $\mathrm{c}(f) = 1$.*

Nun entfernt man redundante Elemente. Ist nämlich F eine Gröbnerbasis und sind $f, g \in F$ mit $f \neq g$ und $\mathrm{L}(f) \mid \mathrm{L}(g)$, so ist $\langle \mathrm{L}(F) \rangle = \langle \mathrm{L}(F \setminus \{g\}) \rangle$. Also kann g aus F gestrichen werden. Dies führt zu:

Definition. *Sei $I \subseteq P_n$ Ideal, sei $F \subset I$ Gröbnerbasis von I bezüglich der Wohlordnung $<$. Dann heißt F* ***minimal****, wenn für jedes $f \in F$ gilt: $\mathrm{L}(f) \notin \langle\, \mathrm{L}(F \setminus \{f\})\,\rangle$.*

Bis hierhin hat man schon einmal die Anzahl der Erzeuger minimiert, aber man kann immer noch an den Erzeugern selbst drehen. Um dies zu sehen seien $f, g \in F$, $f \neq g$, so dass der Leitterm von g irgendein Monom von f teilt: Sei etwa $f = \sum_{\alpha\in\Phi} c_\alpha x^\alpha$ und $\mathrm{L}(g)x^\gamma = x^\beta$ für ein $\beta \in \Phi$. Ist $x^\beta = \mathrm{L}(f)$, so kann man wie gesehen f aus F einfach streichen. Ansonsten setze

$$f' = f - x^\gamma g \in I\ .$$

Dann ist $\mathrm{L}(f') = \mathrm{L}(f)$, und man kann getrost f in F durch f' ersetzen. Ist insbesondere F minimal, so wird durch dieses Prozedere $|\,F\,|$ nicht verändert. Also:

Definition. *Sei $I \subseteq P_n$ Ideal, sei $F \subset I$ Gröbnerbasis von I bezüglich der Wohlordnung $<$. Dann heißt F* ***reduziert****, wenn für jedes $f \in F$ gilt: Kein Monom von f liegt in $\langle\, \mathrm{L}(F \setminus \{f\})\,\rangle$.*

Reduzierte Gröbnerbasen sind somit automatisch minimal.

Bemerkung
In der Literatur werden manchmal nur solche Gröbnerbasen minimal bzw. reduziert genannt, die auch gleichzeitig normiert sind. Betrachtet man aber z.B. Polynome über $\mathbb{Q}$, so werden normierte Gröbnerbasen wegen der vielen Brüche in den Koeffizienten schnell unübersichtlich. Hier scheint es wesentlich sinnvoller, dafür zu sorgen, dass die Koeffizienten in $\mathbb{Z}$ liegen. Erreicht man noch, dass alle Koeffizienten eines Polynoms des Erzeugendensystems teilerfremd sind, so hat man auf eine wesentlich lesbarere Art „normiert“. Da genau diese Methode in SINGULAR verwirklicht ist, wurde auf die Normierung in der Definition zu verzichtet.

Mit diesen drei Definitionen hat man das Ziel erreicht:

Satz 8. *Sei $<$ eine Wohlordnung auf ω^n, sei $I \subseteq P_n$ ein Ideal. Dann gilt:*
a) Es existiert eine normierte, reduzierte Gröbnerbasis von I bezüglich $<$.
b) Eine normierte, reduzierte Gröbnerbasis von I bezüglich $<$ ist eindeutig bestimmt.

Beweis.
zu *a*): Wie gesehen existiert eine Gröbnerbasis von I bezüglich $<$. Die Motivationen für obige Definitionen zeigen, wie daraus eine normierte und reduzierte Gröbnerbasis F' zu konstruieren ist.
zu *b*): Seien $F = \{f_1, \ldots, f_r\}$ und $G = \{g_1, \ldots, g_s\}$ zwei normierte, reduzierte Gröbnerbasen von I. Da F und G minimal sind, sind $\mathrm{L}(F)$ und $\mathrm{L}(G)$ minimale monomiale Erzeuger von $\langle\, \mathrm{L}(I)\,\rangle$. Nach Satz 6 sind solche aber eindeutig bestimmt, d.h. es ist $r = s$ und nach Umsortierung auch $\mathrm{L}(f_i) = \mathrm{L}(g_i)$ für alle i. Da F und G normiert sind, ist auch $\mathrm{c}(f_i) = \mathrm{c}(g_i)$ für alle i.
Annahme: Es sei $f_i \neq g_i$ für ein i. Dann ist also $h = f_i - g_i \neq 0$ und $\mathrm{L}(h) < \mathrm{L}(f_i)$. Sei o.B.d.A. $\mathrm{L}(h)$ ein Monom von f_i. Da $\mathrm{L}(h) \in \langle\, \mathrm{L}(I)\,\rangle = \langle\, \mathrm{L}(F)\,\rangle$, existiert ein k, so dass $\mathrm{L}(f_k) \mid \mathrm{L}(h)$. Da aber F reduziert und $\mathrm{L}(h)$ ein Monom von f_i ist, folgt $k = i$, also $\mathrm{L}(f_i) \mid \mathrm{L}(h)$.
$\Rightarrow$ Widerspruch, da $\mathrm{L}(h) < \mathrm{L}(f_i)$ und $<$ eine Wohlordnung ist. □

Dieser Satz wird falsch, wenn man auf die Voraussetzung verzichtet, dass die Ordnung eine Wohlordnung ist: Für beliebige Ordnungen muss eine reduzierte Standardbasis nicht existieren.

Gröbnerbasen sind bei einer Reihe von Fragestellungen ein überaus mächtiges Werkzeug. Man kann mit ihnen den Schnitt zweier Ideale berechnen, polynomiale Gleichungssysteme lösen, testen, ob ein Polynom in einem Ideal liegt, oder auch zwei Ideale auf Gleichheit testen. Letzteres ist direkt einsichtig: Sind $I = \langle F \rangle$ und $J = \langle G \rangle$ zwei Ideale, so rechnet man für beide eine reduzierte, normierte Gröbnerbasis bezüglich einer gemeinsamen Ordnung aus. Dann ist $I = J$ genau dann, wenn diese beiden Gröbnerbasen übereinstimmen. Da hierbei die Ordnung beliebig ist, kann man sich eine solche aussuchen, für die die Gröbnerbasenberechnung möglichst einfach ist.

Um polynomiale Gleichungssysteme zu lösen, benötig man dagegen eine Gröbnerbasis bezüglich der lexikografischen Ordnung. In den meisten Fällen ist es aber gerade sehr aufwendig, für diese spezielle Ordnung eine Basis auszurechnen. Hier kommt der FGLM-Algorithmus als eine Alternative zum Zuge: Man berechnet die Gröbnerbasis bezüglich einer einfacheren Ordnung und wandelt diese danach mittels FGLM in eine lexikografische Basis um. In manchen Fällen ist dieser vermeintliche Umweg um ein Vielfaches schneller als der direkte Weg bzw. macht eine Berechnung überhaupt erst möglich.

Definierende Systeme

Die Standardbasen bieten also eine sehr schöne Möglichkeit, ein Ideal (wenn man will sogar eindeutig) zu repräsentieren. Im Folgenden wird nun eine alternative Darstellungsmöglichkeit eines Ideals vorgestellt, nämlich gerade als Schnitt über die Kerne einer Menge von linearen Abbildungen. Diese Art der Darstellung – *definierendes System* genannt – hat den Vorteil, dass man sehr leicht prüfen kann, ob ein gegebenes Polynom in dem Ideal liegt oder nicht: Man muss es einfach durch die Abbildungen auswerten. Der FGLM-Algorithmus verwendet definierende Systeme zur Darstellung eines Ideals und nutzt essentiell die Linearität der Abbildungen aus. Doch dazu in Kapitel 5 mehr.

Definition. *Sei $I \subseteq P_n$ ein Ideal. Eine Familie $(l_\varphi \mid \varphi \in \Phi)$ von K-linearen Abbildungen $l_\varphi : P_n \to K$ heißt **definierendes System** für I, wenn*

$$I = \bigcap_{\varphi \in \Phi} \operatorname{Kern} l_\varphi \ .$$

*Ist Φ endlich, so heißt das definierende System $(l_\varphi \mid \varphi \in \Phi)$ **endlich**.*

Ist $(l_\varphi \mid \varphi \in \Phi)$ ein definierendes System, so gilt also

$$p \in I \quad \Leftrightarrow \quad l_\varphi(p) = 0 \ \text{ für alle } \varphi \in \Phi \ .$$

Ist $(l_\varphi \mid \varphi \in \Phi)$ ein endliches definierendes System für I, ist etwa $\Phi = \{1, \ldots, d\}$, so fasse die l_i zu einer K-linearen Abbildung $\mathcal{L} : P_n \to K^d$ zusammen:

$$\mathcal{L}(p) := \big(l_1(p), \ldots, l_d(p)\big) \in K^d \quad \text{für } p \in P_n \ .$$

Dann ist also $I = \operatorname{Kern} \mathcal{L}$ und man spricht von dem endlichen definierenden System $\mathcal{L}$.

Beispiel
Seien $\zeta_1, \ldots, \zeta_d \in K^n$ paarweise verschiedene Punkte in K^n. Setze

$$I = \{p \in P_n \mid p(\zeta_i) = 0 \in K \text{ für alle } i\} \ ,$$

d.h. I ist die Menge aller Polynome, die alle Punkte ζ_i als Nullstellen haben. Dann ist I ein Ideal: Natürlich ist $0 \in I$. Ferner gilt

$$(f+g)(\zeta_i) = f(\zeta_i) + g(\zeta_i) = 0 \quad \text{für alle } i \text{ und alle } f, g \in I\ ,$$
$$(pf)(\zeta_i) = p(\zeta_i) \cdot f(\zeta_i) = p(\zeta_i) \cdot 0 = 0 \quad \text{für alle } i \text{ und alle } f \in I \text{ und } p \in P_n\ .$$

Also $f+g \in I$ und $pf \in I$, daher ist I ein Ideal. Setze nun

$$l_i : P_n \to K \quad \text{vermöge} \quad l_i(p) = p(\zeta_i) \quad \text{für } i = 1, \ldots, d\ .$$

Dann ist l_i als Auswertungshomomorphismus K-linear und

$$\mathcal{L} = (l_1, \ldots, l_d) : P_n \to K^d$$

ist ein endliches definierendes System für I:

$$f \in I \;\Leftrightarrow\; f(\zeta_i) = 0\ \forall\, i \;\Leftrightarrow\; l_i(f) = 0\ \forall\, i \;\Leftrightarrow\; \mathcal{L}(f) = 0 \;\Leftrightarrow\; f \in \operatorname{Kern} \mathcal{L}\ .$$

So wird also durch das endliche definierende System $\mathcal{L}$ das Ideal aller Polynome beschrieben, die Nullstellen an einer vorgegebenen Menge von Punkten $\zeta_1, \ldots, \zeta_d$ haben.

In Kapitel 5 wird ein weiteres definierendes System vorgestellt, das auf die Aufgabe zugeschnitten ist, eine Gröbnerbasis in eine andere umzurechnen.

Homogene Ideale

Wie in der Einleitung erwähnt, ist der ursprüngliche FGLM-Algorithmus (Faugère et al. [11]) nur auf nulldimensionale Ideale anwendbar. Die Nulldimensionalität liefert unter anderem ein Abbruchkriterium nach endlich vielen, sogar vorher bestimmbar vielen, Schritten, das auf allgemeine Ideale nicht mehr anwendbar ist. Im Rahmen dieser Arbeit wurde der Algorithmus nun auf eine weitere Klasse von Idealen, die *homogenen* Ideale, erweitert. Die Idee, Berechnungen im Polynomring auf Rechnungen in einem Vektorraum endlicher Dimension über K zu übertragen, wurde beibehalten. Als Abbruchkriterium wird nun jedoch die Hilbertfunktion zu Rate gezogen, die sich für homogene Ideale berechnen lässt.

Nachfolgend werden also homogene Ideale vor- und einige nützliche Eigenschaften bereitgestellt; die Hilbertfunktion kommt dann in Kapitel 3 zum Zuge.

Definition. *Sei $p \in P_n$ mit $p = \sum_{\alpha\in\Phi} c_\alpha x^\alpha \neq 0$. Ist $k \in \omega$ und gilt*

$$\operatorname{grad}(x^\alpha) = k \quad \textit{für alle } \alpha \in \Phi\ ,$$

*so heißt p **homogen** vom Grad k.*

Homogene Polynome spielen eine zentrale Rolle, wenn man algebraische Geometrie über dem projektiven Raum $\mathbb{P}^n(K)$ betreibt. Betrachtet man dort nämlich Varietäten von Idealen, so hat dies nur Sinn, wenn man sich auf homogene Ideale beschränkt, eine Folge der homogenen Koordinaten im projektiven Raum, die nur bis auf ein skalares Vielfaches eindeutig sind. Eine Einführung in die projektive Welt, in die Berechnung projektiver Varietäten, in den Zusammenhang zwischen affinen und projektiven Varietäten und vieles mehr findet sich in Cox, Little, O'Shea [8].

Ähnlich der Zerlegung eines Polynoms als eindeutige Summe seiner Terme wird nun ein Polynom so in eine Summe zerlegt, dass jeder Summand homogen ist.

Definition. *Sei $p \in P_n$, sei $\Psi \subset \mathbb{N}$ und seien $p_k \in P_n$ für $k \in \Psi$, so dass gilt:*
i) Für $k \in \Psi$ ist p_k homogen vom Grad k.
ii) $p = \sum_{k\in\Psi} p_k$.
*Dann heißt p_k die k-te **homogene Komponente** von p.*

Wie man sich leicht überzeugt, ist die Zerlegung eines Polynoms in seine homogenen Komponenten eindeutig bestimmt, d.h. zwei Polynome sind genau dann gleich, wenn ihre k-ten homogenen Komponenten gleich sind.

Seien nun $p, q \in P_n$ zwei homogene Polynome vom Grad k bzw. l. Dann ist zwar das Produkt $p \cdot q$ homogen zum Grad $k \cdot l$, aber die Summe $p + q$ ist nur dann wieder homogen, wenn $k = l$. Man kann also nicht erwarten, dass es ein Ideal gibt, welches nur aus homogenen Polynomen besteht. Vielmehr muss man folgendermaßen definieren:

Definition. *Ein Ideal $I \subseteq P_n$ heißt **homogen**, wenn für jedes $p \in I$ alle homogenen Komponenten von p ebenfalls wieder in I liegen.*

Wie der nächste Satz zeigt, kann man ein Ideal als homogen enttarnen, wenn man eine reduzierte Gröbnerbasis bezüglich einer beliebigen Ordnung untersucht.

Satz 9. *Sei $I \subseteq P_n$ ein Ideal. Dann sind äquivalent:*
(i) I ist homogen.
(ii) Es existieren homogene Polynome $f_1, \ldots, f_s \in P_n$ mit $I = \langle f_1, \ldots, f_s \rangle$.
(iii) Ist $<$ eine Wohlordnung, so besteht die normierte, reduzierte Gröbnerbasis von I aus homogenen Polynomen.

Beweis.
$(i) \Rightarrow (ii)$: Sei $G \subset P_n$ ein endliches Erzeugendensystem von I. Sei $g \in G$ nicht homogen. Sei $g = \sum_{k\in\Psi} g_k$ die Zerlegung von g in seine homogenen Komponenten. Da I homogen ist, folgt $g_k \in I$, also kann g in G durch seine homogenen Komponenten g_k ersetzt werden. So erhält man schließlich induktiv ein homogenes Erzeugendensystem.
$(ii) \Rightarrow (iii)$: Für diese Richtung werden Ergebnisse und Notationen aus Kapitel 4 verwendet. Sei $F \subset P_n$ ein homogenes Erzeugendensystem von I. Sei $<$ ein Wohlordnung. Nun betrachte man, wie der Buchberger-Algorithmus aus F eine Gröbnerbasis konstruiert. Dort wird F so lange um die Normalform gewisser S-Polynome zweier Elemente aus F erweitert, bis schließlich F eine Gröbnerbasis ist. Nun ist aber die Normalform des S-Polynoms zweier homogener Polynome wiederum homogen, und damit ist die aus F konstruierte Basis ebenfalls homogen.
Betrachtet man nun, wie daraus eine normierte, reduzierte Basis zu erstellen ist, so werden dazu aus F gewisse Polynome f teils einfach gestrichen oder aber durch $f - x^\alpha f'$ ersetzt, wobei x^α gerade so gewählt ist, dass f und $x^\alpha f'$ homogen vom gleichen Grad sind. Also wird in diesem Fall f durch ein neues homogenes Polynom ersetzt. Insgesamt besteht somit die reduzierte Basis nur aus homogenen Polynomen.
$(iii) \Rightarrow (i)$: Sei $<$ eine Wohlordnung und sei die normierte, reduzierte Gröbnerbasis $F \subset I$ von I homogen. Sei $p \in I$ und sei $p = \sum_{k\in\Psi} p_k$ die homogene Zerlegung von p. Es existieren $q_f \in P_n$ für $f \in F$, so dass $p = \sum_{f\in F} q_f f$. Für $f \in F$ sei i_f der homogene Grad von f und $q_f = \sum_{l\in\Psi_f} q_{f,l}$ die homogene Zerlegung von q_f. Dann ist $q_{f,l} f$ homogen von Grad $l + i_f$, und

es gilt

$$p = \sum_{f\in F} q_f f = \sum_{f\in F} \big(\sum_{l\in\Psi_f} q_{f,l} f \big) = \sum_{j\in\Phi} \underbrace{\Big(\sum_{\substack{f\in F\\ l\in\Psi_f\\ i_f+l=j}} q_{f,l} f \Big)}_{\text{homogen vom Grad } j} = \sum_{k\in\Psi} p_k \,.$$

Also ist $p_k = \sum_{\ldots} q_{f,l} f \in I$, d.h. I ist homogen. □

Man erinnere sich noch einmal an die Definition der grad-lexikografischen sowie der lexikografischen Ordnung (siehe Seite 7) und überlege sich, wie das Leitmonom eines homogenen Polynoms aussieht. Die grad-lexikografische Ordnung überprüft zunächst, welches Monom den größten Grad hat. Da das Polynom aber homogen ist, haben alle Monome den gleichen Grad und die Ordnung entscheidet nun danach, welches Monom bezüglich der lexikografischen Ordnung am größten ist. Also: Beide Ordnungen entscheiden sich bei homogenen Polynomen für das gleiche Leitmonom. Daher gilt folgender Satz:

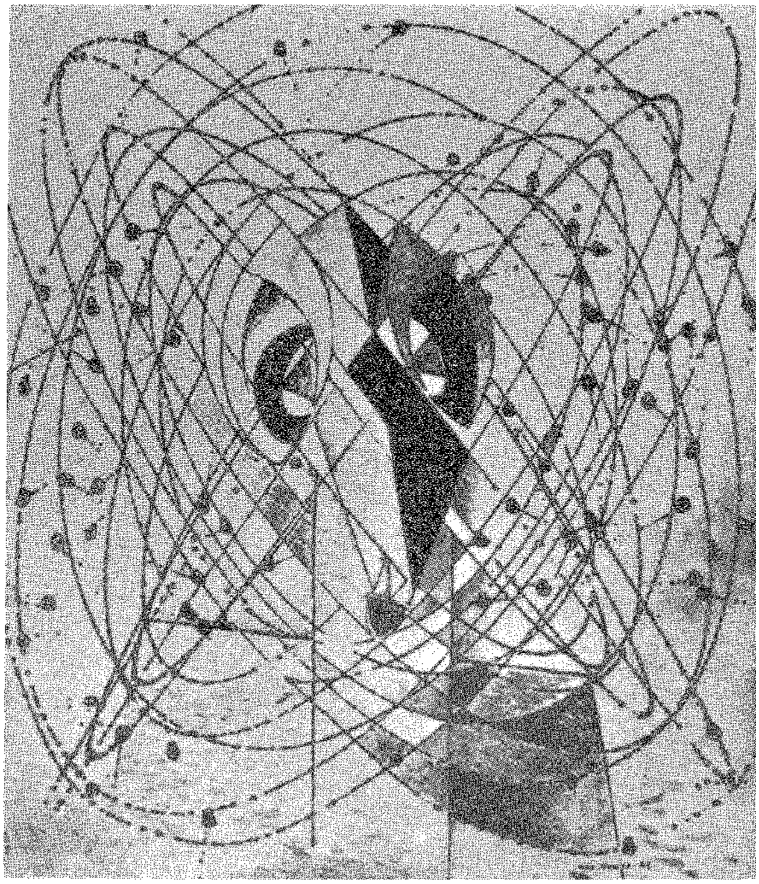
Satz 10. *Sei $I \subseteq P_n$ ein homogenes Ideal. Dann gilt:*

a) *Ist $F \subset I$, so ist F eine Gröbnerbasis bezüglich der Ordnung $<_{lex}$ genau dann, wenn F eine Gröbnerbasis bezüglich der Ordnung $<_{deglex}$ ist.*
b) *Die reduzierte, normierte Gröbnerbasis von I bezüglich der Ordnung $<_{lex}$ ist gleich der bezüglich der Ordnung $<_{deglex}$.*

Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 8, §3.

Allgemeiner gilt dieser Satz für zwei Ordnungen, die sich nur darin unterscheiden, dass die eine zunächst nach dem Grad sortiert, bei gleichem Grad aber wie die zweite entscheidet, also gerade für solche, die für homogene Polynome das gleiche Leitmonom erzeugen.

„Ich gebe zu, daß zwei mal zwei gleich vier eine fabelhafte Sache ist;
aber wenn man schon alles loben will,
so ist auch zwei mal zwei gleich fünf mitunter ein allerliebstes Sächelchen."
F. M. Dostojewskij, Aufzeichnungen aus dem Kellerloch, Reinbek 1962



„Junger Mann, beunruhigt durch den Flug einer nicht-euklidischen Fliege"

Max Ernst, 1942 und 1947

Der Quotientenring

Ziel dieses Abschnittes ist es, ein nulldimensionales Ideal zu definieren. Dies geschieht mit Hilfe der Eigenschaften des Quotientenringes von P_n modulo einem Ideal. Von großer Wichtigkeit wird die Zerlegung der Menge der Monome bezüglich eines Leitideals sein sowie der Zusammenhang dieser Zerlegung zum Quotientenring. Nachdem dann eine ganze Palette von äquivalenten Bedingungen zur Nulldimensionalität vorgeführt werden, wird in Hinblick auf die Verallgemeinerung auf homogene Ideale schließlich noch die Hilbertfunktion definiert, die ebenfalls mit dem Quotientenring zusammenhängt.

Die Zerlegung

In diesem Abschnitt soll die Menge M_n der Monome näher untersucht werden. Durch ein Ideal und eine Wohlordnung ist eine Aufteilung der Monome in verschiedene Teilmengen gegeben, die in enger Verbindung zu einer Gröbnerbasis stehen. Diese sogenannte *Zerlegung* wird von großer Bedeutung für den FGLM-Algorithmus sein.

Sei also $I \subset P_n$ ein Ideal, so dass $I \neq \langle\, 0\,\rangle$ und $I \neq \langle\, 1\,\rangle$. Sei $<$ eine Wohlordnung auf M_n. Dann ist $\mathrm{L}(I) \neq \{\}$ und $\mathrm{L}(I) \neq M_n$. Die Menge der Leitmonome ist abgeschlossen unter der Multiplikation mit Monomen, d.h. ist $m \in \mathrm{L}(I)$, so ist $m \cdot m' \in \mathrm{L}(I)$ für alle $m' \in M_n$. Die Multiplikation führt also nicht aus der Menge heraus. Unter der Division ist $\mathrm{L}(I)$ aber nicht abgeschlossen, d.h. es gibt Monome $m \in \mathrm{L}(I)$, so dass es einen Teiler x_i gibt mit $m/x_i \notin \mathrm{L}(I)$. Ist z.B. $m = \min_< \mathrm{L}(I)$, so ist zunächst $m \neq 1$, da $I \neq \langle\, 1\,\rangle$. Also existiert ein x_i mit $x_i \mid m$. Dann ist $m/x_i < m$, also $m/x_i \notin \mathrm{L}(m)$ aufgrund der Minimalität von m.

Für die Menge $M_n \setminus \mathrm{L}(I)$ verhält es sich genau umgekehrt: Sie ist nicht unter der Multiplikation abgeschlossen, denn sonst wäre $1 \cdot m = m \in M_n \setminus \mathrm{L}(I)$ für alle $m \in M_n$, also $\mathrm{L}(I) = \{\}$, also $I = \langle\, 0\,\rangle$. Ist andererseits $m \in M_n \setminus \mathrm{L}(I)$ und ist m' Teiler von m, so ist $m/m' \in M_n \setminus \mathrm{L}(I)$, denn sonst wäre $m \in \mathrm{L}(I)$, da $\mathrm{L}(I)$ ja unter der Multiplikation abgeschlossen ist.

Bezüglich einer Ordnung werden die Monome also durch ein Ideal bzw. dessen Leitmonome in verschiedene Mengen aufgeteilt. Da diese für den weiteren Verlauf sehr wichtig sind, bekommen sie eine eigene Bezeichnung.

Definition. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung auf M_n. Dann setze*

$$
\begin{aligned}
N &:= M_n \setminus \mathrm{L}(I)\ ,\\
E &:= \left\{ m \in \mathrm{L}(I) \,\middle|\, \forall\, x_i \mid m \ :\ \frac{m}{x_i} \notin \mathrm{L}(I) \right\} ,\\
S &:= \left\{ m \in \mathrm{L}(I) \,\middle|\, \exists\, x_i, x_j \ :\ x_i \mid m,\ x_j \mid m,\ \frac{m}{x_i} \in \mathrm{L}(I),\ \frac{m}{x_j} \notin \mathrm{L}(I) \right\} ,\\
R &:= E \cup S\ .
\end{aligned}
$$

*Man nennt (N, E, S) die **Zerlegung** von I bezüglich der Ordnung $<$, N die **Menge der nicht reduzierbaren Monome**, sowie E die **Ecken**, S die **Seiten** und R den **Rand** von $\mathrm{L}(I)$.*

Beachte: Die Mengen N und S können je nach Ideal endlich oder unendlich sein. Satz 15 wird eine Reihe von äquivalenten Bedingungen liefern, die zur Endlichkeit von N und damit auch von S führen.

Für den FGLM-Algorithmus ist die Zerlegung von essentieller Bedeutung, und ein wichtiges Ziel wird es sein, diese berechnen zu können.

Manchmal benutzt man auch folgende Sprechweise: Ist $m \in M_n$ und existiert ein x_i, so dass x_i ein Teiler von m ist, so heißt $m' = m/x_i$ ein ***direkter Vorgänger*** von m. Damit kann man obige Mengen wie folgt beschreiben: Nicht reduzierbare Monome sind solche, die nicht im Leitideal liegen. Elemente aus dem Rand liegen im Leitideal, aber mindestens ein direkter Vorgänger ist nicht reduzierbar. Ecken wiederum sind genau die Elemente des Randes, für die sämtliche direkten Vorgänger nicht reduzierbar sind. Ein Bild wird gleich Klarheit verschaffen.

Mit Hilfe einer Gröbnerbasis lassen sich diese Mengen sehr einfach in einem Newton-Diagramm darstellen. Sei also $I \subseteq P_n$ ein Ideal und sei $F = \{f_1, \ldots, f_s\}$ eine Gröbnerbasis von I bezüglich einer Wohlordnung $<$. Dann ist nach den Ergebnissen aus Kapitel 2:

$$\mathrm{L}(I) = \langle\, \mathrm{L}(I) \,\rangle \cap M_n = \langle\, \mathrm{L}(f_1), \ldots, \mathrm{L}(f_s) \,\rangle \cap M_n = \bigcup_{i=1}^{s} \mathrm{L}(f_i) \cdot M_n \,.$$

Ist darüber hinaus F eine minimale Gröbnerbasis und ist (N, E, S) die Zerlegung von I bezüglich $<$, so ist sogar

$$E = \{\mathrm{L}(f_1), \ldots, \mathrm{L}(f_s)\}\,,$$

denn: Wäre x_i ein Teiler von $\mathrm{L}(f)$ für ein $f \in F$, so dass $\mathrm{L}(f)/x_i \in \mathrm{L}(I)$, so wäre $\mathrm{L}(f)/x_i \in \langle\, \mathrm{L}(F \setminus \{f\}) \,\rangle$, was im Widerspruch zur Minimalität von F steht.

Mit dieser Erkenntnis ist es nun ein Leichtes, die Zerlegung in ein Newton-Diagramm einzuzeichnen.

Beispiel
Betrachte das Ideal $I = \langle\, y^4 + x^2y,\ x^2y^2 + x^2y,\ x^3y - y^4 \,\rangle \subseteq K[x,y]$. Bezüglich der lexikografischen Ordnung ist $F_{lex} = \{f_1, f_2, f_3\}$ mit

$$f_1 = y^5 + y^4,\ f_2 = xy^4 - y^4,\ f_3 = x^2y + y^4$$

eine minimale Gröbnerbasis von I. Ist (N, E, S) die Zerlegung von I bezüglich $<_{lex}$, so zeigt Bild 3 diese Mengen im Newton-Diagramm.

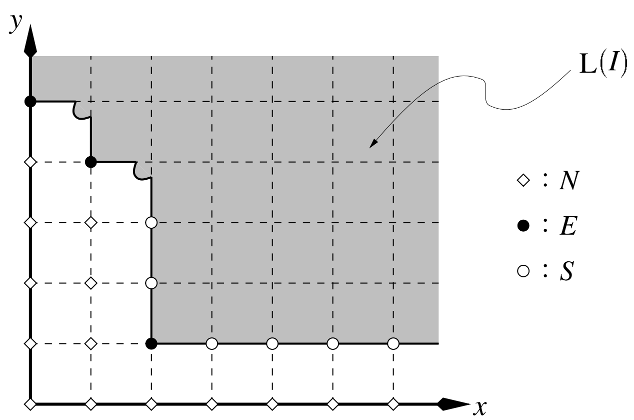


Bild 3: Die Zerlegung (N, E, S) von I bezüglich $<_{lex}$.

Jetzt erklärt sich auch die Namensgebung: Die Menge der Monome wird durch die Leitmonome eines Ideals und durch die nicht reduzierbaren Monome in zwei disjunkte Mengen aufgeteilt. Die „Trennlinie“ zwischen diesen beiden Mengen bildet R, eben der Rand. Für Elemente aus dem Rand gibt es direkte Vorgänger, die nicht reduzierbar sind. Im Rand wiederum sind die Monome ausgezeichnet, für die sämtliche direkten Vorgänger nicht reduzierbar sind, und wie man im Bild sieht, sind dies gerade die „spitzen“ Ecken des Randes. Man beachte, dass etwa x^2y^4 nicht zum Rand gehört.

Um zu sehen, dass die Zerlegung von der Ordnung abhängt, folgt nun die Situation bezüglich der grad-lexikografischen Ordnung. Hier ist $F_{deglex} = \{g_1, g_2, g_3\}$ mit

$$g_1 = y^4 + x^2y,\ g_2 = x^2y^2 + x^2y,\ g_3 = x^3y - x^2y$$

eine minimale Gröbnerbasis. Die Zerlegung (N, E, S) nach der grad-lexikografischen Ordnung kann man sich in Bild 4 ansehen.

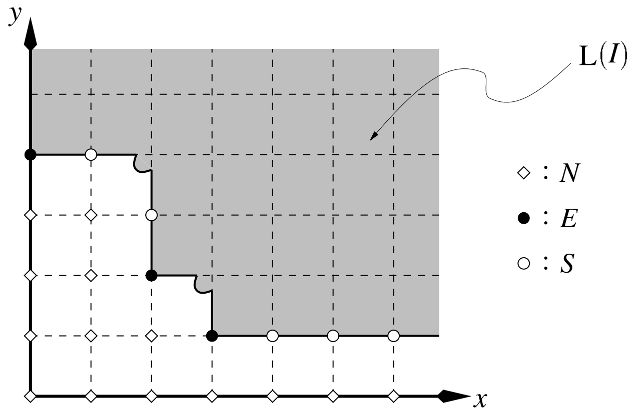


Bild 4: Die Zerlegung (N, E, S) von I bezüglich $<_{deglex}$.

Beachte: In beiden Fällen ist N keine endliche Menge.

Die Erkenntnis, dass die Leitmonome einer minimalen Gröbnerbasis gerade die Ecken der Zerlegung sind, führt direkt zu diesem Lemma:

Lemma 11. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei (N, E, S) die Zerlegung von I bezüglich $<$. Sei $F \subset I$, so dass $\mathrm{L}(f) \neq \mathrm{L}(f')$ für $f, f' \in F$ mit $f \neq f'$. Ist dann $E = \{\mathrm{L}(F)\}$, so ist F eine minimale Gröbnerbasis von I.*

Beweis.
Es ist doch

$$\langle\, \mathrm{L}(I)\, \rangle = \langle\, \bigcup_{e \in E} e \cdot M_n\, \rangle = \langle\, E\, \rangle = \langle\, \mathrm{L}(F)\, \rangle\,.$$

Die Minimalität folgt dann direkt aus der Definition von E. □

Hier zeichnet sich schon die Grundidee des FGLM-Algorithmus ab: Man berechnet die Zerlegung des Ideals bezüglich der gewünschten Ordnung und versucht, für jedes $e \in E$ ein Polynom aus dem Ideal zu finden, das dieses e als Leitmonom hat. Lemma 11 sagt dann, dass man auf diese Art eine Gröbnerbasis des Ideals gefunden hat. Beim Aufspüren der entsprechenden Polynome werden dabei definierende Systeme als Handwerkszeug zur Verfügung stehen und gute Dienste leisten.

Es zeigt sich leider, dass diese Idee nicht für alle Ideale in endlicher Zeit ein Ergebnis zu Tage bringt: Ist nämlich die Zerlegung des Ideals nicht endlich, d.h. sind N bzw. R unendliche Mengen, so terminiert der Algorithmus nicht mehr. Daher funktioniert er in seiner ursprünglichen Version nur für solche Ideale, für die die Zerlegung endlich ist.

Die Normalform

Man kann den Polynomring P_n auf kanonische Weise auch als einen K-Vektorraum auffassen. Dann ist dieser ein Vektorraum von unendlicher Dimension, und als Basis fällt einem sofort die Menge M_n der Monome ein. Dass diese eine K-Basis von P_n bilden, ist offensichtlich.

Ist nun $I \subseteq P_n$ ein Ideal, so ist I auch ein Unterraum von P_n als Vektorraum. (Die Umkehrung gilt natürlich nicht: Ist $f \in P_n$, so ist $\{kf \mid k \in K\}$ zwar ein Unterraum, aber kein Ideal.) Also kann man P_n/I ebenfalls wieder als K-Vektorraum betrachten. Insgesamt wird somit der Quotientenring zu einer K-Algebra.

Der nächste Satz gibt Aufschluss darüber, wie man eine Basis von P_n/I findet.

Satz 12. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei (N, E, S) die Zerlegung von I bezüglich $<$. Dann ist*

$$B = \{b + I \mid b \in N\}$$

eine K-Basis von P_n/I als K-Vektorraum.

Beweis.
Zunächst ist zu bemerken, dass offensichtlich $b + I \neq I$ für $b \in N$. Ferner ist $b_1 + I \neq b_2 + I$ für $b_1, b_2 \in N$ mit $b_1 \neq b_2$, denn sonst wäre $b_1 - b_2 \in I$, was im Widerspruch zu $b_1, b_2 \in N$ steht. Mit diesem Argument sieht man auch, dass B linear unabhängig ist: Sei $\sum(\lambda_i b_i + I) = I$, d.h. $(\sum \lambda_i b_i) + I = I$, d.h. $\sum \lambda_i b_i \in I$. Sind nicht alle λ_i gleich 0, so ist $\mathrm{L}(\sum \lambda_i b_i) = b_l$ für ein l. Widerspruch zu $b_i \in N$, also ist B linear unabhängig.
Bleibt noch zu zeigen, dass B ein Erzeugendensystem ist. Dazu sei $X \in P_n/I$ eine Restklasse mit $X \neq I$. Sei $f \in P_n$ ein Repräsentant dieser Restklasse mit minimalem Leitterm. (Diesen gibt es, da $<$ eine Wohlordnung ist.) Dann ist $\mathrm{L}(f) \in N$: Wäre $\mathrm{L}(f) \notin N$, so gäbe es ein $p \in I$ mit $\mathrm{L}(p) \mid \mathrm{L}(f)$. Dann könnte man setzen

$$\widetilde{f} = f - \frac{\mathrm{c}(f)}{\mathrm{c}(p)} \frac{\mathrm{L}(f)}{\mathrm{L}(p)} \cdot p\ ,$$

es wäre $\mathrm{L}(\widetilde{f}) < \mathrm{L}(f)$ (insbesondere ist $\widetilde{f} \neq 0$, denn sonst wäre $f \in I$), und es gälte

$$X = f + I = \left(\widetilde{f} + \frac{\mathrm{c}(f)}{\mathrm{c}(p)} \frac{\mathrm{L}(f)}{\mathrm{L}(p)} \cdot p\right) + I = \widetilde{f} + I\ .$$

Dies ist jedoch ein Widerspruch zu der Minimalität von f. Also ist $\mathrm{L}(f) \in N$. Demnach hat f die Gestalt $f = \lambda_1 b_1 + \mathrm{tail}(f)$ mit $b_1 \in N$ und somit gilt

$$X = f + I = (\lambda_1 b_1 + I) + (\mathrm{tail}(f) + I)\ .$$

Ist $\mathrm{tail}(f) = 0$, so ist man fertig. Ist $\mathrm{tail}(f) \neq 0$, so ist natürlich $\mathrm{L}(\mathrm{tail}(f)) < \mathrm{L}(f)$. Es bleibt also nur noch zu zeigen, dass $X' = \mathrm{tail}(f) + I$ von B erzeugt wird. Verfährt man nun mit X' genauso wie mit X, so gelangt man schließlich nach endlich vielen Schritten – denn $<$ ist ja eine Wohlordnung und der Leitterm von $\mathrm{tail}(f)$ ist echt kleiner als der Leitterm von f – zu

$$X = f + I = (\lambda_1 b_1 + I) + (\mathrm{tail}(f) + I) = \cdots = (\lambda_1 b_1 + I) + \cdots + (\lambda_d b_d + I)\ .$$

Also ist B auch ein Erzeugendensystem von P_n/I. □

Der Satz liefert als Korollar

Korollar. *Unter den Voraussetzungen von Satz 12 gilt:*

$$P_n/I \cong \operatorname{span}_K(N) .$$

Beweis.
Alles schon gezeigt.

Jede Restklasse in P_n/I hat nach dem Satz also einen ausgezeichneten Repräsentanten, wenn man sich einmal auf eine Ordnung geeinigt hat. Dieser verdient eine eigene Bezeichnung.

Definition. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei (N,E,S) die Zerlegung von I bezüglich $<$. Dann sei die Abbildung $\mathrm{NF} : P_n \to P_n$ wie folgt definiert: Sei $f \in P_n$. Es existieren (nach Satz 12) eindeutig bestimmte $b_1,\ldots,b_d \in N$ und $\lambda_1,\ldots,\lambda_d \in K$, so dass*

$$f + I = \sum \lambda_i b_i + I .$$

Dann setze

$$\mathrm{NF}(f) := \sum \lambda_i b_i .$$

*Man nennt $\mathrm{NF}(f)$ die **Normalform** von f bezüglich I und der Ordnung $<$.*

Wenn nicht eindeutig hervorgeht, bezüglich welcher Ordnung die Normalform gemeint ist, schreibt man auch $\mathrm{NF}_<(f)$. Wenn es auch noch Zweifel über das Ideal gibt, so wird auch dies vermerkt, indem man $\mathrm{NF}_<(f,I)$ schreibt.

Beispiel
Betrachte wieder das Beispiel von Seite 22 bezüglich der lexikografischen Ordnung. Sei also $I = \langle f_1, f_2, f_3 \rangle \subseteq K[x,y]$ mit

$$f_1 = y^5 + y^4,\ f_2 = xy^4 - y^4,\ f_3 = x^2y + y^4 .$$

Die zugehörige Zerlegung (N,E,S) zeigte Bild 3. Berechne die Normalform von

$$f = x^2y^5 + 2xy^6 + xy^5 - 2y^6,\ g = x^4 - x^2y^2 + 2xy^2,\ h = x^3 + xy^3 + y^3 .$$

Es ist

$$f = x \cdot f_1 + (2y^2 - x) \cdot f_2,\ g = (x^4 + 2xy^2 - y^4) + f_1 - y \cdot f_3 ,$$

also ist $f + I = 0 + I$ und $g + I = (x^4 + 2xy^2 - y^4) + I$. Ferner ist $x^4, xy^2, y^4 \in N$ (vergleiche Bild 3). Für die Normalform von h beachte man, dass $x^3, xy^3, y^3 \in N$, d.h. es ist $h \in \operatorname{span}_K(N)$. Also gilt insgesamt:

$$\mathrm{NF}(f) = 0,\ \mathrm{NF}(g) = x^4 + 2xy^2 - y^4,\ \mathrm{NF}(h) = h .$$

Beachte insbesondere, dass $f \in I$ und $\mathrm{NF}(f) = 0$ sowie $h \in \operatorname{span}_K(N)$ und $\mathrm{NF}(h) = h$.

Die Normalform ist ein zentrales Instrument z.B. bei der Berechnung von Standardbasen. Will man mit dem Buchberger-Algorithmus arbeiten, so wird man die Normalform auf eine andere Art einführen, üblicherweise über den Algorithmus, mit dem man die Normalform berechnen kann. Dieser behandelt den Begriff etwas allgemeiner, da er die Normalform eines Polynoms nicht nur bezüglich eines Ideals zulässt, sondern auch bezüglich einer (beliebigen)

Menge von Polynomen. Bezüglich Standardbasen berechnet der Algorithmus die Normalform in Übereinstimmung mit obiger Definition. Siehe hierzu Kapitel 4.

Der algorithmische Zugang zur Normalform hat im Rahmen des Buchberger-Algorithmus den Vorteil, dass man mit seiner Hilfe ein sehr schönes Kriterium zur Verfügung hat, ob ein Erzeugendensystem eine Standardbasis ist oder nicht (vergleiche Satz 21 aus Kapitel 4). Im Rahmen dieser Arbeit erschien jedoch der Zugang über den Quotientenring sinnvoller, da hier der Abbildungscharakter besser hervortritt. Außerdem lassen sich damit einige Aussagen recht einfach beweisen. Die Arbeit, die man zu leisten hat (und die im Prinzip der Algorithmus Normalform leistet), steckt im Nachweis, dass N eine Basis des Quotientenrings bildet.

Nun aber einige nützliche Eigenschaften der Normalform. Auch wenn sie teilweise trivial sind, sollen sie hier doch explizit formuliert werden, da sie für den weiteren Verlauf von großer Bedeutung sind.

Lemma 13. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei* NF *die Normalform bezüglich I und $<$. Sei (N, E, S) die Zerlegung von I. Dann gilt:*

a) Ist $f \in P_n$, so ist $f \in I$ genau dann, wenn $\mathrm{NF}(f) = 0$.

b) Ist $m \in M_n$, so ist $m \in N$ genau dann, wenn $\mathrm{NF}(m) = m$.

c) $\mathrm{L}(f) \geq \mathrm{L}(\mathrm{NF}(f))$ für $f \in P_n \setminus I$.

Beweis.

a) und *b*) sind offensichtlich.

zu *c*): Da $f - \mathrm{NF}(f) \in I$, ist $\mathrm{L}(f - \mathrm{NF}(f)) \in \mathrm{L}(I)$. Wäre nun $\mathrm{L}(f) < \mathrm{L}(\mathrm{NF}(f))$, so wäre $\mathrm{L}(f - \mathrm{NF}(f)) = \mathrm{L}(\mathrm{NF}(f)) \in \mathrm{L}(I)$. Widerspruch zu $\mathrm{L}(\mathrm{NF}(f)) \in N$. □

Wie der nächste Satz zeigt, ist die Normalform K-linear. Die zweite Aussage des Satzes wird später bei der Formulierung des FGLM-Algorithmus sehr nützlich sein.

Satz 14. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei* NF *die Normalform bezüglich I und $<$. Dann gilt:*

$$\mathrm{NF}(\alpha p + \beta q) = \alpha\,\mathrm{NF}(p) + \beta\,\mathrm{NF}(q) \qquad \text{für } \alpha, \beta \in K \text{ und } p, q \in P_n$$

und

$$\mathrm{NF}(p \cdot q) = \mathrm{NF}(\mathrm{NF}(p)\,\mathrm{NF}(q)) \qquad \text{für } p, q \in P_n\,.$$

Beweis.

Sei $\mathrm{NF}(p) = \sum \rho_i b_i$, $\mathrm{NF}(q) = \sum \sigma_i b_i$ und $\mathrm{NF}(\alpha p + \beta q) = \sum \tau_i b_i$ mit $\rho_i, \sigma_i, \tau_i \in K$, $b_i \in N$ für $i = 1, \ldots, d$. Dann ist

$$\begin{aligned}\sum \tau_i b_i + I &= \mathrm{NF}(\alpha p + \beta q) + I = (\alpha p + \beta q) + I = (\alpha p + I) + (\beta q + I)\\ &= (\alpha\,\mathrm{NF}(p) + I) + (\beta\,\mathrm{NF}(q) + I) = \alpha\left(\sum \rho_i b_i + I\right) + \beta\left(\sum \sigma_i b_i + I\right)\\ &= \sum(\alpha\rho_i + \beta\sigma_i) b_i + I\,.\end{aligned}$$

Da N eine Basis von P_n/I ist, folgt $\tau_i = \alpha\rho_i + \beta\sigma_i$, also:

$$\begin{aligned}\mathrm{NF}(\alpha p + \beta q) &= \sum \tau_i b_i = \sum(\alpha\rho_i + \beta\sigma_i) b_i\\ &= \alpha\left(\sum \rho_i b_i\right) + \beta\left(\sum \sigma_i b_i\right) = \alpha\,\mathrm{NF}(p) + \beta\,\mathrm{NF}(q)\,.\end{aligned}$$

Außerdem gilt:

$$\begin{aligned}\mathrm{NF}(p \cdot q) + I &= p \cdot q + I = (p + I)\,(q + I) = (\mathrm{NF}(p) + I)\,(\mathrm{NF}(q) + I)\\ &= \mathrm{NF}(p)\,\mathrm{NF}(q) + I = \mathrm{NF}(\mathrm{NF}(p)\,\mathrm{NF}(q)) + I\,.\end{aligned}$$

Damit folgt auch die zweite Behauptung, da die Normalform ein eindeutiger Repräsentant der Restklasse ist. □

Das nulldimensionale Ideal

Satz 12 hat gezeigt, dass der Quotientenring P_n/I genau dann von endlicher Dimension ist, wenn die Menge N der nicht reduzierbaren Monome bezüglich einer Monomordnung endlich ist. Da die Dimension des Quotientenrings aber nicht von der Ordnung abhängt, hängt auch die Tatsache, ob N endlich ist oder nicht, von der gewählten Ordnung nicht ab. Ist N bezüglich einer Wohlordnung endlich, so auch bezüglich jeder anderen. Der Satz sagt sogar noch mehr: Ist N endlich, so hängt $|\,N\,|$ nicht von der gewählten Ordnung ab (N selbst hängt natürlich sehr wohl von der Ordnung ab).

Der FGLM-Algorithmus in seiner ursprünglichen Form ist nur für solche Ideale formuliert, für die der Quotientenring von endlicher Dimension ist. Solche Ideale sollen fortan *nulldimensional* heißen:

Definition. *Sei $I \subseteq P_n$ ein Ideal. Ist P_n/I als K-Vektorraum von endlicher Dimension, so heißt I* ***nulldimensional****.*
Ist I nulldimensional, so setze

$$\mathrm{vdim}(I) := \dim_K(\,P_n/I\,)\ .$$

Es ist also $\mathrm{vdim}(I) = |\,N\,|$. Verwirrend erscheint auf den ersten Blick die Namensgebung „nulldimensional"; intuitiver wäre wohl „von endlicher Dimension" gewesen. Doch der folgende Satz rechtfertigt die Bezeichnung: Die von nulldimensionalen Idealen erzeugten affinen Varietäten sind endliche Mengen im $\mathbb{C}^n$, also gerade nulldimensionale Varietäten. Auf den Begriff einer Varietät soll im Rahmen dieser Arbeit nicht näher eingegangen. Nur so viel: Sind $f_1,\dots,f_s \in P_n$, so bezeichnet man

$$\mathbf{V}(f_1,\dots,f_s) := \{a \in K^n \mid f_1(a) = \cdots = f_s(a) = 0\}$$

als eine *affine Varietät*. Ist ferner $I = \langle\, g_1,\dots,g_t\,\rangle \subseteq P_n$ ein Ideal, so setzt man

$$\mathbf{V}(I) := \{a \in K^n \mid f(a) = 0 \text{ für alle } f \in I\}\ .$$

Dann zeigt man, dass $\mathbf{V}(I) = \mathbf{V}(g_1,\dots,g_t)$, also $\mathbf{V}(I)$ ebenfalls eine affine Varietät ist. Sie enthält somit die Punkte, an denen alle Polynome des Ideals gleichzeitig verschwinden.

Satz 15. *Sei $I \subseteq K[x_1,\dots,x_n]$ ein Ideal. Dann sind äquivalent:*

(i) *I ist nulldimensional, d.h. $K[x_1,\dots,x_n]/I$ ist von endlicher Dimension.*
(ii) *Ist $<$ eine Wohlordnung, so existiert für jedes x_i ein $k \in \omega$, so dass $x_i^k \in \mathrm{L}(I)$.*
(iii) *Ist $<$ eine Wohlordnung und F eine Gröbnerbasis von I für $<$, so existiert für jedes x_i ein $k \in \omega$, so dass $x_i^k \in \mathrm{L}(F)$.*
(iv) *Für jedes x_i ist $I \cap K[x_i] \neq \{0\}$.*
(v) *Ist $<$ eine Wohlordnung, so ist $\mathrm{span}_K(x^\alpha \mid x^\alpha \notin \mathrm{L}(I))$ als K-Vektorraum von endlicher Dimension.*
(vi) *Es existiert ein endliches definierendes System $\mathcal{L} : K[x_1,\dots,x_n] \to K^d$ für I.*

Ist darüberhinaus $K = \mathbb{C}$, so sind (i) bis (vi) äquivalent zu

(vii) *Die Varietät $\mathbf{V}(I) \subset \mathbb{C}^n$ ist eine endliche Menge.*

Beweis.
$(i) \Rightarrow (ii)$: Sei $<$ eine Wohlordnung, sei (N,E,S) die Zerlegung. Da $N+I$ eine Basis von P_n/I ist, ist N endlich. Also gibt es ein k, so dass $x_i^k \notin N$, also $x_i^k \in \mathrm{L}(I)$.
$(ii) \Rightarrow (iii)$: Sei $x_i^k \in \mathrm{L}(I)$. Dann ist $x_i^k \in \langle\, \mathrm{L}(I)\,\rangle = \langle\, \mathrm{L}(F)\,\rangle$. Also gibt es ein $f \in \mathrm{L}(F)$ mit $\mathrm{L}(f) \mid x_i^k$. Also ist $\mathrm{L}(f) = x_i^l$ für ein $l \leq k$.
$(iii) \Rightarrow (iv)$: Wähle als Ordnung eine lexikografische Ordnung, wobei die Variablen x_j so angeordnet seien, dass $x_i < x_j$ für alle $j \neq i$. Sei $F \subset I$ bezüglich dieser Ordnung eine Gröbnerbasis. Dann ist also $x_i^k \in \mathrm{L}(F)$, d.h. es gibt ein $f \in F$ mit $\mathrm{L}(f) = x_i^k$. Nach Wahl der Ordnung und speziell nach Anordnung der Variablen folgt $f \in K[x_i]$, also $f \in I \cap K[x_i]$.
$(iv) \Rightarrow (i)$: Sei (N,E,S) die Zerlegung von I bezüglich einer Wohlordnung $<$. Sei $f_i \in I$, so dass $f_i \in I \cap K[x_i]$. Dann ist offensichtlich $\mathrm{L}(f_i) = x_i^{k_i}$ für ein $k_i \in \omega$, da f_i univariat ist. Also ist $x_i^{k_i} \notin N$ für alle i. Dann ist aber $|\,N\,| \leq \prod k_i$, d.h. $|\,N\,|$ ist endlich, und da N eine Basis von $K[x_1,\ldots,x_n]/I$ ist, folgt, dass I nulldimensional ist.
$(i) \Leftrightarrow (v)$: Folgt aus dem Korollar zu Satz 12.
$(i) \Leftrightarrow (vi)$: Wird in Kapitel 5 gezeigt (Satz 23).
$(i) \Leftrightarrow (vii)$: Führt hier zu weit vom Thema ab. Siehe Cox, Little, O'Shea [8], Kapitel 5, §3. □

Die verschiedenen Bedingungen sollen an einem Beispiel verdeutlicht werden.

Beispiel
Betrachte das Ideal $I = \langle\, xy^3 - x^2,\ x^3y^2 - y\,\rangle \subset K[x,y]$. Bezüglich der grad-lexikografischen Ordnung ist

$$F_{deglex} = \left\{x^3y^2 - y,\ x^4 - y^2,\ xy^3 - x^2,\ y^4 - xy\right\}$$

eine reduzierte Gröbnerbasis von I. Also ist

$$\langle\, \mathrm{L}(I)\,\rangle = \langle\, x^3y^2,\ x^4,\ xy^3,\ y^4\,\rangle\ .$$

Man sieht, dass $x^4, y^4 \in \mathrm{L}(F_{deglex})$, also ist I nulldimensional nach Bedingung (iii). Die Menge der nicht reduzierbaren Monome zeigt Bild 5.

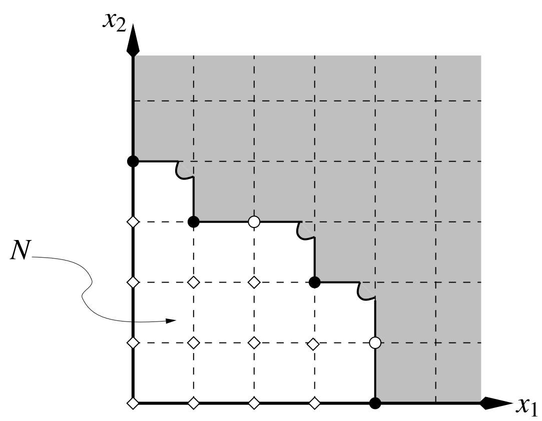


Bild 5: Die Zerlegung von I bezüglich $<_{deglex}$.

Wie Satz 15 verspricht, ist N endlich, also ist $\mathrm{vdim}(I) = |\,N\,| = 12$. Damit ist dann

$$K[x,y]/I \cong \mathrm{span}_K(1,\ x,\ x^2,\ x^3,\ y,\ xy,\ x^2y,\ x^3y,\ y^2,\ xy^2,\ x^2y^2,\ y^3)\ .$$

Um Bedingung (*iv*) zu überprüfen, stellt man fest, dass

$$x^{12} - x^2 \in I \quad \text{und} \quad y^{11} - y \in I\ ,$$

denn wie man leicht nachrechnet, ist

$$\begin{aligned} x^{12} - x^2 &= (x^8 + x^4y^2 + y^4) \cdot (x^4 - y^2) + 1 \cdot (xy^3 - x^2) + y^2 \cdot (y^4 - xy)\ , \\ y^{11} - y &= 1 \cdot (x^3y^2 - y) + (y^5 + xy^2) \cdot (xy^3 - x^2) + y^7 \cdot (y^4 - xy)\ . \end{aligned}$$

In Kapitel 6 wird ein Algorithmus vorgestellt, der im nulldimensionalen Fall für jede Variable x_i ein $f_i \in K[x_i]$ berechnet, so dass $I \cap K[x_i] = \langle f_i \rangle \subseteq K[x_i]$.

Betrachtet man dieses Beispiel bezüglich der lexikografischen Ordnung, so ist

$$F_{lex} = \{y^{11} - y,\ xy - y^4,\ x^2 - y^6\}$$

eine Gröbnerbasis, und man überzeugt sich leicht, dass $|\,N\,| = 12$ auch in diesem Fall (obwohl N selbst eine gänzlich andere Gestalt hat).

Die Hilbertfunktion

Die Hilbertfunktion verwendet man in der algebraischen Geometrie, um einer Varietät (sowohl affin als auch projektiv) einen Dimensionsbegriff zuzuordnen. Hier zeigt sich die schöne Symmetrie zwischen Algebra, d.h. Idealen, und Geometrie, d.h. Varietäten, da man den geometrischen Dimensionsbegriff zufriedenstellend über Eigenschaften von Idealen definieren kann.

Die algebraischen Eigenschaften der Hilbertfunktion ermöglichen es, sie auch als Hilfsmittel bei der Berechnung von Gröbnerbasen einzusetzen. Zum einen kann man sie direkt im Buchbergeralgorithmus verwenden, man erhält dann den sogenannten *hilbertgetriebenen Buchberger-Algorithmus*. Zum anderen leistet sie aber auch gute Dienste, wenn man den FGLM-Algorithmus auf eine größere Klasse von Idealen erweitern möchte, wie in Kapitel 6 erklärt wird.

Wie zu Beginn dieses Kapitels gesehen, ist der Polynomring aufgefasst als Vektorraum über K von unendlicher Dimension. Dieses Handikap kann man beseitigen, indem man nicht mehr den gesamten Polynomring betrachtet, sondern nur noch jene Polynome, die einen gewissen Grad nicht überschreiten. Folgende Notation vereinfacht die Schreibweise dabei.

Notation. *Sei $s \in \omega$. Dann setze*

$$\begin{aligned} M_{\leq s} &:= \{m \in M_n \mid \operatorname{grad}(m) \leq s\}\ , \\ P_{\leq s} &:= \{p \in P_n \mid \operatorname{grad}(p) \leq s\}\ . \end{aligned}$$

Ist ferner $I \subset P_n$ ein Ideal mit $I \neq P_n$, so setze

$$I_{\leq s} := \{f \in I \mid \operatorname{grad}(f) \leq s\}\ .$$

Zur Erinnerung: Der Grad eines Polynoms war definiert als der maximale Grad seiner Monome. Man sieht direkt, dass $M_{\leq s}$ eine endliche K-Vektorraumbasis von $P_{\leq s}$ bildet. Ebenso leicht sieht man, dass $I_{\leq s}$ ein Teilraum von $P_{\leq s}$ ist und damit ebenfalls als Vektorraum von endlicher Dimension ist. Allerdings ist $I_{\leq s}$ kein Ideal mehr, da es weder additiv noch multiplikativ abgeschlossen ist. Mit etwas Kombinatorik kann man die Anzahl der Monome in $M_{\leq s}$ zählen und erhält:

$$|\,M_{\leq s}\,| = \binom{n+s}{s}\ .$$

Da $I_{\leq s}$ ein Unterraum ist, kann man den Quotienten bilden und erhält wieder einen Vektorraum endlicher Dimension. Und gerade diese Dimension misst nun die Hilbertfunktion.

Definition. *Sei $I \subset P_n$ ein Ideal. Dann heißt*

$$\mathrm{Hf}_I : \omega \to \omega \quad \text{vermöge} \quad \mathrm{Hf}_I(s) = \dim_K(P_{\leq s} \,/\, I_{\leq s})$$

*die **affine Hilbertfunktion** von I. Dabei sei $\dim_K$ die Vektorraumdimension über K.*

Da hier nur die affine und nicht die projektive Hilbertfunktion zur Verwendung kommt, wird im Folgenden der Zusatz *affin* einfach weggelassen.

Betrachtet man die Hilbertfunktion eines monomiales Ideals, so erhält man eine sehr schöne anschauliche Bedeutung.

Satz 16. *Sei $I \subset P_n$ ein monomiales Ideal. Dann gilt: Für alle $s \in \omega$ ist $\mathrm{Hf}_I(s)$ die Anzahl der Monome vom Grad kleiner oder gleich s, die nicht in I liegen.*
Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 9, §3.

Diese Anschauung geht natürlich verloren, wenn das Ideal nicht mehr monomial ist. So enthält etwa das Ideal $I = \langle\, x + 1,\, y + 1 \,\rangle \subset K[x, y]$ kein einziges Monom, obwohl die Hilbertfunktion nicht obigem Binomialkoeffizienten genügt. Es besteht aber dennoch ein Zusammenhang, wenn man die richtige Ordnung wählt. Wie der nächste Satz zeigt, entspricht dann die Hilbertfunktion gerade der Anzahl der Monome bis zu einem gewissen Grad, die nicht im Leitideal liegen.

Satz 17. *Sei $I \subset P_n$ ein Ideal und sei $<$ eine gradorientierte Wohlordnung. Dann haben I und $\langle\, \mathrm{L}(I) \,\rangle$ die gleiche affine Hilbertfunktion.*
Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 9, §3.

Wenn man den Beweis zu Satz 17 genauer untersucht, so stellt man fest, dass man auf die Voraussetzung einer gradorientierten Ordnung verzichten kann, wenn man die Hilbertfunktion eines homogenen Ideals berechnen möchte. Denn als wichtige Eigenschaft der Gradordnung wurde im Beweis ausgenutzt, dass der Grad des Leitmonoms gleich dem Grad des Polynoms ist. Diese Eigenschaft haben aber homogene Polynome in jedem Fall, unabhängig von der gewählten Wohlordnung, so dass man mit geringfügigen Änderungen im Beweis sofort auch folgende Eigenschaft zur Verfügung hat:

Satz 18. *Sei $I \subset P_n$ ein homogenes Ideal, sei $<$ eine Wohlordnung. Dann haben I und $\langle\, \mathrm{L}(I) \,\rangle$ die gleiche affine Hilbertfunktion.*
Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 9, §3.

Bei einer näheren Untersuchung erkennt man, dass sich die Hilbertfunktion für grosse Werte wie ein Polynom verhält, da sich die Anzahl der Monome, die nicht in einem monomialen Ideal liegen, ab einem bestimmten Grad ebenfalls polynomial verändert. Dieses Polynom wird dann entsprechend *Hilbertpolynom* genannt.

Satz 19. *Sei $I \subset P_n$ ein Ideal. Ist $s \in \omega$ hinreichend groß, so ist*

$$\mathrm{Hf}_I(s) = \sum_{i=0}^{d} b_i \binom{s}{d - i}$$

mit $d \in \omega$, $b_0 \in \mathbb{N}$ und $b_1, \ldots, b_d \in \mathbb{Z}$.
Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 9, §3.

Definition. *Sei $I \subset P_n$ ein Ideal. Sei $d \in \omega$, $b_0 \in \mathbb{N}$ und $b_1, \ldots, b_d \in \mathbb{Z}$. Sei $s_0 \in \omega$ minimal gewählt, so dass*

$$\mathrm{Hf}_I(s) = \sum_{i=0}^{d} b_i \binom{s}{d-i} \quad \textit{für } s \geq s_0 \ .$$

Dann heißt

$$\mathrm{Hp}_I(s) : \omega \to \omega \quad \text{vermöge} \quad \mathrm{Hp}_I(s) = \sum_{i=0}^{d} b_i \binom{s}{d-i}$$

*das **affine Hilbertpolynom** von I und s_0 der **Regularitätsindex**.*

Die Dimension einer Varietät $V \subseteq K^n$ wird dann als der Grad des Hilbertpolynoms der Hilbertfunktion von $\mathbf{I}(V)$ definiert. Dies aber nur der Vollständigkeit halber, der Dimensionsbegriff wird für das Folgende nicht benötigt.

„Besonders fortgeschrittene Programmierer werden den großen Speicherraum des Commodore 64 zu schätzen wissen. Mit seinen 64 K RAM und 20 K ROM bietet der Commodore 64 genügend Speicherreserven auch für das anspruchsvollste Programm.“

Angermann-Brückmann-Englisch-Gerits, 64 intern, Düsseldorf 1983

Der Buchberger-Algorithmus

In diesem Kapitel soll eine kurze Einführung in den Buchberger-Algorithmus gegeben werden, nicht zuletzt, um die unterschiedlichen Vorgehensweisen im Vergleich zum FGLM-Algorithmus zu verdeutlichen. Hier ist kein Anspruch auf Vollständigkeit gelegt, die Sätze werden ohne Beweis erfolgen, und der Algorithmus selbst taucht nur in seiner Grundvariante, nicht aber mit seinen zahlreichen Verfeinerungen auf. Ebenso zahlreich ist auch die Literatur zu diesem Thema, auf die an dieser Stelle verwiesen sei.

Der erste Schritt auf dem Weg zum Buchberger-Algorithmus besteht darin, die Polynomdivision, die man im univariaten Polynomring $K[x]$ zur Verfügung hat, auf den multivariaten Fall zu übertragen. Dies geht jedoch nicht ohne Verluste vonstatten: Teilt man ein $f \in P_n$ durch $f_1, \ldots, f_s \in P_n$, so hängt der Rest der Division nicht nur von der verwendeten Ordnung, sondern im Allgemeinen auch von der Reihenfolge der f_i ab, ist also insbesondere nicht eindeutig bestimmt.

Teilt man jedoch durch eine Gröbnerbasis, so erhält man bei der Division wieder einen eindeutig bestimmten Rest. Sieht man sich die Eigenschaften dieses Restes an, so erkennt man eine enge Verwandtschaft zur Normalform, wie sie in Kapitel 3 vorgestellt wurde. Dies ist kein Zufall, denn die Normalform eines Polynoms bezüglich eines Ideal ist wirklich gerade der Rest bei der Division durch eine Gröbnerbasis dieses Ideals. Und diesen kann man algorithmisch berechnen.

Definition. *Seien $p, q \in P_n \setminus \{0\}$, sei $<$ eine Wohlordnung und sei $m = \mathrm{kgV}(\mathrm{L}(p), \mathrm{L}(q))$. Dann heißt*

$$\mathrm{spoly}_<(p,q) = \frac{m}{\mathrm{L}(p)} \cdot p - \frac{\mathrm{c}(p)}{\mathrm{c}(q)} \frac{m}{\mathrm{L}(q)} \cdot q$$

*das **S-Polynom** von p und q.*

S-Polynome sind so gebaut, dass sich die Leitterme der beiden involvierten Polynome gerade wegheben. Der folgende Algorithmus **lnf** (Algorithmus 1) ist ein erster Schritt zur vollständigen Division. Er teilt ein Polynom so lange durch Polynome $f_1, \ldots, f_s$, bis dessen Leitterm nicht mehr durch einen Leitterm der f_i teilbar ist. Ein Divisionschritt erfolgt dabei durch S-Polynome. Da dieser Algorithmus schon so schnell aufgibt, erhält er den Zusatz „**l**“ (für *lazy*) in seinem Namen.

Damit ist ein Polynom p aber noch nicht vollständig reduziert, man muss mit $\mathrm{tail}(p)$ noch weiterarbeiten. Diese Aufgabe erledigt der Algorithmus **nf**, der etwas fleißiger ist: Wenn der Leitterm des Polynoms nicht mehr reduzierbar ist, wird dieser zum Rest geschrieben, und es wird $\mathrm{tail}(p)$ weiterreduziert.

Wie der Name schon verrät und wie oben auch angedeutet, berechnet der Algorithmus **nf** tatsächlich gerade die Normalform. Mit anderen Worten: Die Normalform eines Polynoms ist nichts anderes als der Rest bei Division durch eine Gröbnerbasis.

Algorithmus 1: lnf.
Eingabe: *Sei $p \in P_n$ und seien $f_1,\ldots,f_s \in P_n \setminus \{0\}$, sei $<$ eine Wohlordnung.*
Ausgabe: *Ein $h \in P_n$, so dass es $q_i \in P_n$ gibt mit:*
a) $h = p - \sum q_i f_i$
b) *Ist* $q_i f_i \neq 0$, *so ist* $\operatorname{multideg}(q_i f_i) \leq \operatorname{multideg}(p)$.
c) $\mathrm{L}(h)$ *wird von keinem* $\mathrm{L}(f_i)$ *geteilt.*
Algorithmus:

$$
\begin{aligned}
&(h) = \mathbf{lnf}(p, \{f_1, \ldots, f_s\}, <) \\
&\quad h \leftarrow p \\
&\quad \textbf{while } (h \neq 0) \textbf{ and } (\exists\, i \mid \mathrm{L}(f_i) \mid \mathrm{L}(h)) \\
&\quad\quad j \leftarrow \min(i \in \{1, \ldots, s\} \mid \mathrm{L}(f_i) \mid \mathrm{L}(h)) \\
&\quad\quad h \leftarrow \operatorname{spoly}(h, f_i) \\
&\quad \textbf{return } (h)\ \bullet
\end{aligned}
$$

Algorithmus 2: nf.
Eingabe: *Sei $p \in P_n$ und seien $f_1,\ldots,f_s \in P_n \setminus \{0\}$, sei $<$ eine Wohlordnung.*
Ausgabe: *Ein $h \in P_n$, so dass es $q_i \in P_n$ gibt mit:*
a) $h = p - \sum q_i f_i$
b) *Ist* $q_i f_i \neq 0$, *so ist* $\operatorname{multideg}(q_i f_i) \leq \operatorname{multideg}(p)$.
c) *Kein Monom von h wird von einem* $\mathrm{L}(f_i)$ *geteilt.*
Algorithmus:

$$
\begin{aligned}
&(h) = \mathbf{nf}(p, \{f_1, \ldots, f_s\}, <) \\
&\quad h \leftarrow \mathbf{lnf}(p, \{f_1, \ldots, f_s\}, <) \\
&\quad \textbf{if } (h \neq 0) \textbf{ and } (\operatorname{tail}(h) \neq 0) \\
&\quad\quad \operatorname{tail}(h) \leftarrow \mathbf{nf}(\operatorname{tail}(h), \{f_1, \ldots, f_s\}, <) \\
&\quad \textbf{return } (h)\ \bullet
\end{aligned}
$$

Satz 20. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei $F = \{f_1, \ldots, f_s\} \subset I$ eine Gröbnerbasis von I bezüglich $<$. Dann ist $\mathbf{nf}(p, F, <) = \mathrm{NF}_<(p)$ für alle $p \in P_n$.*

Beweis.
Sei $h = \mathbf{nf}(p, F, <)$. Nach Eigenschaft *c)* des Algorithmus **nf** ist $h \in \operatorname{span}_K(N)$, da F eine Gröbnerbasis ist. Nach Eigenschaft *a)* ist $p + I = h + I$. Die Aussage des Satzes folgt dann mit Satz 12 aus Kapitel 3 und der Definition der Normalform. □

Sei $I \subset P_n$ ein Ideal und sei $F \subset I$. Ist dann $p \in P_n$ mit $\mathbf{nf}(p, F, <) = 0$, so ist also $p \in I$, nach den im Algorithmus behaupteten Eigenschaften von **nf**. Leider gilt die Umkehrung nicht (es sei denn, es handelt sich bei F um eine Gröbnerbasis). Ist aber $p \in I$, so ist wenigstens auch $\mathbf{nf}(p, F, <) \in I$.

Hinter der Aussage des nächsten Satzes verbirgt sich nun die zentrale Idee des Buchberger-Algorithmus. Der Satz behauptet, dass genau dann eine Gröbnerbasis vorliegt, wenn sich die paarweisen S-Polynome des Erzeugendensystems zu Null reduzieren. Reduziert sich ein S-Polynom nicht zu Null, liegt also noch keine Gröbnerbasis vor, so kann man den Divisionsrest wie gesehen einfach zum Erzeugendensystem hinzupacken, so dass sich nun natürlich das

S-Polynom bezüglich dieser neuen Basis zu Null reduziert. Wiederholt man dies so lange, bis sich alle S-Polynome zu Null reduzieren, so hat man eine Gröbnerbasis bekommen und den Buchberger-Algorithmus entdeckt.

Satz 21. *Sei $I \subseteq P_n$ ein Ideal und sei $F = \{f_1, \ldots, f_s\} \subset I$. Sei ferner $<$ eine Wohlordnung. Dann ist F genau dann eine Gröbnerbasis von I, wenn gilt*

$$\mathbf{nf}(\mathrm{spoly}_<(f,g), F, <) = 0 \qquad \textit{für alle } f,\, g \in F \textit{ mit } f \neq g\ .$$

Beweis.
Siehe Cox, Little, O'Shea [8], Kapitel 2, §9.

Der Algorithmus formuliert sich dann wie folgt:

Algorithmus 3: buchberger.
Eingabe: *Seien $p_1, \ldots, p_t \in P_n \setminus \{0\}$ und sei $<$ eine Wohlordnung.*
Ausgabe: *Ein $F \subset \langle\, p_1, \ldots, p_t \,\rangle$, so dass F Gröbnerbasis von $\langle\, p_1, \ldots, p_t \,\rangle$ bezüglich $<$ ist.*
Algorithmus:

$(F) = \mathbf{buchberger}(p_1, \ldots, p_t, <)$

- $F \leftarrow \{p_1, \ldots, p_t\}$
- $P \leftarrow \{(p_i\,, p_j) \mid i < j\}$
- **while** $(P \neq \{\})$
  - **choose** $(p\,, q) \in P$
  - $P \leftarrow P \setminus \{(p\,, q)\}$
  - $h \leftarrow \mathbf{nf}(\mathrm{spoly}(p, q), F)$
  - **if** $(h \neq 0)$
    - $P \leftarrow P \cup \{(p\,, f) \mid f \in F\}$
    - $F \leftarrow F \cup \{h\}$
- **return** (F) •

Auch wenn der Algorithmus so harmlos aussieht, kann die Ausführung eine aufwendige und langwierige Sache werden. Zum einen kann die Tupelmenge der noch zu bearbeitenden S-Polynome sehr groß werden, zum anderen ist Normalformen-Berechnen eine meist teure Arbeit. Um die Effizienz zu steigern, wird mal also als erstes versuchen, diese Tupelmenge zu verkleinern, sich also Kriterien zu überlegen, die a priori sicherstellen, dass sich ein S-Polynom zu Null reduziert, ohne dass man dies nachrechnen muss.

Es hat mittlerweile eine ganze Reihe von Verbesserungen des Buchberger-Algorithmus gegeben, und die in SINGULAR implementierte Variante zählt zu den schnellsten auf dem Markt. Nichtsdestotrotz gibt es aber immer wieder Beispiele, vor allem wenn man bezüglich der so nützlichen lexikografischen Ordnung arbeitet, die sich mit dem Buchberger-Algorithmus einfach nicht mehr berechnen lassen, sowohl aus Speicher- als auch aus Zeitgründen. Hinzu kommt auch noch, dass eine gegebene Basis es sich im Allgemeinen nicht ansehen lässt, ob ihre Gröbnerbasenberechnung aufwendig ist oder nicht.

So liegt es nahe, sich nach anderen Berechnungsmöglichkeiten umzusehen. Wenn es sich um nulldimensionale Ideale handelt, verspricht der FGLM-Algorithmus, eine günstige Alternative zu sein.

„Aaaaahhh, what a fine day – for science!“

Dexter

Der nulldimensionale Fall

In diesem Kapitel soll der FGLM-Algorithmus für nulldimensionale Ideale beschrieben werden. Zuerst soll dabei die Idee für diesen Algorithmus motiviert werden. Dann wird der noch ausstehende Beweis der Äquivalenz der Bedingungen (i) und (vi) von Satz 15 aus Kapitel 3 nachgeholt. Nachdem dann ein Algorithmus vorgestellt ist, mit dessen Hilfe man ein (spezielles) definierendes System eines nulldimensionalen Ideals berechnen kann, folgt schließlich der FGLM-Algorithmus.

Sei also $I \subseteq P_n$ ein nulldimensionales Ideal, sei $<$ eine Wohlordnung und sei (N, E, S) die Zerlegung von I bezüglich $<$. Man erinnere sich: Das Ideal war gerade dann nulldimensional, wenn $|N|$ endlich war. Dann erinnere man sich weiter an Lemma 11 aus Kapitel 3: Kennt man die Ecken E eines Ideals und hat man ein $F \subset I$ mit $\mathrm{L}(F) = E$, so ist F eine Gröbnerbasis von I. Ziel wird also sein, zunächst die Zerlegung von I zu berechnen und dann (bzw. währenddessen) für jedes $m \in E$ ein $f \in I$ zu finden, so dass $\mathrm{L}(f) = m$. Wie findet man aber die Zerlegung, ohne eine Gröbnerbasis für das Ideal zur Verfügung zu haben?

Die nachfolgende Motivation soll durch das Beispiel aus Kapitel 3 (Seite 28) begleitet werden. Gesucht ist also die Zerlegung von $I = \langle xy^3 - x^2, x^3y^2 - y \rangle$ bezüglich der grad-lexikografischen Ordnung $<_{deglex}$. Die reduzierte Gröbnerbasis war gegeben durch

$$F_{deglex} = \left\{x^3y^2 - y,\ x^4 - y^2,\ xy^3 - x^2,\ y^4 - xy\right\} .$$

Bild 6 verdeutlicht die Situation.

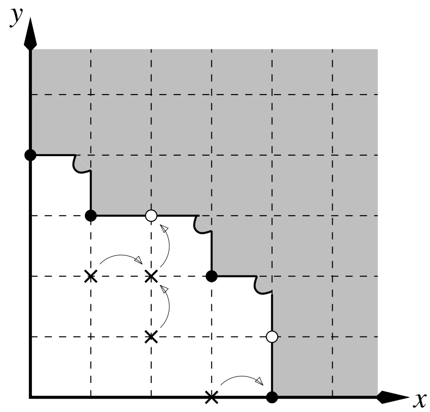


Bild 6: Die Zerlegung von I bezüglich $<_{deglex}$.

Schaut man sich die Definition der Zerlegung genauer an, so erkennt man, dass

$$N \cup E \cup S = N \cup R = \{m \cdot x_i \mid m \in N,\ i = 1, \ldots, n\} .$$

Man gelangt also zu allen Elementen aus $N \cup R$, indem man alle Monome aus N mit sämtlichen Ringvariablen multipliziert. Dabei erzeugt man die meisten Monome mehrmals, im Beispiel ist etwa $x^2y^2 = xy^2 \cdot x = x^2y \cdot y$. Ein – wie sich sogleich zeigen wird – nützlicher Trick besteht nun darin, sich zu merken, wie oft man ein Monom erzeugt hat. Dazu folgende Notation.

Notation. *Ist $I \subseteq K[x_1,\ldots,x_n]$ ein Ideal und (N,E,S) die Zerlegung von I bezüglich einer Wohlordnung, so setze für $m \in M_n$:*
a) $\operatorname{vars}(m) := \big|\, \{m' \in M_n \mid m = m' \cdot x_i \quad \textit{für ein } x_i\} \,\big|$.
b) $\mathrm{Z}(m) := \big|\, \{m' \in N \mid m = m' \cdot x_i \quad \textit{für ein } x_i\} \,\big|$.

Es zählt $\operatorname{vars}(m)$ die Anzahl aller direkten Vorgänger von m, wohingegen $\mathrm{Z}(m)$ die Anzahl der direkten Vorgänger zählt, die in N liegen; es ist also $\operatorname{vars}(m) \geq \mathrm{Z}(m)$ für alle $m \in M_n$;

Ist $m \in N \cup R$, so überlegt man sich leicht, dass

$$\begin{aligned} m \in N \cup E \;&\Leftrightarrow\; \operatorname{vars}(m) = \mathrm{Z}(m) \ , \\ m \in S \;&\Leftrightarrow\; \operatorname{vars}(m) > \mathrm{Z}(m) \ . \end{aligned}$$

Im Beispiel also etwa

$$\begin{array}{lclcl} x^2y^2 = xy^2 \cdot x = x^2y \cdot y & & \mathrm{Z}(x^2y^2) = 2 = \operatorname{vars}(x^2y^2) & & x^2y^2 \in N \cup E \\ x^4 = x^3 \cdot x & \Rightarrow & \mathrm{Z}(x^4) = 1 = \operatorname{vars}(x^4) & \Rightarrow & x^4 \in N \cup E \\ x^2y^3 = x^2y^2 \cdot y & & \mathrm{Z}(x^2y^3) = 1 \neq \operatorname{vars}(x^2y^3) & & x^2y^3 \in S \end{array}$$

Beachte: $x^2y^3 = xy^3 \cdot x$ zählt nicht, da $xy^3 \notin N$ (siehe auch Bild 6).

Die Berechnung der Zerlegung erfolgt nun schrittweise nach folgendem Schema: Angenommen, man hat schon drei Mengen $\overline{N} \subseteq N$, $\overline{E} \subseteq E$, $\overline{S} \subseteq S$ gefunden, so dass gilt

$$\overline{N} \cup \overline{E} \cup \overline{S} = \{m_1,\ldots,m_k\} = \{m \in N \cup E \cup S \mid m \leq m_k\} \ .$$

Man hat die Zerlegung bis zu m_k also schon lückenlos erstellt. Nun muss man den nächsten Kandidaten m_{k+1} finden, so dass $m_{k+1} \in N \cup R$.

Beachte: Im Allgemeinen ist $m_{k+1} \neq \min_<\{m \in M_n \mid m > m_k\}$. Vielmehr gilt:

$$\begin{aligned} m_{k+1} &= \min_<\{m \cdot x_i \mid m \in \overline{N} \ , \ m \cdot x_i > m_k\} \\ &= \min_<\{m \cdot x_i \mid m \in \overline{N} \ , \ m \cdot x_i \notin \overline{N} \cup \overline{E} \cup \overline{S}\} \ . \end{aligned}$$

Letzteres Minimum ist aber leicht zu berechnen, da man $\overline{N} \cup \overline{E} \cup \overline{S}$ kennt. Den nächsten Kandidaten kann man also schon einmal einfach finden. In dieser Situation gilt dann aber doch offensichtlich

$$\mathrm{Z}(m_{k+1}) = |\,\{m \in N \mid m_{k+1} = m \cdot x_i\}\,| = |\,\{m \in \overline{N} \mid m_{k+1} = m \cdot x_i\}\,| \ .$$

Da man $\overline{N}$ ja kennt, kann man also auch $\mathrm{Z}(m_{k+1})$ berechnen. Wie gesehen gilt dann:

$$\begin{aligned} \operatorname{vars}(m_{k+1}) = \mathrm{Z}(m_{k+1}) \;&\Rightarrow\; m_{k+1} \in N \cup E \ , \\ \operatorname{vars}(m_{k+1}) > \mathrm{Z}(m_{k+1}) \;&\Rightarrow\; m_{k+1} \in S \ . \end{aligned}$$

Man kann somit ohne Probleme zwischen $N \cup E$ und S trennen. Nun bleibt nur noch zu entscheiden, ob $m_{k+1} \in N$ oder $m_{k+1} \in E$. Das ist der Knackpunkt. Folgende Sprechweise wird dabei verwendet.

Notation. *Sei $I \subseteq P_n$ ein Ideal und sei $\Phi \subseteq M_n$. Dann sagt man, die Menge Φ sei* ***linear unabhängig modulo*** *I, wenn gilt: Sind $m_1, \ldots, m_s \in \Phi$ und sind $\lambda_1, \ldots, \lambda_s \in K$, so dass $\sum \lambda_i m_i \in I$, so folgt $\lambda_1 = \cdots = \lambda_s = 0$.*

Nun betrachte die Menge $\Phi = \overline{N} \cup \{m_{k+1}\}$ mit $m_{k+1} \in N \cup E$. Die Lösung des Problems besteht in der folgenden Feststellung: Die Menge Φ ist genau dann linear unabhängig modulo I, wenn $m_{k+1} \in N$.

Dies sieht man folgendermaßen: Sei Φ linear unabhängig modulo I und sei $m_{k+1} \in E$. Dann gibt es ein $f \in I$ mit $\mathrm{L}(f) = m_{k+1}$ und $\mathrm{tail}(f) \in \mathrm{span}_K(N)$, also insbesondere $\mathrm{tail}(f) \in \mathrm{span}_k(\overline{N})$, also $f \in \mathrm{span}_K(\Phi)$. Dies ist ein Widerspruch zu der Annahme, dass Φ linear unabhängig modulo I ist, also ist $m_{k+1} \in N$. Sei nun umgekehrt $m_{k+1} \in N$ und sei $\sum \lambda_i m_i \in I$. Sind nicht alle λ_i gleich Null, so ist $\mathrm{L}(\sum \lambda_i m_i) = m_l \in \mathrm{L}(I)$ für ein l. Das ist aber ein Widerspruch zu $m_i \in N$, also ist Φ linear unabhängig modulo I.

Die Aufgabe besteht somit darin, Φ auf lineare Unabhängigkeit modulo I zu testen. Das ist die Stelle, an der die definierenden Systeme ins Spiel kommen: Ist nämlich $\mathcal{L} : P_n \to K^d$ ein endliches definierendes System für I, dann gilt doch

$$\mathcal{L}(\sum \lambda_i m_i) = \sum \lambda_i \mathcal{L}(m_i) = 0 \;\Leftrightarrow\; \sum \lambda_i m_i \in I \, .$$

Aber $\mathcal{L}(m_i) \in K^d$, also hat man die Aufgabe, lineare Unabhängigkeit modulo I zu überprüfen, abgewälzt auf die Aufgabe, lineare Abhängigkeit von Vektoren zu überprüfen. Das ist aber ein wohlbekanntes, leicht zu lösendes Problem. Man gewinnt sogar noch ein wenig mehr: Hat man nämlich $\lambda_1, \ldots, \lambda_{k+1}$ gefunden, so dass $\mathcal{L}(\sum \lambda_i m_i) = 0$, so setzt man $f = \sum \lambda_i m_i$ und hat damit direkt ein Element aus der reduzierten Gröbnerbasis von I erhalten! (Beachte: Es ist $\mathrm{tail}(f) \in \mathrm{span}_K(\overline{N})$ und $\mathrm{L}(f) = m_{k+1} \in E$.)

Also konnte in diesem Schritt m_{k+1} zunächst gefunden werden und dann genau einer der Mengen N, E oder S zugeordnet werden. Man wird entsprechend eine der Mengen $\overline{N}$, $\overline{E}$ oder $\overline{S}$ vergrößern und wieder von vorne beginnen. Dies wiederholt man so lange, bis gilt:

$$\{m \cdot x_i \mid m \in \overline{N} \, , \; m \cdot x_i \notin \overline{N} \cup \overline{R}\} = \{\} \, ,$$

d.h. bis keine weiteren Kandidaten mehr zur Verfügung stehen. Während der Berechnung hat man dann für jedes $m \in E = \overline{E}$ ein $f \in I$ erhalten, so dass $\mathrm{L}(f) \in E$ und $\mathrm{tail}(f) \in \mathrm{span}_K(N)$. Voilà: Man hat die reduzierte Gröbnerbasis von I berechnet.

Dies sollte vorerst nur die grobe Idee skizzieren. Der genaue Beweis, warum der Algorithmus terminiert und alle Mengen richtig berechnet, wird später erbracht. Man beachte noch einmal Folgendes: Bei der eben beschriebenen Motivation lag der Knackpunkt gerade darin, für ein $m \in N \cup E$ zu entscheiden, ob $m \in N$ oder $m \in E$. Wenn diese Entscheidung getroffen werden konnte, wurde die Zerlegung gefunden. (Denn die Frage, ob $m \in S$ ist, konnte dann durch ein reines Zählargument beantwortet werden.) Oben wurde diese Entscheidung mit Hilfe eines endlichen definierenden Systems getroffen, da man eine Gröbnerbasis ja gerade nicht zur Hand hat.

Berechnung der Zerlegung

Nachfolgend soll nun zunächst erläutert werden, wie man ein definierendes System berechnen kann. Dabei steht einem folgende Aufgabe vor Augen: Man hat eine reduzierte Gröbnerbasis eines nulldimensionalen Ideals bezüglich einer Ordnung $<_1$ zur Verfügung. Davon ausgehend will man ein definierendes System für dieses Ideal berechnen und sich mit Hilfe dieses Systems nach obiger Motivation eine reduzierte Gröbnerbasis bezüglich einer neuen Ordnung $<_2$ verschaffen. Überlegt man sich nun, wie man aus einer Gröbnerbasis ein definierendes System berechnen kann, so gelangt man wieder zu dem Problem, dass man die Zerlegung bezüglich der Ordnung $<_1$ kennen muss. Um dann zwei im äußeren Aussehen sehr ähnliche Algorithmen nicht zweimal auf ihre Korrektheit überprüfen zu müssen, wird daher zuerst eine „Algorithmus-Schablone" formuliert (Satz 22) und verifiziert, in die die beiden Algorithmen dann perfekt hineinpassen. Diese Schablone ist mehr ein mathematischer Satz denn ein Algorithmus.

Mit der Motivation von oben ist aber auch schon alles getan, um besagte Schablone zu formulieren. Der nachfolgende Satz gibt also eine Vorschrift, wie man die Zerlegung eines nulldimensionalen Ideals berechnen kann. Die dabei verwendeten Variablen werden auf Seite 42 erklärt.

Satz 22. *Sei $I \subseteq P_n$ ein nulldimensionales Ideal. Sei $<$ eine Wohlordnung. Sei (N, E, S) die Zerlegung von I, sei $N \cup E \cup S = \{m_1, \ldots, m_k\}$ mit $m_i < m_j$ für $i < j$. Sei ein Mechanismus gegeben, der folgende Entscheidung treffen kann:*

„Gegeben ein $m \in N \cup E$ und alle $m_i \in N$ mit $m_i < m$. Ist $m \in N$ oder ist $m \in E$?"

Dann berechnet die folgende Vorschrift nach genau k Schritten durch die Schleife die Mengen $\overline{N}$, $\overline{S}$, $\overline{E}$, so dass am Ende schließlich gilt: $\overline{N} = N$, $\overline{E} = E$, $\overline{S} = S$.

$C = \{1\} \subset M_n$
- (0) $C \subseteq N \cup E \cup S$ und $C \cap (\overline{N} \cup \overline{E} \cup \overline{S}) = \{\}$
- (1) Es gilt zu jeder Zeit: $\overline{N} \subseteq N$, $\overline{E} \subseteq E$, $\overline{S} \subseteq S$.

while $(C \neq \{\})$
    $m \leftarrow \min_< C$
- (2) $m = m_s$, wenn dies der s-te Durchlauf ist
- (3) $Z_m = |\{m' \in N \mid m' \cdot x_i = m\}| = Z(m)$

    $C \leftarrow C \setminus \{m\}$
    if $(\mathbf{vars}(m) = Z_m)$
- (4) $m \in N \cup E$ und $\overline{N} = \{m_i \in N \mid m_i < m\}$

        if $(m \in E)$
- (4.1) $m \in E$

            $\overline{E} \leftarrow \overline{E} \cup \{m\}$
        else
- (4.2) $m \in N$

            $\overline{N} \leftarrow \overline{N} \cup \{m\}$
            for $(i = 1, \ldots, n)$
                $C \leftarrow C \cup \{x_i \cdot m\}$
                $Z_{x_i \cdot m} \leftarrow Z_{x_i \cdot m} + 1$
    else
- (4.3) $m \in S$

        $\overline{S} \leftarrow \overline{S} \cup \{m\}$
- (5) $C \subseteq N \cup E \cup S$ und $C \cap (\overline{N} \cup \overline{E} \cup \overline{S}) = \{\}$
- (6) $Z_m = |\{m' \in \overline{N} \mid m' \cdot x_i = m\}|$ für alle $m \in C$
- (7) $C = \{\} \Leftrightarrow \overline{N} \cup \overline{S} \cup \overline{E} = \{m_1, \ldots, m_k\}$

end•

Beweis.
Der Beweis erfolgt induktiv, indem die in den Kommentaren angegebenen Schleifeninvari-

anten nachgeprüft werden. Nachfolgend bezeichne ein Index an einer Invarianten, etwa $(5)_s$, die Aussage der Invarianten im s-ten Schleifendurchlauf. Mit einem $'$ werden die im aktuellen Schritt veränderten Variablen bezeichnet.

Vorbemerkung: Gilt $(0)_i$ und $(3)_i$, so gelten automatisch auch $(4.1)_i$, $(4.2)_i$ und $(4.3)_i$, wie in der Motivation gesehen. Vorausgesetzt natürlich, dass der in der Voraussetzung geforderte Mechanismus richtig entscheidet.

Induktionsanfang: $(0)_1$, $(1)_1$, $(2)_1$ und $(3)_1$ gelten trivialerweise. Ist nun $1 \in E$, so folgt $\overline{E}' = \{1\} \subseteq E$ und $C' = \{\}$. Ist $1 \in N$, so folgt $\overline{N}' = \{1\} \subseteq N$ und $C' = \{x_1, \ldots, x_n\}$ sowie $\mathrm{Z}'_{x_i} = 1$ für alle i. Also folgt in beiden Fällen $(1)_1$, $(5)_1$, $(6)_1$ und $(7)_1$. Damit ist der Induktionsanfang gezeigt.

Induktionsschritt: Es gelte $(0)_i, \ldots, (7)_i$ für alle $i \leq s$.

Für $(2)_{s+1}$ ist zu zeigen: $m = \min_< C = m_{s+1}$. Nach $(2)_1, \ldots, (2)_s$ ist $\overline{N} \cup \overline{E} \cup \overline{S} = \{m_1, \ldots, m_s\}$. Da nach $(5)_s$ gilt $C \subseteq N \cup E \cup S$ und $C \cap \overline{N} \cap \overline{E} \cap \overline{S} = \{\}$, genügt es zu zeigen, dass $m_{s+1} \in C$. Es ist $m_{s+1} \neq 1$. Außerdem ist $m_{s+1} \in N \cup E \cup S$. Also existiert ein x_i mit $m_{s+1}/x_i \in N$, also etwa $m_{s+1}/x_i = m_l$ für ein $l < s+1$. Nach $(1)_i$ folgt $m_l \in \overline{N}$. Also wurde im l-ten Schritt $m_l x_i = m_{s+1}$ in C eingefügt. Nach $(2)_1, \ldots, (2)_s$ konnte m_{s+1} aber nicht aus C entfernt werden, also ist $m_{s+1} \in C$. Somit ist $m_{s+1} = \min_< C$. Also gilt $(2)_{s+1}$.

Zu $(3)_{s+1}$: Nach $(6)_s$ gilt:

$$\mathrm{Z}_{m_{s+1}} = \left| \{m' \in \overline{N} \mid m' \cdot x_i = m_{s+1}\} \right| .$$

(Beachte: Zum Zeitpunkt $(6)_s$ ist $m_{s+1} \in C$.)

Nach $(2)_1, \ldots, (2)_s$ ist aber $\overline{N} \cup \overline{E} \cup \overline{S} = \{m_1, \ldots, m_s\}$. Also gilt sogar

$$\mathrm{Z}_{m_{s+1}} = \left| \{m' \in \overline{N} \mid m' \cdot x_i = m_{s+1}\} \right| ,$$

also gilt $(3)_{s+1}$.

Zu $(5)_{s+1}$: Sei $m_{s+1} \in E \cup S$. Dann ist $C' = C \setminus \{m_{s+1}\}$, also $C' \subseteq N \cup E \cup S$ und weiter

$$\begin{aligned} C' \cap (\overline{N}' \cup \overline{E}' \cup \overline{S}') &= (C \setminus \{m_{s+1}\}) \cap (\overline{N}' \cup \overline{E}' \cup \overline{S}') \\ &= C \cap ((\overline{N}' \cup \overline{E}' \cup \overline{S}') \setminus \{m_{s+1}\}) = C \cap (\overline{N} \cup \overline{E} \cup \overline{S}) = \{\} . \end{aligned}$$

Sei $m_{s+1} \in N$. Dann ist $C' = C \setminus \{m_{s+1}\} \cup \{m_{s+1}x_i \mid i = 1, \ldots, n\}$, also $C' \subseteq N \cup E \cup S$ und weiter

$$\begin{aligned} C' \cap (\overline{N}' \cup \overline{E}' \cup \overline{S}') &= (C' \setminus \{m_{s+1}\} \cup \{m_{s+1}x_i \mid \forall i\}) \cap (\overline{N} \cup \{m_{s+1}\} \cup \overline{E} \cup \overline{S}) \\ &= (C \cap (\overline{N} \cup \overline{E} \cup \overline{S})) \cup (\{m_{s+1}x_i \mid \forall i\} \cap (\overline{N} \cup \overline{E} \cup \overline{S})) = \{\} . \end{aligned}$$

Nach $(2)_1, \ldots, (2)_s$ gilt aber $\overline{N} \cup \overline{E} \cup \overline{S} = \{m_1, \ldots, m_s\}$, also gilt $(5)_{s+1}$ in beiden Fällen.

Zu $(6)_{s+1}$: Ist $m_{s+1} \in E \cup S$, so wurde $\mathrm{Z}_{m_{s+1}}$ nicht verändert. Dann gilt $(6)_{s+1}$, da $(6)_s$ gilt. Sei also $m_{s+1} \in N$. Dann ist $\mathrm{Z}'_{m_{s+1}x_i} = \mathrm{Z}_{m_{s+1}x_i} + 1$. Es ist

$$\begin{aligned} &\left| \{m' \in \overline{N}' \mid m'x_i = m\} \right| \\ &\qquad = \left| \{m' \in \overline{N} \mid m'x_i = m\} \right| + \left| \{l \in \{1, \ldots, n\} \mid m_{s+1}x_l = m\} \right| \\ &\qquad = \mathrm{Z}_m + \begin{cases} 1, & \exists\, l \text{ mit } m = m_{s+1}x_l; \\ 0, & \text{sonst.} \end{cases} \end{aligned}$$

Also gilt auch $(6)_{s+1}$.

Zu $(7)_{s+1}$: „$\Leftarrow$“ ist klar wegen $(5)_{s+1}$. „$\Rightarrow$“ Sei $C' = \{\}$ und $s+1 < k$. Dann ist $m_{s+1} \in E \cup S$, da sonst $C' \neq \{\}$. Also $\overline{N}' = \overline{N}$ und $\overline{E}' \cup \overline{S}' = \overline{E} \cup \overline{S} \cup \{m_{s+1}\}$.

Ferner ist $m_{s+2} \notin \overline{N}' \cup \overline{E}' \cup \overline{S}' = \{m_1, \ldots, m_{s+1}\}$. Aber $m_{s+2}/x_i = m_l \in N$ für ein i und ein l. Dann ist $l < s+2$. Also $m_l \in \overline{N}$, d.h. $m_l x_i = m_{s+2} \in C$ am Ende des l-ten

Schrittes nach (2). Ebenfalls mit (2) folgt dann, dass $m_{s+2} \in C$ auch im s-ten Schritt, da nur Elemente m_i mit $i < s+2$ aus C entfernt wurden nach (2). Dies ist jedoch ein Widerspruch, also gilt auch $(7)_{s+1}$.
Da $(4.1)_{s+1}$, $(4.2)_{s+1}$ und $(4.3)_{s+1}$ wie oben bemerkt dann automatisch gelten, ist der Induktionsschritt damit gezeigt.
Somit gilt (2) zu jedem Zeitpunkt, d.h. die Monome $m_1, \dots, m_k$ werden genau in dieser Reihenfolge bearbeitet. Da außerdem in jedem Schritt genau eine der Mengen $\overline{N}$, $\overline{E}$ bzw. $\overline{S}$ um genau ein Element vergrößert wird, folgt mit (7), dass der Algorithmus nach genau k Schritten durch die Schleife mit dem richtigen Ergebnis terminiert. Damit ist der Satz vollständig bewiesen. □

Bemerkungen zum Algorithmus
Folgende Variablen werden im Algorithmus verwendet:

• $C \subset M_n$ ist die Menge der „Kandidaten“ $m \in N \cup R$, die noch nicht bearbeitet wurden. Wird ein Element aus N gefunden, so wird diese Menge um neue Kandidaten erweitert, vegleiche Zeile 10 bis 12 im Algorithmus.

• $\overline{N}, \overline{E}, \overline{S} \subset M_n$ werden Schritt für Schritt mit Monomen aus N, E und S aufgefüllt, bis sie am Ende mit N, E und S übereinstimmen. Sie werden bei Beginn als leere Mengen initialisiert.

• Z_m für $m \in N \cup R$ ist der zuvor beschriebene Zähler $Z(m)$, der aufzeichnet, wie oft ein Monom in die Liste der Kandidaten eingefügt worden ist. Startwert: $Z_m = 0$ für alle $m \in N \cup R$.

Der angegebene Weg, die Zerlegung eines nulldimensionalen Ideals zu finden, ist nicht der einzig mögliche. Denkbar ist zum Beispiel auch ein rekursiver Ansatz, bei dem man startend bei $1 \in M_n$ zunächst in eine Variablenrichtung bis zum Rand vorstößt, dann für alle so gefundenen Monome in die nächste Variablenrichtung geht, usw. Will man wirklich nur die Zerlegung berechnen, so ist dies ein effektiver Ansatz. Für das vorliegende Problem zeigt Satz 22 aber einen entscheidenden Vorteil: Hat man ein Monom m der Zerlegung gefunden, so stellt der Algorithmus sicher, dass man zu diesem Zeitpunkt schon alle m' mit $m = m' x_i$, also alle direkten Vorgänger von m, bearbeitet hat. Dies ist sowohl für den FGLM-Algorithmus als auch für das Berechnen des definierenden Systems äußerst hilfreich.

Am Ende des Algorithmus ist $|N| = |\overline{N}| = \mathrm{vdim}(I)$. Führt man also noch zusätzlich einen Zähler ein, der mitzählt, wie oft ein Element zu $\overline{N}$ hinzugefügt worden ist, so hat man automatisch auch noch $\mathrm{vdim}(I)$ berechnet.

Berechnung des definierenden Systems

Zunächst wie versprochen der noch ausstehende Teil des Beweises von Satz 15 aus Kapitel 3, hier noch einmal zur Erinnerung als eigener Satz formuliert.

Satz 23. *Sei $I \subseteq P_n$ ein Ideal. Dann ist I genau dann nulldimensional, wenn es ein endliches definierendes System $\mathcal{L}$ für I gibt.*

Beweis.
„$\Rightarrow$“ Sei I nulldimensionalund sei $<$ eine Wohlordnung, sei (N, E, S) die Zerlegung von I bezüglich $<$. Dann ist $d = |N| < \infty$, etwa $N = \{b_1, \dots, b_d\}$.
Sei $\sigma : P_n/I \to K^d$ vermöge $\sigma(b_i + I) = e_i$, σ K-linear. Sei $\pi : P_n \to P_n/I$ die kanonische Projektion. Dann ist

$$\mathcal{L} : P_n \to K^d \quad \text{mit} \quad \mathcal{L} = \sigma \circ \pi \circ \mathrm{NF}$$

ein endliches definierendes System:

$$\begin{aligned} \mathcal{L}(f)=0 \;&\Leftrightarrow\; \sigma\circ\pi\circ \mathrm{NF}(f)=0 \;\Leftrightarrow\; \sigma(\mathrm{NF}(f)+I)=0 \\ &\Leftrightarrow\; \mathrm{NF}(f)+I=0+I \;\Leftrightarrow\; \mathrm{NF}(f)\in I \;\Leftrightarrow\; \mathrm{NF}(f)=0 \;\Leftrightarrow\; f\in I\,. \end{aligned}$$

„$\Leftarrow$“ Sei $\mathcal{L}: P_n \to K^s$ ein endliches definierendes System von I. Zeige: $|\,N\,| \leq s$. Da $N+I$ eine Basis von P_n/I ist, folgt dann, dass I nulldimensional ist.
Annahme: Seien $m_1,\ldots,m_{s+1} \in N$ mit $m_i \neq m_j$ für $i \neq j$. Dann ist $\mathcal{L}(m_i) \neq \mathcal{L}(m_j)$; sonst wäre $\mathcal{L}(m_i - m_j) = 0$, d.h. $m_i - m_j \in I$. Widerspruch zu $m_i, m_j \in N$.
Also sind $\mathcal{L}(m_1),\ldots,\mathcal{L}(m_{s+1})$ genau $s+1$ verschiedene Vektoren in K^s. Somit existieren $\lambda_1,\ldots,\lambda_{s+1} \in K$, nicht alle $\lambda_i = 0$, mit $\sum \lambda_i \mathcal{L}(m_i) = 0$. $\Rightarrow \mathcal{L}(\sum\lambda_i m_i) = 0 \Rightarrow \sum \lambda_i m_i \in I$, also gibt es ein k mit $\mathrm{L}(\sum\lambda_i m_i) = m_k$ und also $m_k \in \mathrm{L}(I)$. Widerspruch zu $m_k \in N$. Also ist $|\,N\,| \leq s$. □

Der Beweis hat gezeigt: Ist I ein nulldimensionales Ideal und ist $\mathcal{L}: P_n \to K^s$ ein endliches definierendes System, so ist $\mathrm{vdim}(I) \leq s$. Das macht folgende Definition sinnvoll.

Definition. *Sei I ein nulldimensionales Ideal. Ein definierendes System $\mathcal{L}: P_n \to K^s$ von I heißt **minimal**, wenn* $\mathrm{vdim}(I) = s$.

In den Situationen, in denen definierende Systeme in dieser Arbeit Verwendung finden, müssen sie folgende Aufgabe meistern: Man möchte $\mathcal{L}(m)$ berechnen für ein $m \in M_n$, kennt aber aus vorherigen Schritten schon $\mathcal{L}(m')$ für einen direkten Vorgänger m' von m. Bei der Berechnung von $\mathcal{L}(m)$ könnte man also Informationen aus $\mathcal{L}(m')$ mit einfließen lassen. Unter diesem Aspekt ist es besonders geschickt, endliche definierende Systeme durch spezielle Matrizen zu repräsentieren; dies wird nachfolgend motiviert.

Sei also $\mathcal{L}: P_n \to K^d$ ein definierendes System eines Ideals I. Sei $<$ eine Wohlordnung und sei (N,E,S) die Zerlegung von I bezüglich $<$. Sei etwa $N = \{b_1,\ldots,b_d\}$. Sei o.B.d.A. $b_1 = 1$. Dann ist natürlich $\mathcal{L}(b_i) \neq \mathcal{L}(b_j)$ für $i \neq j$, und die Menge $\{\mathcal{L}(b_1),\ldots,\mathcal{L}(b_d)\}$ ist linear unabhängig und fortan als Basis für K^d gewählt, d.h. es ist $e_i = \mathcal{L}(b_i)$. Bezüglich dieser Basis seien die Matrizen $\mathcal{M}_1,\ldots,\mathcal{M}_n \in \mathrm{Mat}(d\times d, K)$ definiert durch

$$\mathcal{M}_k\, e_i = \mathcal{M}_k \mathcal{L}(b_i) := \mathcal{L}(x_k\, b_i) \quad \text{für } k=1,\ldots,n \text{ und } i=1,\ldots,d\,.$$

Beachte: Es ist $x_k\, b_i \in N \cup R$. Da die Multiplikation $\mathcal{M}_k\, e_i$ der Auswertung von b_i multipliziert mit der Ringvariable x_k durch $\mathcal{L}$ entspricht, werden die Matrizen nachfolgend oft auch als *Multiplikationsmatrizen* bezeichnet. Sind die Matrizen so definiert, dann gilt auch

$$\mathcal{L}(x_k m) = \mathcal{M}_k \mathcal{L}(m) \quad \text{für } m \in M_n\,.$$

Um das zu sehen, sei $p \in P_n$ und $\mathcal{L}(p) = \sum\lambda_i e_i$, also $\mathcal{L}(p) = \mathcal{L}(\sum\lambda_i b_i)$. Dann ist aber $p - \sum\lambda_i b_i \in I$. Ist nun $q \in P_n$, so ist auch $qp - q\sum\lambda_i b_i \in I$, also auch $\mathcal{L}(qp) = \mathcal{L}(q\sum\lambda_i b_i)$. Ist speziell $m \in M_n$ und $\mathcal{L}(m) = \mathcal{L}(\sum\lambda_i b_i) = \sum\lambda_i e_i$, so ist

$$\begin{aligned} \mathcal{L}(x_k m) &= \mathcal{L}\big(x_k(\textstyle\sum\lambda_i b_i)\big) = \sum\lambda_i \mathcal{L}(x_k b_i) = \sum\lambda_i \mathcal{M}_k e_i = \mathcal{M}_k\big(\sum\lambda_i e_i\big) \\ &= \mathcal{M}_k \mathcal{L}\big(\textstyle\sum\lambda_i b_i\big) = \mathcal{M}_k \mathcal{L}(m) \quad \text{für } k=1,\ldots,n \text{ und } m \in M_n\,. \end{aligned}$$

Hier sieht man schon, dass die Multiplikationsmatrizen den oben erwähnten Anforderungen gerecht werden. Speziell folgt

$$\begin{aligned} \mathcal{M}_k\mathcal{M}_l e_i &= \mathcal{M}_k\mathcal{L}(x_l b_i) = \mathcal{L}(x_k x_l b_i) = \mathcal{L}(x_l x_k b_i) = \mathcal{M}_l\mathcal{L}(x_k b_i) \\ &= \mathcal{M}_l\mathcal{M}_k\mathcal{L}(b_i) = \mathcal{M}_l\mathcal{M}_k e_i \quad \text{für } k,l=1,\ldots,n \text{ und } i=1,\ldots,d\,. \end{aligned}$$

Ist also $m = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$, so folgt mit $\mathcal{L}(1) = (1, 0, \dots, 0)^t$:

$$\mathcal{L}(m) = \mathcal{L}(x_1^{\alpha_1} \cdots x_n^{\alpha_n}) = \mathcal{M}_1^{\alpha_1} \cdots \mathcal{M}_n^{\alpha_n} \cdot \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} .$$

Auswertung von $\mathcal{L}$ an der Stelle $m \in M_n$ entspricht somit der Multiplikation des ersten Einheitsvektors mit den entsprechenden Multiplikationsmatrizen. Ein definierendes System kann also vollständig über seine Multiplikationsmatrizen dargestellt werden. Für den FGLM-Algorithmus ist die Darstellung definierender Systeme über Matrizen ausgesprochen vorteilhaft. Will man $\mathcal{L}(m)$ berechnen, so sucht man sich ein m' mit $m = m'x_k$. Für dieses m' hat man aber $\mathcal{L}(m')$ schon berechnet, und somit erhält man $\mathcal{L}(m)$ durch einfache Multiplikation von $\mathcal{L}(m')$ mit $\mathcal{M}_k$. (Beachte hierzu auch besonders die Bemerkung zu Satz 22.)

Die Definition der Multiplikationsmatrizen weist den Weg, wie diese zu berechnen sind. Man muss die Zerlegung (N, E, S) des Ideals finden, für jedes $m \in N \cup R$ teilende Variablen $x_{\sigma(1)}, \dots, x_{\sigma(s)}$ suchen, schließlich $\mathcal{L}(m) \in K^d$ auswerten und in die entsprechenden Spalten der Matrizen $\mathcal{M}_{\sigma(1)}, \dots, \mathcal{M}_{\sigma(s)}$ eintragen. Diese Aufgabe passt perfekt in die Schablone von Satz 22. Die Matrizen werden im Algorithmus dabei Schritt für Schritt spaltenweise aufgebaut. Der in Satz 22 geforderte Entscheidungsmechanismus fällt sehr einfach aus, wenn man eine reduzierte Gröbnerbasis $F \subset I$ von I zur Verfügung hat. Ist nämlich $m \in N \cup E$, so ist $m \in E$ genau dann, wenn es ein $f \in F$ gibt mit $\mathrm{L}(f) = m$.

Von nun an soll aber nicht mehr der ganz allgemeine Fall betrachtet werden. Vielmehr hat man die schon mehrmals erwähnte Aufgabe vor Augen: Gegeben eine reduzierte Gröbnerbasis eines nulldimensionalen Ideals bezüglich einer Wohlordnung $<_1$. Finde mit ihrer Hilfe ein definierendes System für dieses Ideal. Mit Hilfe dieses definierenden Systems erstelle nun wiederum bezüglich einer neuen Wohlordnung $<_2$ eine reduzierte Gröbnerbasis.

Welches definierende System bietet sich für diese Problemstellung an? Der Beweis von Satz 23 gibt die Antwort: Wähle das dort erwähnte definierende System

$$\mathcal{L}_{\mathrm{NF}} = \sigma \circ \pi \circ \mathrm{NF}_{<_1} : P_n \to K^d .$$

Dabei ist $\pi : P_n \to P_n/I$ die kanonische Projektion, und $\sigma : P_n/I \to K^d$ ist definiert als $\sigma(b_i + I) = e_i$ mit $N = \{b_1, \dots, b_d\}$ (σ K-linear). Wie sich gleich zeigen wird, gestaltet sich die Berechnung der Multiplikationsmatrizen für dieses spezielle System sehr einfach und effizient, wenn man eine reduzierte Basis zur Verfügung hat. Der Algorithmus **multmat** zur Berechnung der Matrizen ist also nur für dieses definierende System $\mathcal{L}_{\mathrm{NF}}$ angegeben; es ist aber direkt ersichtlich, an welchen Stellen Änderungen im Algorithmus anzubringen sind, will man den allgemeinen Fall betrachten.

Wie angedeutet besteht die Aufgabe darin, $\mathcal{L}(m)$ auszurechnen für alle $m \in N \cup R$, hier also gerade darin, $\mathrm{NF}(m)$ für diese Monome zu finden. Das folgende Lemma hilft dabei. Es ist der Grund dafür, dass für das vorliegende Problem $\mathcal{L}_{\mathrm{NF}}$ so effizient zu berechnen ist.

Lemma 24. *Sei $I \subseteq P_n$ ein nulldimensionales Ideal und sei $<$ eine Wohlordnung. Ferner sei (N, E, S) die Zerlegung und $F \subset I$ die normierte, reduzierte Gröbnerbasis von I bezüglich dieser Ordnung. Dann gilt:*

a) Ist $m \in N$, so ist $\mathrm{NF}(m) = m$.

b) Ist $m \in E$, so existiert ein $f \in F$ mit $\mathrm{L}(f) = m$. Dann ist $\mathrm{NF}(m) = -\,\mathrm{tail}(f)$.

c) Ist $m \in S$, so existiert ein $m' \in R$ mit $m = m'x_k$. Sei $\mathrm{NF}(m') = \sum \lambda_i b_i$ mit $b_i \in N$ und $\lambda_i \in K$. Dann ist $\mathrm{NF}(m) = \sum \lambda_i \,\mathrm{NF}(x_k b_i)$.

Beweis.
zu a): Klar.
zu b): Dass f existiert, ist klar. Da F eine reduzierte Gröbnerbasis ist, ist $\operatorname{tail}(f) \in \operatorname{span}_K(N)$. Da F normiert ist, ist $\operatorname{LT}(f) = m$. Also $m - (-\operatorname{tail}(f)) = m + \operatorname{tail}(f) = f \in I$.
zu c): Die Existenz von m' ist wieder klar. Mit Satz 14 aus Kapitel 3 folgt dann:

$$\begin{aligned}\operatorname{NF}(m) &= \operatorname{NF}(m'x_k) = \operatorname{NF}\big(\operatorname{NF}(x_k)\operatorname{NF}(m')\big) = \operatorname{NF}\big(\operatorname{NF}(x_k)(\textstyle\sum \lambda_i b_i)\big) \\ &= \sum \lambda_i \operatorname{NF}\big(\operatorname{NF}(x_k) b_i\big) = \sum \lambda_i \operatorname{NF}(x_k b_i)\ .\end{aligned}$$

Also gilt auch c). □

Damit hat man es aber erreicht, die Multiplikationsmatrizen für $\mathcal{L} = \mathcal{L}_{\mathrm{NF}}$ zu konstruieren, ohne auch nur einmal die Normalform wirklich ausrechnen zu müssen. In einer reduzierten Gröbnerbasis steckt nach Lemma 24 so viel Information, dass man ohne eine Berechnung auskommt. Ist nämlich $m \in N$ oder $m \in E$, so kann man $\mathcal{L}(m)$ wie gesehen direkt angeben, also direkt als Spalten in die entsprechenden Matrizen einsetzen. Ist $m = x_k b_i \in S$, so ist zunächst

$$\begin{aligned}\mathcal{L}(m) &= \mathcal{L}\big(\textstyle\sum \lambda_i \operatorname{NF}(x_k b_i)\big) = \sum \lambda_i \mathcal{L}\big(\operatorname{NF}(x_k b_i)\big) = \sum \lambda_i \mathcal{L}(x_k b_i) \\ &= \sum \lambda_i \mathcal{M}_k e_i\ .\end{aligned}$$

Es wird sich zeigen: Ist man im Algorithmus in der Situation, $\mathcal{L}(m)$ für ein $m \in S$ bestimmen zu müssen, so hat man in den vorherigen Schritten bereits genügend Spalten der Multiplikationsmatrizen aufgebaut, um obige Summe berechnen zu können. Damit konnte auch in diesem Fall eine direkte Berechnung von $\operatorname{NF}(m)$ vermieden werden. Stattdessen müssen einfach gewisse Spalten der bis hierhin gefundenen Matrizen aufsummiert werden.

Im Vergleich zu Satz 22 muss der Algorithmus geringfügig erweitert werden. Will man $\mathcal{L}(m)$ in die Multiplikationsmatrizen eintragen, so benötigt man hierfür alle teilenden Variablen x_k von m, da die entsprechenden Matrizen $\mathcal{M}_k$ verändert werden müssen. Daher wird zu jedem Monom eine Indexmenge mitgeführt, die aufzeichnet, welche Variablen dieses Monom teilen und welche Monome die direkten Vorgänger sind. Näheres hierzu entnimmt man am besten dem Algorithmus selbst.

Der FGLM-Algorithmus benötigt lediglich die Multiplikationsmatrizen, an der Zerlegung, die während der Berechnung der Matrizen automatisch gefunden wurde, ist er nicht interessiert. Daher liefert der Algorithmus **multmat** nur die Vektorraumdimension und die Matrizen zurück.

Bemerkungen zum Algorithmus
Größtenteils werden jene Variablen verwendet, die auch schon in Satz 22 benutzt wurden (siehe Seite 40).

- $d \in \mathbb{N}$ enthält die Anzahl der Monome aus N, die bisher gefunden worden sind. Am Ende ist demnach $d = \operatorname{vdim}(I)$. Startwert: $d = 0$.
- $b_i \in M_n$ für $1 \leq i \leq \operatorname{vdim}(I)$ ist das i-te Monom aus N, das gefunden wurde. Es ist $b_1 < b_2 < \ldots$ und am Ende gilt $N = \{b_1, \ldots, b_d\}$.
- $D_m \subset \mathbb{N} \times \mathbb{N}$ für $m \in N \cup R$. Diese Tupelmenge enthält Paare $(i\,, j) \in \mathbb{N} \times \mathbb{N}$, so dass $m = x_i b_j$. Wenn D_m an **insertcols** (siehe unten) übergeben wird, enthält es alle Tupel mit dieser Eigenschaft. Startwert: $D_m = \{\}$ für alle $m \in N \cup R$.
- $v_m \in K^d$ für $m \in N \cup R$ enthält die Vektorraumrepräsentation von $\operatorname{NF}(m)$, d.h. es ist $\operatorname{NF}(m) = \sum v_m^{(i)} b_i$.

Implementierungstechnisch gibt es noch ein Problem. Man kennt zu Beginn des Algorithmus noch nicht die Vektorraumdimension des Ideals, also auch nicht die Dimension der zu erstellenden Multiplikationsmatrizen. Nun könnte man einfach zu Beginn diese Dimension berechnen, aber das ist sicher kein sehr effizienter Weg. Sinnvoller scheint es, die Matrizen so zu implementieren, dass ihre Dimension während der Erstellung wachsen kann. Der höhere Aufwand bei der Implementierung macht sich in den kürzeren Ausführungszeiten bezahlt.

Der Algorithmus ist so angelegt, dass die Matrizen spaltenweise konstruiert werden und zwar in der richtigen Reihenfolge: Wurde die Spalte i in eine Matrix eingesetzt, so kann man sicher sein, dass alle Spalten j dieser Matrix mit $j < i$ in vorherigen Schritten schon erstellt worden sind. Das Einsetzen einer Spalte in die Matrizen geschieht im Algorithmus über die Funktion

$$\textbf{insertcols}(D_m, v_m) \ .$$

Diese bearbeitet alle Tupel $(i\,,j) \in D_m$ und setzt den Vektor v_m als j-te Spalte in die Matrix $\mathcal{M}_i$ ein. Denn beachte: Ist $(i\,,j) \in D_m$, so bedeutet dies, dass $m = x_i b_j$. Da v_m die Vektorraumrepräsentation von $\mathrm{NF}(m)$ ist, gilt $\mathcal{L}_{\mathrm{NF}}(m) = v_m$ und dann

$$\mathcal{L}_{\mathrm{NF}}(m) = \mathcal{L}_{\mathrm{NF}}(x_i b_j) = v_m = \mathcal{M}_i e_j = \mathcal{M}_i^{(j)} \ ,$$

wobei $\mathcal{M}_i^{(j)}$ die j-te Spalte der Matrix $\mathcal{M}_i$ bezeichnet. Somit werden über **insertcols** die richtigen Spalten in die richtigen Matrizen eingesetzt.

Satz 25. *Sei $I \subset K[x_1,\ldots,x_n]$ ein nulldimensionales Ideal, sei $<$ eine Wohlordnung und sei $F \subset I$ die normierte, reduzierte Gröbnerbasis von I. Ferner sei (N,E,S) die Zerlegung von I und sei $|\,N \cup E \cup S\,| = k$. Dann berechnet der Algorithmus* **multmat** *(Algorithmus 4) in genau k Schritten durch die Schleife die Multiplikationsmatrizen des definierenden Systems $\mathcal{L}_{\mathrm{NF}}$ für I.*

Beweis.
Zusammen mit Satz 22 und Lemma 24 gestaltet sich der Korrektheitsbeweis sehr einfach.
Zu (1): Angenommen, es sei $(i\,,j) \notin D_m$, aber $m = x_i b_j$. Dies hieße, dass b_j noch nicht die Zeilen 17 bis 19 durchlaufen hätte. Diese Situation kann jedoch nicht auftreten, da $b_i < m$ und die b_i nach Satz 22 in lückenloser, aufsteigender Reihenfolge berechnet werden. Also gilt (1).
Zu (2): Nach der Definition von E folgt unmittelbar, dass $m \in E$. Die Aussage über die Normalform folgt dann mit Lemma 24. Nach Ausführung von Zeile 9 gilt $\mathcal{L}_{\mathrm{NF}}(m) = v_m$.
Zu (3): Dies folgt direkt aus Satz 22. Nach Ausführung von Zeile 14 gilt dann $\mathcal{L}_{\mathrm{NF}}(m) = v_m$.
Zu (4): Dass $m \in S$ gilt, folgt wiederum aus Satz 22. Nach Definition von S folgt dann, dass $\overline{m} \in E \cup S$ und wegen Satz 22 sogar $\overline{m} \in \overline{S} \cup \overline{E}$. Also kann Zeile 22 ausgeführt werden. Die Aussage über die Normalform folgt mit Lemma 24.
Zu Zeile 23 ist folgendes zu bemerken: Ist $\mathrm{NF}(m) = \sum \lambda_i m_i$, so folgt wegen $\overline{m} \in \overline{E} \cup \overline{S}$, dass $\overline{m} > b_i$ für alle i mit $\lambda_i \neq 0$. Insbesondere gilt $x_k b_i < x_k \overline{m} = m$.
Da $x_k b_i \in N \cup R$, wurde $x_k b_i$ in einem vorherigen Durchlauf schon bearbeitet. Damit wurde also für alle i mit $\lambda_i \neq 0$ die Spalte i in die Matrix $\mathcal{M}_k$ eingefügt. Dies sind genau die Spalten, die man braucht, um $\mathcal{M}_k \cdot v_{\overline{m}}$ ausrechnen zu können.
Somit gilt nach der Ausführung von Zeile 23 $\mathcal{L}_{\mathrm{NF}}(m) = v_m$.
Zu (5): Aus (1)und der Spezifikation von **insertcols** folgt, dass $\mathcal{M}_i^{(j)} = v_m$. Die Aussage über $\mathcal{L}_{\mathrm{NF}}(m)$ wurde in den Schritten oben jeweils gezeigt.
Insgesamt folgt aus den Invarianten, dass die Matrizen mit den richtigen Spalten aufgefüllt werden. Wiederum mit Satz 22 folgt, dass der Algorithmus nach genau k Schritten mit dem korrekten Ergebnis terminiert. Damit ist der Satz bewiesen. □

Algorithmus 4: multmat.
Eingabe: *Sei $<$ eine Wohlordnung auf $K[x_1, \ldots, x_n]$ und $F \subset P_n$ eine normierte, reduzierte Gröbnerbasis von $\langle F \rangle$.*
Ausgabe: *Ein $d \in \mathbb{N}$ mit $\operatorname{vdim}(\langle F \rangle) = d$ und $\mathcal{M}_1, \ldots, \mathcal{M}_n \in \operatorname{Mat}(d \times d, K)$, so dass*

$$\mathcal{L}_{\mathrm{NF}}(x_1^{\alpha_1} \cdots x_n^{\alpha_n}) = \mathcal{M}_1^{\alpha_1} \cdots \mathcal{M}_n^{\alpha_n} \cdot e_1$$

Algorithmus:

$(d, \mathcal{M}_1, \ldots, \mathcal{M}_n) = \mathbf{multmat}(F, <)$

- $C \leftarrow \{1\}$
- **while** $(C \neq \{\})$
  - $m \leftarrow \min_< C$
  - $C \leftarrow C \setminus \{m\}$
  - • (1) $D_m = \{(i, j) \mid m = x_i b_j\}$
  - **if** $(\mathbf{vars}(m) = \mathrm{Z}_m)$
    - **if** $(\exists f \in F \mid \mathrm{L}(f) = m)$
      - • (2) $m \in E$ und $\mathrm{NF}(m) = -\,\mathrm{c}(f)^{-1}\,\mathrm{tail}(f)$
      - $\overline{E} \leftarrow \overline{E} \cup \{m\}$
      - **let** $-\,\mathrm{c}(f)^{-1}\,\mathrm{tail}(f) = \lambda_1\, b_1 + \cdots + \lambda_d\, b_d$
      - $v_m \leftarrow \lambda_1\, e_1 + \cdots + \lambda_d\, e_d$
      - $\mathbf{insertcols}(D_m, v_m)$
    - **else**
      - • (3) $m \in N$ und $\mathrm{NF}(m) = m$
      - $d \leftarrow d + 1$
      - $b_d \leftarrow m$
      - $v_m \leftarrow e_d$
      - $\mathbf{insertcols}(D_m, v_m)$
      - **for** $(i = 1, \ldots, n)$
        - $C \leftarrow C \cup \{x_i \cdot m\}$
        - $D_{x_i \cdot m} \leftarrow D_{x_i \cdot m} \cup \{(i, d)\}$
        - $\mathrm{Z}_{x_i \cdot m} \leftarrow \mathrm{Z}_{x_i \cdot m} + 1$
  - **else**
    - • (4) $m = \overline{m} x_k \in S$, $\mathrm{NF}(\overline{m}) = \sum \lambda_i b_i$ und $\mathrm{NF}(m) = \sum \lambda_i \,\mathrm{NF}(x_k b_i)$
    - $\overline{S} \leftarrow \overline{S} \cup \{m\}$
    - **find** $\overline{m} \in \overline{E} \cup \overline{S}$ **with** $m = x_k \cdot \overline{m}$
    - $v_m \leftarrow \mathcal{M}_k \cdot v_{\overline{m}}$
    - $\mathbf{insertcols}(D_m, v_m)$
  - • (5) $\forall\, b_j$ mit $m = x_i b_j$: $\mathcal{M}_i^{(j)} = v_m$ und $\mathcal{L}_{\mathrm{NF}}(m) = v_m$.
- **return** $(d, \mathcal{M}_1, \ldots, \mathcal{M}_n)$ •

In der Praxis hat sich gezeigt, dass die Multiplikationsmatrizen oft nur sehr dünn besetzt sind. Da ihre Größe durch die Vektorraumdimension bestimmt ist und damit nach oben nicht begrenzt ist, ist es sinnvoll, eine speicherplatzsparende Repräsentation der Matrizen zu implementieren. Da außerdem der Algorithmus so angelegt ist, dass die Matrizen spaltenweise erzeugt werden, kann man sie auch spaltenorientiert aufbauen und innerhalb einer Spalte nur diejenigen Einträge speichern, die sich von Null unterscheiden. Ferner existieren die gleichen Spalten eventuell in mehreren Matrizen, je nachdem, wieviele Tupel in D_m enthalten sind, so dass ein mehrfaches Abspeichern überflüssig ist. Verschiedene Matrizen können sich gemeinsame Spalten einfach teilen. Wie dies hier konkret gelöst ist, findet sich in den Kommentaren zum Quelltext (Seite 89).

Für eine effiziente Implementierung ist es wichtig, die Menge C der noch zu bearbeitenden Kandidaten geschickt zu verwalten, da diese ebenfalls sehr groß werden kann. Es ist sicher nicht sinnvoll, in jedem Schritt das Minimum der Menge erneut zu berechnen (Zeile 3). Vielmehr wird man sie als sortierte Liste implementieren und neue Elemente, die der Liste hinzugefügt werden müssen, gleich an die richtige Stelle einsetzen. Erzeugt man die neuen Elemente noch in aufsteigender Reihenfolge, so kann die Liste von vorne an durchlaufen werden, und ein Element nach dem anderen eingesetzt werden. Hierzu siehe auch Seite 93.

Berechnung der Gröbnerbasis

Nun erfolgt der finale Schritt auf dem Weg zur Berechnung einer reduzierten Gröbnerbasis. Die abschließende Aufgabe lautet: Gegeben ein definierendes System $\mathcal{L} : P_n \to K^d$ eines nulldimensionalen Ideals I in Form von Multiplikationsmatrizen, gegeben eine Wohlordnung, finde die normierte, reduzierte Gröbnerbasis von I bezüglich dieser Ordnung.

Die Vorgehensweise ist seit der Motivation zu Beginn dieses Kapitels klar: Man versucht, die Zerlegung (N, E, S) zu finden, wobei man sich wieder des Satzes 22 bedient. Der dort verlangte Entscheidungsmechanismus wird nun wie folgt realisiert: Sei $m \in N \cup E$ und sei $\{m_1, \ldots, m_k\} = \{m' \mid m' < m\}$. Dies ist genau die Situation, die Satz 22 entspricht. Wie gesehen ist $m \in E$ genau dann, wenn $\{m_1, \ldots, m_k\} \cup \{m\}$ linear unabhängig modulo I ist (vergleiche Seite 39). Sei nun $v = \mathrm{L}(m)$ und $v_i = \mathrm{L}(m_i)$ für alle i. Aus der Definition und der Linearität von $\mathcal{L}$ folgt: Die Menge $\{m_1, \ldots, m_k\} \cup \{m\}$ ist genau dann linear unabhängig modulo I, wenn die Menge $\{v_1, \ldots, v_k\} \cup \{v\}$ linear unabhängig über K ist. Somit besteht der Entscheidungsmechanismus gerade darin, die Menge $\{\mathrm{L}(m_1), \ldots, \mathrm{L}(m_k)\} \cup \{\mathrm{L}(m)\}$ auf lineare Abhängigkeit über K zu testen, also kann man Satz 22 anwenden und die Zerlegung finden. Da man damit aber für jedes $m \in E$ auch ein $f \in I$ mit $\mathrm{L}(f) = m$ gefunden hat, ist nach Lemma 11 aus Kapitel 3 die reduzierte Gröbnerbasis gefunden.

Der Algorithmus ist auf die Verwendung von Multiplikationsmatrizen zugeschnitten: Zu jedem $m \in N \cup R$, das er bearbeitet, merkt er sich ein Paar (i, k), so dass $m = x_i b_k$. Dann ist $\mathcal{L}(m) = \mathcal{M}_i \mathcal{L}(b_k)$, und $\mathcal{L}(b_k)$ wurde in einem vorherigen Schritt berechnet und gespeichert.

Bemerkungen zum Algorithmus

Wiederum wird ein Großteil der Variablen aus Satz 22 und dem Algorithmus **multmats** verwendet.

- $D_m \in \mathbb{N} \times \mathbb{N}$ für $m \in N \cup R$ enthält im Gegensatz zum Algorithmus **multmats** hier nur noch ein einziges Tupel (i, j), so dass $m = x_i b_j$. Dieses Tupel wird verwendet, damit die geeignete Multiplikationsmatrix gefunden wird, mit der man die Vektorraumrepräsentation von m berechnen kann, wie es in Zeile 7 geschieht. Startwert: $D_m = \{\}$ für alle $m \in N \cup R$.
- $v_1, \ldots, v_d \in K^k$ sind die Vektorraumrepräsentationen von $b_1, \ldots, b_d$, es ist also $\mathcal{L}(b_i) = v_i$.
- $v \in K^k$ enthält die Vektorraumrepräsentation von m, das im aktuellen Schritt bearbeitet wird, d.h. es ist $\mathcal{L}(m) = v$. Dieses v gilt es, in Zeile 11 mit Hilfe der $v_1, \ldots, v_d$ linear zu Null zu

kombinieren.

• $F \subset P_n$ schließlich wird nach und nach mit den Elementen der reduzierten Gröbnerbasis aufgefüllt und am Ende als Ergebnis zurückgegeben. Startwert: $F = \{\}$.

Die lineare Unabhängigkeit wird mit der Funktion

$$(w, p) = \mathbf{gaussreduce}(v_1, \ldots, v_s, v)$$

getestet, die folgendermaßen spezifiziert ist: Gegeben $v_1, \ldots, v_s \in K^d$ und ein $v \in K^d$, finde ein $w \in K^d$ und ein $p \in K^s$, so dass gilt:

a) $w = \sum_{i=1}^{s} p^{(i)} v_i + v$.

b) Es ist $w = 0$ genau dann, wenn $\{v_1, \ldots, v_s\} \cup \{v\}$ linear abhängig über K ist.

Der Algorithmus hierfür ist nicht ausformuliert, da es sich schlicht um Gauss-Elimination handelt.

Satz 26. *Sei $I \subset P_n$ ein nulldimensionales Ideal, sei $<$ eine Wohlordnung, sei $\mathcal{L} : P_n \to K^d$ ein definierendes System $\mathcal{L}$ von I und seien $\mathcal{M}_1, \ldots, \mathcal{M}_n \in \mathrm{Mat}(d \times d, K)$ die Multiplikationsmatrizen von $\mathcal{L}$. Sei (N, E, S) die Zerlegung von I und sei $|N \cup E \cup S| = k$. Dann berechnet der Algorithmus* **fglm** *(Algorithmus 5) in genau k Schritten die normierte, reduzierte Gröbnerbasis von I.*

Beweis.

Zu (1): Dies folgt unmittelbar aus Zeile 20.

Zu (2): Folgt aus (1) und der Definition der Multiplikationsmatrizen.

Zu (3): Da $w = 0$ folgt aus der Spezifikation von **gaussreduce**, dass $v = \sum p^{(i)} v_i$. Wegen (2) gilt $\mathcal{L}(m) = \sum p^{(i)} \mathcal{L}(b_i)$, also $m - \sum p^{(i)} b_i \in I$. Somit ist $m \in \mathrm{L}(I)$ und da nach Satz 22 ferner $m \in N \cup E$ gilt, folgt dann $m \in E$.

Nach Satz 22 werden alle $m \in E$ gefunden und mit (2) folgt zusammen mit Lemma 11 aus Kapitel 3, dass F die reduzierte Gröbnerbasis ist, wenn der Algorithmus nach genau k Schritten terminiert. Damit ist der Satz bewiesen. □

In der konkreten Realisierung wurde der Gaussreduktionsschritt leicht abgewandelt. Um auf lineare Unabhängigkeit zu testen, muss die Matrix

$$M = \begin{pmatrix} {v_1}^t \\ \vdots \\ {v_s}^t \\ v^t \end{pmatrix}$$

auf obere Dreiecksgestalt gebracht werden. Nun ist es aber unnötig, in jedem Schritt die Vektoren $v_1, \ldots, v_s$ erneut auf Dreiecksgestalt zu bringen, da man dies ja schon vorher berechnet hat. Daher wird man sich zu jedem v_i ein $\overline{v}_i$ merken, so dass

$$\overline{M} = \begin{pmatrix} {\overline{v}_1}^t \\ \vdots \\ {\overline{v}_s}^t \end{pmatrix}$$

obere Dreiecksgestalt hat. Die hierzu nötigen Permutationen werden in gesonderten Vektoren gespeichert. Für die Gauss-Reduktion wurde eine eigene Klasse implementiert, die unter anderem diesen Verwaltungsaufwand übernimmt. Die Definition dieser Klasse und Erklärungen dazu findet man ab Seite 83.

Algorithmus 5: fglm.
Eingabe: *Ein* $k \in \mathbb{N}$ *und Multiplikationsmatrizen* $\mathcal{M}_1, \ldots, \mathcal{M}_n \in \mathrm{Mat}(k \times k, K)$ *eines definierenden Systems* $\mathcal{L}$ *für ein nulldimensionales Ideal* $I \subset K[x_1, \ldots, x_n]$. *Ferner eine Wohlordnung* $<$.
Ausgabe: *Ein* $F \subset I$, *so dass* F *die normierte, reduzierte Gröbnerbasis von* I *bezüglich* $<$ *ist.*
Algorithmus:

$(F) = \mathbf{fglm}(k, \mathcal{M}_1, \ldots, \mathcal{M}_n, <)$

1 $\quad C \leftarrow \{1\}$
2 $\quad$ **while** $(C \neq \{\})$
3 $\qquad m \leftarrow \min_< C$
4 $\qquad C \leftarrow C \setminus \{m\}$
$\qquad$ • (1) $\;D_m = (i\,, j)$ und $m = x_i b_j$
5 $\qquad$ **if** $(m \neq 1)$
6 $\qquad\quad (i\,, j) \leftarrow D_m$
7 $\qquad\quad v \leftarrow \mathcal{M}_i\, v_j$
$\qquad\quad$ • (2) $\;\mathcal{L}(m) = v$
8 $\qquad$ **else**
9 $\qquad\quad v \leftarrow e_1$
10 $\qquad$ **if** $(\mathbf{vars}(m) = \mathrm{Z}_m)$
11 $\qquad\quad (w, p) = \mathbf{gaussreduce}(v_1, \ldots, v_d, v)$
12 $\qquad\quad$ **if** $(w = 0)$
$\qquad\qquad$ • (3) $\;m \in E$ und $m - \sum_{i=1}^{d} p^{(i)} b_i \in I$
13 $\qquad\qquad F \leftarrow F \cup \{m - \sum_{i=1}^{d} p^i b_i\}$
14 $\qquad\quad$ **else**
$\qquad\qquad$ • (4) $\;m \in N$
15 $\qquad\qquad d \leftarrow d + 1$
16 $\qquad\qquad b_d \leftarrow m$
17 $\qquad\qquad v_d \leftarrow v$
18 $\qquad\qquad$ **for** $(i = 1, \ldots, n)$
19 $\qquad\qquad\quad C \leftarrow C \cup \{x_i \cdot m\}$
20 $\qquad\qquad\quad D_{x_i \cdot m} \leftarrow (i\,, d)$
21 $\qquad\qquad\quad \mathrm{Z}_{x_i \cdot m} \leftarrow \mathrm{Z}_{x_i \cdot m} + 1$
22 $\quad$ **return** (F) •

Es hat sich als sehr effektiv erwiesen, bei Rechnungen über $\mathbb{Q}$ sofort dafür zu sorgen, dass keine Brüche auftreten können. Da die Matrizen eventuell Brüche enthalten, ist es günstig, nach der Multiplikation (Zeile 7) den Vektor durch eine geeignete skalare Multiplikation ganzzahlig zu machen. Implementiert man den Gauss-Algorithmus dann so, dass er das Pivotelement nicht auf 1 normiert, treten in der weiteren Berechnung keine Brüche mehr auf. Die Pivotstrategie selbst hat keinen signifikanten Einfluß auf die Berechnungsdauer gezeigt.

Um nicht Gefahr zu laufen, dass die Koeffizientengröße im Laufe der Rechnung explodiert, müssen die Vektoren an geeigneten Stellen immer wieder mit dem ggT ihrer Einträge gekürzt werden. Wie oft und an welchen Stellen dies geschehen sollte, muß man genau abwägen, um nicht durch häufiges aufwendiges ggT-Berechnen zu viel Zeit zu verlieren.

Als sehr vorteilhaft hat sich die Implementierung eines Referenzmodells für die Vektoren erwiesen. Dadurch kann unnötiges Kopieren der Vektorelemente (die bei Rechnungen über $\mathbb{Q}$ sehr groß werden können) vermieden werden. Einzelheiten hierzu findet man ab Seite 78.

Die Bemerkung zum Algorithmus **multmats** bezüglich der Kandidatenliste (Seite 48) gilt hier ebenfalls.

Etwas störend am Algorithmus erscheint Zeile 5: Man muss in jedem Durchlauf den Test durchführen, obwohl er nur im ersten Schritt zutreffen kann. Man wird daher den Algorithmus so implementieren, dass man den ersten Durchlauf mit $m = 1$ vorwegnimmt, und die Variablen so initialisiert, als ob dieser schon ausgeführt worden wäre.

Bemerkung
Sieht man sich die beiden Algorithmen **multmats** und **fglm** genau an, so stellt man fest, dass sie an keiner Stelle wirklich Polynomarithmetik verwenden. Vor allem durch Lemma 24 ist es gelungen, sämtliche Berechnungen auf dem Grundkörper K durchzuführen. Dieser Gesichtspunkt kann unter dem Implementierungsaspekt von Bedeutung sein.

„Bitte warten Sie, bis das Gerät nicht mehr funktioniert,
dann Geld hinein werfen.“
Tafel an einem Popcorn-Automaten in Montana, Schweiz

Verallgemeinerungen

Das Kapitel über die Verallgemeinerungen gliedert sich in zwei Abschnitte. Im ersten Teil wird der FGLM-Algorithmus so abgeändert, dass er auch Gröbnerbasen von homogenen Idealen berechnen kann. Im zweiten Teil werden mit Hilfe der FGLM-Methoden Monome derart linear kombiniert, dass ihre Summe in einem Ideal liegt.

Der homogene Fall

Ein Nachteil des FGLM-Algorithmus liegt darin, dass man ihn nur auf nulldimensionale Ideale anwenden kann. Diese Restriktion ist eigentlich unumgänglich, denn sobald das Ideal nicht mehr nulldimensional ist, ist die Zerlegung nicht mehr endlich, und der Algorithmus terminiert nicht mehr. Betrachtet man etwa $I = \langle\, xy,\ x^3 + x \,\rangle \subset K[x, y]$ bezüglich der lexikografischen Ordnung, so ist $y^k \in N$ für alle $k \in \mathbb{N}$, d.h. die Menge $\{y^k \mid k \in \mathbb{N}\}$ ist linear unabhängig modulo I. Nun ist aber $y^k < x^l y^m$ für $l > 0$ und $m \geq 0$, der FGLM-Algorithmus würde also zuerst in y-Richtung wandern, kann dort aber natürlich keine lineare Abhängigkeit finden. Es gibt keine Grenze, die a priori vorschreibt, wie weit man in y-Richtung maximal gehen muss, um sicherzustellen, dass man keine lineare Abhängigkeit mehr finden wird. Der tieferliegende Grund, warum der Algorithmus in diesem Fall fehlschlägt, liegt darin, dass

$$\widetilde{N} = \{m \in M_n \mid m \in N, m < \mathrm{L}(f_i) \text{ für alle } i\}$$

hier nicht endlich ist (da z.B. wie gesehen $\{y^k \mid k \in \mathbb{N}\} \subset \widetilde{N}$). Der FGLM-Algorithmus muss aber gerade alle Elemente aus $\widetilde{N}$ bearbeiten, um die Ecken und damit die Gröbnerbasis zu finden.

Ein erster Versuch besteht darin zu bewirken, dass $|\widetilde{N}| < \infty$. Dazu sei $I \subset P_n$ ein beliebiges Ideal, sei $<$ eine gradorientierte Ordnung und sei $F \subset I$ die reduzierte Gröbnerbasis von I bezüglich $<$. Sei wie oben

$$\widetilde{N} = \{m \in M_n \mid m \in N, m < \mathrm{L}(f) \text{ für alle } f \in F\}\ .$$

Da $<$ gradorientiert ist, gilt: Ist $m \in \widetilde{N}$, so ist

$$\mathrm{grad}(m) \leq \max\big(\,\mathrm{grad}\big(\mathrm{L}(f)\big) \mid f \in F\,\big)\ ,$$

der Grad aller Monome in $\widetilde{N}$ ist also beschränkt. Dann muss $|\widetilde{N}|$ endlich sein. Will man nur Gröbnerbasen bezüglich gradorientierter Ordnungen mit Hilfe von FGLM-Methoden berechnen, hat man damit die erste Hürde schon genommen.

Doch das nächste Hindernis folgt sogleich: Ist I nicht nulldimensional, so gibt es kein endliches definierendes System $\mathcal{L}$ für I (Satz 23 aus Kapitel 5), und folglich kann auch nicht mit den handlichen Multiplikationsmatrizen gearbeitet werden. Benutzt man etwa $\mathcal{L}_{\mathrm{NF}_{<_1}}$, so kommt man nicht umhin, wirklich die Normalform der Monome bezüglich einer Ordnung $<_1$ auszurechnen. Das kann durchaus eine sehr zeitaufwendige Angelegenheit sein, die im nulldimensionalen Fall gerade vermieden werden konnte, wie Lemma 24 aus Kapitel 5 gezeigt hat. Dass $\mathrm{L}(m)$ ein Element eines Vektorraumes von unendlicher Dimension ist, bringt dagegen kein wirkliches Problem, da immer nur endliche viele Komponenten von Null verschieden sind. Hierauf wird gleich noch einmal eingegangen.

Problematisch gestaltet sich zu guter Letzt auch das Abbruchkriterium. Im nulldimensionalen Fall hört man mit der Arbeit auf, wenn man alle nicht reduzierbaren Monome gefunden hat, und kann dann sicherstellen (ohne dies explizit überprüfen zu müssen), dass die berechneten Polynome eine reduzierte Gröbnerbasis bilden. Im allgemeinen Fall ist die Situation wieder weniger schön: Hier kann man (für gradorientierte Ordnungen) zwar a priori sicherstellen, dass der Algorithmus nach endlich vielen Schritten terminiert, aber die einzige Möglichkeit festzustellen, ob man fertig ist, lautet: Nachrechnen. Wann immer man ein neues Basispolynom findet, muss man überprüfen, ob die bis hier gesammelten Polynome eine Gröbnerbasis des Ideals bilden. Dies wiederum bedeutet, viele Normalformen berechnen zu müssen.

Fazit: Für allgemeine Ideale und gradorientierte Ordnungen sind die FGLM-Methoden zwar anwendbar, jedoch kommen die Vorteile, die sie im nulldimensionalen Fall bringen, nicht zum Zuge. Der rechnerische Aufwand ist einfach zu hoch, als dass Hoffnung auf einen Geschwindigkeitsvorteil gegenüber dem Buchberger-Algorithmus bestünde. Treibt man die Theorie noch etwas weiter, so kann man sogar auf die Einschränkung auf gradorientierte Ordnungen verzichten und FGLM-Methoden für den ganz allgemeinen Fall verwenden. Das hat jedoch rein theoretischen Nutzen, denn Geschwindigkeitsvorteile wird diese Methode sicher nicht bringen. Näheres zu solchen Verallgemeinerungen findet man bei M. Niermann [15]. Deshalb wurde auf die Implementierung eines Algorithmus, der beliebige Ideale behandelt, verzichtet.

Beschränkt man sich dagegen auf homogene Ideale (vergleiche Kapitel 2), so fallen die Verschlechterungen gegenüber dem nulldimensionalen Fall geringer aus. Man muss zwar immer noch auf Multiplikationsmatrizen verzichten und Normalformen explizit ausrechnen, jedoch hat man wieder ein pfiffiges Abbruchkriterium zur Verfügung. Statt direkt zu überprüfen, ob man schon eine Gröbnerbasis gefunden hat, wird nun die Hilbertfunktion (Kapitel 3) die Antwort geben.

Die Beschränkung auf homogene Ideale ist dabei nicht so schwerwiegend wie etwa die auf nulldimensionale Ideale. Ist nämlich ein beliebiges Ideal gegeben, so kann man dieses bezüglich einer neuen Unbestimmten homogenisieren, hiervon die Gröbnerbasis berechnen und anschließend das Ergebnis wieder dehomogenisieren. Ob der Geschwindigkeitsvorteil, den homogene Ideale mit sich bringen, groß genug ist, um den Mehraufwand zu kompensieren, den eine zusätzliche Variable erzeugt, muss sich im Einzelfall zeigen.

Nachfolgend sollen nun die Änderungen motiviert werden, die man im Vergleich zum nulldimensionalen Fall am FGLM-Algorithmus vorzunehmen hat. Sei $I \subseteq P_n$ ein homogenes Ideal, sei $<$ eine Wohlordnung, sei $F = \{f_1, \ldots, f_s\} \subset I$ die normierte, reduzierte Gröbnerbasis von I und sei schließlich (N, E, S) die Zerlegung von I. Satz 9 aus Kapitel 2 sagt, dass F dann aus homogenen Polynomen besteht. Jedes $f_i \in F$ ist also eine K-Linearkombination aus Monomen vom Grad k_i. Insbesondere gilt

$$\mathrm{tail}(f_i) \in \mathrm{span}_K \left(m \in N \mid \mathrm{grad}(m) = k_i\right) .$$

Im nulldimensionalen Fall galt: Ist $m \in N \cup E$, so ist $m \in N$ genau dann, wenn die Menge $\{m' \in N \mid m' < m\} \cup \{m\}$ linear unabhängig modulo I ist (vergleiche Seite 39). Für homogene Ideale kann man dies abschwächen zu der Bedingung: Ist $m \in N \cup E$, so ist $m \in N$ genau dann, wenn die Menge

$$\Phi = \{m' \mid m' < m,\ \mathrm{grad}(m') = \mathrm{grad}(m)\} \cup \{m\}$$

linear unabhängig modulo I ist. Der Beweis ist analog zum nulldimensionalen Fall, wenn man beachtet, dass $\operatorname{tail}(f_i)$ homogen vom Grad k_i ist. Da Φ nur Monome gleichen Grades enthält, ist Φ eine endliche Menge, so dass man die lineare Unabhängigkeit in endlicher Zeit überprüfen kann.

Man muss nicht für jedes $m \in N \cup E$ diesen Test auf lineare Unabhängigkeit durchführen (was auch gar nicht möglich wäre, da $|N \cup E| = \infty$). Um das zu sehen, sei $k \in \omega$, so dass $\operatorname{grad}(f) \neq k$ für alle $f \in F$. Insbesondere ist dann $\operatorname{grad}(\mathrm{L}(f)) \neq k$ für alle $f \in F$ und somit

$$E \cap \{m \in M_n \mid \operatorname{grad}(m) = k\} = \{\} .$$

Ist also $m \in N \cup E$ mit $\operatorname{grad}(m) = k$, so folgt automatisch $m \in N$. Es genügt, sich auf solche $m \in N \cup E$ zu beschränken, für die $\operatorname{grad}(m) = \operatorname{grad}(f)$ für ein $f \in F$. Über die Hilbertfunktion kann man gerade alle $k \in \omega$ erhalten, für die es ein $f \in F$ gibt, mit $\operatorname{grad}(f) = k$. (Denn beachte: Es soll F ja gerade berechnet werden, den Test $\operatorname{grad}(m) = \operatorname{grad}(f)$ für ein $f \in F$ kann man gar nicht ausführen.)

Angenommen, man hat die Polynome der reduzierten Gröbnerbasis bis zu einem Grad τ schon berechnet, d.h. es sei

$$F_\tau = \{f \in F \mid \operatorname{grad}(f) \leq \tau\}$$

bekannt. In dieser Situation gilt folgender Satz (vergleiche auch Bild 7):

Satz 27. *Sei $I \subseteq P_n$ ein homogenes Ideal, sei $<$ eine Wohlordnung und sei $F \subset I$ die normierte, reduzierte Gröbnerbasis von I. Für $\tau \in \omega$ sei*

$$F_\tau = \{f \in F \mid \operatorname{grad}(f) \leq \tau\} ,$$

und sei h die Hilbertfunktion von $\langle \mathrm{L}(F) \rangle$ und $\overline{h}$ die Hilbertfunktion von $\langle \mathrm{L}(F_\tau) \rangle$. Dann gilt:

$$\begin{aligned} \langle \mathrm{L}(F_\tau) \rangle_{\leq s} &= \langle \mathrm{L}(F) \rangle_{\leq s} && \textit{für alle } s \leq \tau , \\ \overline{h}(s) &= h(s) && \textit{für alle } s \leq \tau . \end{aligned}$$

Ist ferner $h \neq \overline{h}$, so sei $\sigma = \min(s \in \omega \mid h(s) \neq \overline{h}(s))$. Dann gilt weiter:

a) Für alle $f \in F$ ist entweder $\operatorname{grad}(f) \leq \tau$ oder $\sigma \leq \operatorname{grad}(f)$.

b) Es gibt ein $f \in F$ mit $\operatorname{grad}(f) = \sigma$.

Beweis.
Es gilt

$$\begin{aligned} f \in \langle \mathrm{L}(F) \rangle_{\leq s} &\Leftrightarrow \operatorname{grad}(f) \leq s \text{ und } f = \sum p_i \mathrm{L}(f_i) \\ &\Leftrightarrow f = \sum_{f_i \in F} p_i \mathrm{L}(f_i) \text{ und } \operatorname{grad}(p_i f_i) \leq s \, \forall i \\ &\Leftrightarrow f = \sum_{\operatorname{grad}(\mathrm{L}(f_i)) \leq s} p_i \mathrm{L}(f_i) \text{ und } \operatorname{grad}(p_i f_i) \leq s \, \forall i \\ &\Leftrightarrow f = \sum_{f_i \in F_\tau} p_i \mathrm{L}(f_i') \text{ und } \operatorname{grad}(p_i \mathrm{L}(f_i')) \leq s \, \forall i \;\Leftrightarrow\; f \in \langle \mathrm{L}(F_\tau) \rangle_{\leq s} \end{aligned}$$

Damit folgt, dass $h(s) = \overline{h}(s)$ für alle $s \leq \tau$.
Sei nun $h \neq \overline{h}$ und sei $\sigma = \min(s \in \omega \mid h(s) \neq \overline{h}(s))$. Es gibt dann ein $f \in F$ mit $\operatorname{grad}(f) > \tau$, da sonst $F = F_\tau$, d.h. $h = \overline{h}$ wäre.
Annahme: Es gäbe ein $f \in F$ mit $\tau < \operatorname{grad}(f) < \sigma$. Da F reduziert ist, gilt $\mathrm{L}(f') \nmid \mathrm{L}(f)$ für alle $f' \in F$ mit $f' \neq f$, insbesondere ist $\mathrm{L}(F) \notin \langle \mathrm{L}(F_\tau) \rangle$, also

$$\mathrm{L}(f) \notin \langle F_\tau \rangle_{\leq \operatorname{grad}(f)} = \langle \mathrm{L}(F) \rangle_{\leq \operatorname{grad}(f)} .$$

$\Rightarrow$ Widerspruch, da $\mathrm{L}(f) \in \langle\, \mathrm{L}(F) \,\rangle$. Also ist $\operatorname{grad}(f) \geq \sigma$ für alle f mit $\operatorname{grad}(f) > \tau$.
Nach Voraussetzung existiert ein $m \in \langle\, \mathrm{L}(F) \,\rangle$ mit $m \notin \langle\, \mathrm{L}(F_\tau) \,\rangle$ und $\operatorname{grad}(m) = \sigma$, also auch ein $f \in \mathrm{L}(F)$ mit $\mathrm{L}(f) \,|\, m$. Wäre nun $\mathrm{L}(f) \neq m$, so wäre $\operatorname{grad}(f) < \sigma$, also nach dem eben Gesehenen $\operatorname{grad}(f) \leq \tau$. Somit wäre $f \in \mathrm{L}(F_\tau)$ und dann $m \in \langle\, \mathrm{L}(F_\tau) \,\rangle \Rightarrow$ Widerspruch. Also gilt $m = \mathrm{L}(f)$ und damit ist gezeigt, dass es ein $f \in F$ gibt mit $\operatorname{grad}(f) = \sigma$, womit der Satz bewiesen ist. □

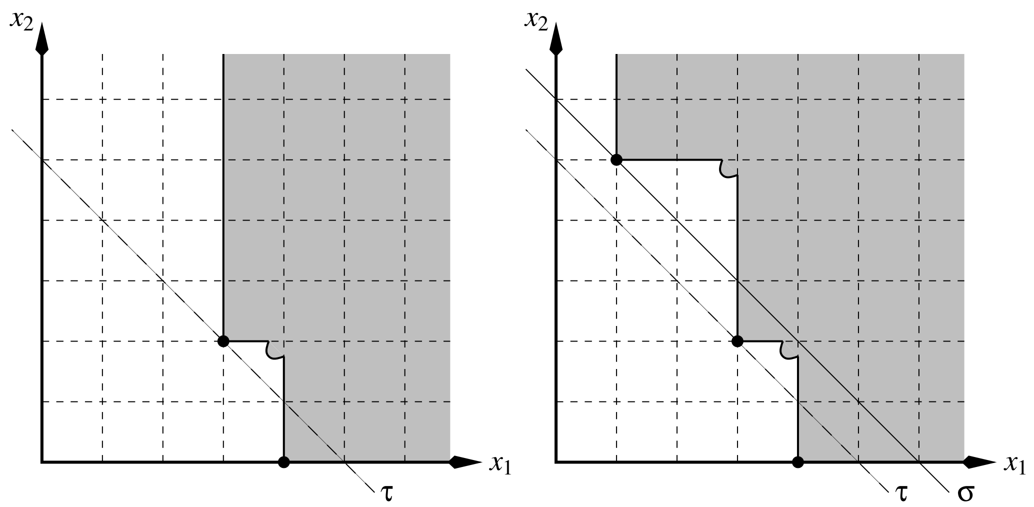


Bild 7: Die vorläufige (links) und die endgültige (rechts) Zerlegung von I.

Satz 27 gibt somit die Anweisung, wie man zu verfahren hat, wenn man die vorläufige Basis F_τ auf dem Weg zu einer vollständigen reduzierten Basis um ein neues Element erweitern möchte: Finde die kleinste Zahl σ, an der sich die beiden Hilbertfunktionen h und $\overline{h}$ unterscheiden und suche das neue Polynom aus der Gröbnerbasis, wie oben beschrieben, unter den Monomen aus $N \cup E$ vom Grad σ. Diese Monome kann man mit Hilfe der vorläufigen Basis F_τ bestimmen. Es ist nämlich

$$\{m \in N \cup E \mid \operatorname{grad}(m) = \sigma\} = \{m \in M_n \mid \operatorname{grad}(m) = \sigma,\ \mathrm{L}(f) \nmid m \text{ für alle } f \in F_\tau\}\ .$$

Der Clou liegt in Satz 18 aus Kapitel 3: Um die Hilbertfunktion eines homogenen Ideals I zu berechnen, reicht es, die Hilbertfunktion des monomialen Ideals $\langle\, \mathrm{L}(G) \,\rangle$ zu suchen, wobei G eine Gröbnerbasis bezüglich einer beliebigen Wohlordnung ist. Hat man eine solche Gröbnerbasis zur Hand, so reicht diese, um h und damit σ berechnen zu können.

Jetzt hat man alles parat, um das Schema zu skizzieren: Gegeben eine reduzierte Gröbnerbasis bezüglich irgendeiner Wohlordnung $<_1$, erstelle damit die Hilbertfunktion h von I. Berechne sukzessive die Gröbnerbasis bezüglich der Ordnung $<$ nach folgenden Schritten:

- Sei $F_\tau = \{f \in F \mid \operatorname{grad}(f) \leq \tau\}$ bekannt. Berechne die Hilbertfunktion $\overline{h}$ von $\langle\, \mathrm{L}(F_\tau) \,\rangle$.
- Sei $\sigma = \min(s \in \omega \,|\, \overline{h}(s) \neq h(s))$. (Beachte: Dann ist $\sigma > \tau$.)
- Sei $\Psi = \{m_1, \ldots, m_k\} = \{m \in M_n \mid \operatorname{grad}(m) = \sigma, \mathrm{L}(f) \nmid m$ für alle $f \in F_\tau\}$.
- Finde in Ψ die linearen Abhängigkeiten modulo I und erweitere F_τ um die entsprechenden Polynome.

Damit hat man F_τ so mit neuen Elementen versehen, dass gilt: $F_\tau = \{f \in F \mid \operatorname{grad}(f) \leq \sigma\}$. Wiederholt man diese Schritte so lange, bis $h(s) = \overline{h}(s)$ für alle s, so ist $F_\tau = F$.

Die letzte noch offene Frage betrifft den Test auf lineare Abhängigkeit modulo I, der natürlich wieder mittels eines definierenden Systems auf lineare Abhängigkeit von Vektoren übersetzt werden soll. Da man zur Berechnung der Hilbertfunktion schon eine Gröbnerbasis bezüglich einer „Ersatzordnung“ $<_1$ benötigt, liegt es auf der Hand, als definierendes System $\mathcal{L} = \mathcal{L}_{\mathrm{NF}<_1}$ zu verwenden. Unglücklicherweise ist das System aber nicht mehr endlich, wenn das Ideal nicht nulldimensional ist; man müsste eigentlich in einem Vektorraum von unendlicher Dimension arbeiten. Doch es ist wieder einmal eine Hintertür offen: Sei $m \in M_n$ und sei $v = \mathcal{L}(m)$. Dann ist

$$\mathrm{NF}(m) = \sum_{k\in\Phi} c_k b_k \quad \text{also} \quad \mathcal{L}(m) = \sum_{k\in\Phi} c_k e_k \,,$$

wobei $\Phi \subset \omega$ endlich ist; die Komponenten von v sind also nur an endlich vielen Stellen von Null verschieden.

In jedem Schritt muss $v_i = \mathcal{L}(m_i)$ für jedes $m_i \in \Psi$ berechnet werden, und da Ψ ebenfalls endlich ist, treten bei allen Vektoren v_i insgesamt nur an endlich vielen Stellen Werte ungleich Null auf. Es genügt somit, sich auf endlich viele Komponenten des unendlich-dimensionalen Vektorraumes $\mathrm{span}_K(N)$ zu beschränken, da an den restlichen Stellen nichts passieren kann. Welche Komponenten man beachten muss, stellt man wie folgt fest: Berechne $p_i = \mathrm{NF}(m_i)$ für alle $m_i \in \Psi$ und sammle alle Monome b_k, die als Summand in den p_i auftauchen. Die entsprechenden e_k-Komponenten sind dann bei der Berechnung zu beachten.

Genau diese Aufgabe erledigt der Algorithmus **getvectorrep** (Algorithmus 6). Zu einer Menge von Polynomen p_i berechnet er die Projektion von $\mathcal{L}(p_i)$ auf die interessanten Komponenten. Beachte: Man ist mal wieder nur an den Vektoren selbst interessiert und nicht daran, wie die Projektion zustandegekommen ist. Daher werden die b_k aus den Normalformen nicht zurückgegeben. Der Algorithmus ist direkt ersichtlich, ein Korrektheitsbeweis erübrigt sich für ihn.

Algorithmus 6: getvectorrep.

Eingabe: *Seien $p_1, \ldots, p_s \in P_n$. Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei $G \subset I$ eine Gröbnerbasis von I bezüglich dieser Ordnung.*

Ausgabe: *Ein minimales $d \in \mathbb{N}$ und $v_1,\ldots,v_s \in K^d$, so dass gilt: Es gibt eine Menge $\{b_1, \ldots, b_d\} \subset M_n$, so dass $\mathrm{NF}_<(p_i) = \sum_{k=1}^d v_i^{(k)} \cdot b_k$.*

Algorithmus:

$(d, v_1, \ldots, v_s) = \textbf{getvectorrep}(p_1, \ldots, p_s, G, <)$

- **for** $(i = 1, \ldots, s)$
  - $f_i \leftarrow \textbf{nf}(p_i, G, <)$
  - **let** $f_i = \sum_{\alpha\in\Phi} c_\alpha x^\alpha$, **where** $c_\alpha \in K^*$
  - $N \leftarrow N \cup \{x^\alpha \mid \alpha \in \Phi\}$
- $d \leftarrow |N|$
- $\{b_1, \ldots, b_d\} \leftarrow N$
- **for** $(i = 1, \ldots, s)$
  - **let** $f_i = \sum_{k=1}^d c_k b_k$, **where** $c_k \in K$
  - $v_i \leftarrow \sum_{k=1}^d c_k e_k$
- **return** $(d, v_1, \ldots, v_s)$ •

Damit ist dann aber auch schon die ganze Arbeit geleistet, um den Algorithmus für homogene Ideale zu beschreiben. Im Rahmen dieser Arbeit soll nicht näher darauf eingegangen werden, wie man die Hilbertfunktion eines Ideals berechnen kann, ein solcher Algorithmus soll als vorausgesetzt gelten. Siehe hierzu etwa Bigatti et al. [3] und Caboara et al. [6].

Bemerkungen zum Algorithmus

• $F \subset P_n$ wird gradweise mit den Elementen der Gröbnerbasis aufgefüllt und wird am Ende als Ergebnis zurückgegeben. Startwert: $F = \{\}$.

• $h, \overline{h} : \omega \to \omega$ sind Hilbertfunktionen. Dabei ist h die Hilbertfunktion von I und $\overline{h}$ wird in jedem Durchgang durch die **repeat**-Schleife als Hilbertfunktion von $\langle F \rangle$ berechnet, vergleiche Zeile 17.

• $\sigma \in \mathbb{N}$ enthält den Grad, in dem im nächsten Schritt Polynome für die Gröbnerbasis zu erwarten sind.

Jeder Schritt durch die **repeat**-Schleife ähnelt dem Algorithmus **fglm**, auch in den verwendeten Variablen:

• $C \subset N \cup R$ enthält die „Kandidaten“ vom Grad σ, aus denen im nächsten Schritt die linearen Abhängigkeiten herausgesucht werden müssen.

• $v_m \in K^\delta$ für $m \in C$ sind die Vektorraumrepräsentationen, d.h. es ist $\mathcal{L}_{\mathrm{NF}}(m) = v_m$ für alle $m \in C$.

• $d \in \mathbb{N}$ zählt für einen Grad die Anzahl der Monome in N, die bislang gefunden wurden.

• $b_1, \ldots, b_d \in M_n$ sind die Monome aus N vom Grade σ, die im Laufe der Berechnung gefunden werden.

• $v_1, \ldots, v_d \in K^\delta$ sind die Vektorraumrepräsentationen der Monome b_i: $\mathcal{L}_{\mathrm{NF}}(b_i) = v_i$. Diese Vektoren stehen **gaussreduce** zum Auffinden von linearen Abhängigkeiten zur Verfügung.

Satz 28. *Sei $I \subset P_n$ ein homogenes Ideal, sei $<$ eine Wohlordnung. Sei ferner $<_1$ eine weitere Wohlordnung und $G \subset I$ eine Gröbnerbasis bezüglich $<_1$. Dann berechnet der Algorithmus* **homogfglm** *(Algorithmus 7) in endlich vielen Schritten die normierte, reduzierte Gröbnerbasis von I bezüglich der Ordnung $<$.*

Beweis.
(1) ergibt sich aus der Definition von N und E und der Tatsache, dass F schon eine partielle Gröbnerbasis ist.
Die **while**-Schleife (Zeile 7 bis 16) entspricht im Wesentlichen dem FGLM-Algorithmus, so dass mit (1) und Satz 27 folgt, dass in Zeile 12 F gerade um Polynome aus der Gröbnerbasis erweitert wird. Ganz analog zum Algorithmus **fglm** ergibt sich, dass auch alle Polynome vom Grad σ aus der Basis gefunden werden.
(2) ergibt sich dann direkt aus Satz 27. Ist $H \subset I$ die normierte, reduzierte Gröbnerbasis von I, so gilt wie gesehen zu jedem Zeitpunkt $F \subseteq H$. Da F bei jedem Durchgang durch die Schleife strikt größer wird, gilt nach endlich vielen Schritten $F = H$ und also $h = \overline{h}$. Daher terminiert der Algorithmus nach endlich vielen Schritten mit dem richtigen Ergebnis und damit ist der Satz bewiesen. □

Lineare Abhängigkeiten

Sei $I \subset K[x_1, \ldots, x_n]$ ein nulldimensionales Ideal. Wie Satz 15 aus Kapitel 3 aussagt, gibt es dann für jede Variable x_i ein univariates Polynom $f_i \in K[x_i]$, so dass $f_i \in I$ (vergleiche

Algorithmus 7: homogfglm.
Eingabe: *Sei $I \subset P_n$ ein Ideal und sei $<$ eine Wohlordnung. Sei ferner $<_1$ eine weitere Wohlordnung und $G \subset I$ eine Gröbnerbasis von I bezüglich $<_1$.*
Ausgabe: *Ein $F \subset I$, so dass F die normierte, reduzierte Gröbnerbasis von I bezüglich $<$ ist.*
Algorithmus:

$(F) = \textbf{homogfglm}(<, G, <_1)$

- $h \leftarrow \mathrm{Hf}_I,\ \overline{h} \leftarrow 0$
- **repeat**
  - $d \leftarrow 0 \in \omega$
  - $\sigma \leftarrow \min(s \in \omega \mid h(s) \neq \overline{h}(s))$
  - $C \leftarrow \{m \in M_n \mid \mathrm{grad}(m) = \sigma,\ \mathrm{L}(f) \nmid m\, \forall f \in F\}$
  - • (1) $C = \{m \in N \cup E \mid \mathrm{grad}(m) = \sigma\}$
  - $(\delta, \{v_m \mid m \in C\}) \leftarrow \textbf{getvectorrep}(C, G, <_1)$
  - **while** $(C \neq \{\})$
    - $m \leftarrow \min_< C$
    - $C \leftarrow C \setminus \{m\}$
    - $(w, p) \leftarrow \textbf{gaussreduce}(v_1, \ldots, v_d, v_m)$
    - **if** $(w = 0)$
      - $F \leftarrow F \cup \{m - \sum_{i=1}^{d} p^{(i)} b_i\}$
    - **else**
      - $d \leftarrow d + 1$
      - $b_d \leftarrow m$
      - $v_d \leftarrow v_m$
  - $\overline{h} \leftarrow \mathrm{Hf}_{\langle \mathrm{L}(F) \rangle}$
  - • (2) Es ist $h = \overline{h}$, oder $\exists\, \overline{\sigma} > \sigma$ mit $\overline{h}(\overline{\sigma}) \neq h(\overline{\sigma})$
- **until** $(h = \overline{h})$
- **return** (F) •

Bedingung (iv) des Satzes). Ziel dieses Abschnittes ist es, einen Algorithmus zu erstellen, der genau diese univariaten Polynome findet.

Ein möglicher Lösungsansatz verwendet Elimination: Sei dazu $<_i$ eine lexikografische Wohlordnung, so dass $x_i <_i x_j$ für alle j mit $j \neq i$, d.h. x_i ist die kleinste Variable. Sei $F_i \subset I$ eine Gröbnerbasis von I bezüglich dieser Ordnung. Dann kann man zeigen, dass $F_i \cap K[x_i]$ eine Gröbnerbasis von $I \cap K[x_i]$ ist, und hat damit ein univariates Polynom $f_i \in K[x_i]$ gefunden, so dass $f_i \in I$. Eine mögliche Lösung des Problems besteht also aus den folgenden Schritten:

- Für jedes $i \in \{1, \ldots, n\}$ berechne bezüglich der Ordnung $<_i$ eine Gröbnerbasis F_i von I.
- Berechne $F_i \cap K[x_i]$ für alle i.

Dies ist jedoch eine sehr aufwendige Angelegenheit, da man n verschiedene Gröbnerbasen berechnen muss und das auch noch bezüglich der im Allgemeinen sehr teuren lexikografischen Ordnung. Mit den Ergebnissen aus den vorherigen Kapiteln hat man aber noch eine andere Berechnungsmöglichkeit zur Auswahl, die wesentlich effizienter sein kann.

Dazu sei $f_i = \sum_{k=0}^{d} c_k x_i^k \in K[x_i]$. Dann ist die Menge $\{1, x_i, x_i^2, \ldots, x_i^d\}$ genau dann linear unabhängig modulo I, wenn $f_i \in I$ ist. Damit ist die Idee aber auch schon klar: Man teste die Menge $\{1, x_i, \ldots, x_i^d\}$ auf lineare Abhängigkeit – natürlich wieder mit Hilfe eines definierenden Systems – und wenn sie noch nicht abhängig ist, erweitere man sie um das Element x_i^{d+1} und beginne von vorne. Dann tritt irgendwann die lineare Abhängigkeit ein, und man hat automatisch die Koeffizienten c_k gewonnen, so das $f = \sum_{k=0}^{d} c_k x_i^k \in I$. Schon ist man fertig.

Der große Vorteil liegt hier wieder darin, dass man die Multiplikationsmatrizen des definierenden Systems $\mathcal{L}$ von I verwenden kann. Denn wie schon beim FGLM-Algorithmus steht man vor der Situation, $\mathcal{L}(x_i^{d+1})$ berechnen zu müssen und $\mathcal{L}(x_i^d)$ im vorherigen Schritt schon gefunden zu haben. Dies erledigen aber doch genau die Multiplikationsmatrizen, man vergleiche Seite 43.

Der oben skizzierte Weg, ein univariates Polynom $f_i \in K[x_i]$ zu finden, ist genau der Weg, den der FGLM-Algorithmus ginge, wenn man ihn bezüglich einer lexikografischen Ordnung $<_i$ anwenden würde, für die $x_i <_i x_j$ für alle $j \neq i$. Man simuliert also im Grunde genommen den FGLM-Algorithmus für eine solche Ordnung, bis man das erste Basispolynom gefunden hat, das dann automatisch univariat ist. Da man aber genau weiss, welches die kleinsten Monome bezüglich dieser Ordnung sind (nämlich $1 < x_i < x_i^2 < \ldots$), kann man auf die Liste der möglichen Kandidaten sowie auf den Zähler, wie oft diese in die Liste eingefügt worden sind, verzichten (siehe Algorithmus **fglm**, Seite 49).

Die Vorgehensweise lautet: Berechne mit dem Algorithmus **multmat** bezüglich einer beliebigen Ordnung die Multiplikationsmatrizen des Ideals und simuliere dann für alle i die ersten Schritte des FGLM-Algorithmus bezüglich einer Ordnung $<_i$, wie sie oben beschrieben ist. Das heißt mit anderen Worten: Finde für alle $i \in \{1, \ldots, n\}$ ein minimales d, so dass die Menge $\{1, x_i, \ldots, x_i^d\}$ linear unabhängig modulo I ist. Beachte nochmals: Die Ordnung $<_i$ taucht nur implizit auf, da man genau weiss, dass $1 <_i x_i <_i x_i^2 <_i \ldots$ die kleinsten Monome sind.

Die Ausformulierung des Algorithmus ist nun leicht. Sie verwendet wieder den Algorithmus **gaussreduce**, wie er in Kapitel 5 (Seite 49) spezifiziert wurde.

Bemerkungen zum Algorithmus

In jedem Durchgang durch die **for**-Schleife (die die Ringvariablen abarbeitet), haben die Variablen folgende Bedeutung:

- $k \in \omega$ zählt die bislang gefundenen linear unabhängigen Vektoren v_j.
- $v_j \in K^d$ ist die Vektorraumrepräsentation von x_i^{j-1}, d.h. es ist $\mathcal{L}(x_i^{j-1}) = v_j$.
- $v \in K^d$ ist der aktuelle Vektor, den es mit Hilfe der $v_1, \ldots, v_k$ zu reduzieren gilt.

Satz 29. *Sei $I \subset P_n$ ein nulldimensionales Ideal, sei $\mathcal{L} : P_n \to K^d$ ein definierendes System von I und seien $\mathcal{M}_1, \ldots, \mathcal{M}_n \in \mathrm{Mat}(d \times d, K)$ die Multiplikationsmatrizen von $\mathcal{L}$. Dann berechnet der Algorithmus* **finduni** *(Algorithmus 8) Polynome $f_1 \in K[x_1], \ldots, f_n \in K[x_n]$, so dass für alle i gilt: f_i hat minimalen Grad mit der Eigenschaft, dass $f_i \in I$.*

Beweis.
Wie im Text schon erwähnt imitiert die **repeat**-Schleife (Zeile 4 bis 11) die ersten Schritte des FGLM-Algorithmus bezüglich der Ordnung $<_i$. Der Satz ergibt sich damit direkt aus dem Beweis von Satz 26 aus Kapitel 5. □

Die Vorteile dieser Methode liegen auf der Hand:

- Die Berechnung der Multiplikationsmatrizen kann bezüglich einer beliebigen Wohlordnung erfolgen, man kann sich eine „billige“ heraussuchen.

Algorithmus 8: finduni.
Eingabe: *Ein $d \in \mathbb{N}$ und Multiplikationsmatrizen $\mathcal{M}_1, \ldots, \mathcal{M}_n \in \mathrm{Mat}(d \times d, K)$ eines definierenden Systems für ein nulldimensionales Ideal $I \subset P_n$.*
Ausgabe: *$f_1 \in K[x_1], \ldots, f_n \in K[x_n]$, so dass für alle i gilt: f_i hat minimalen Grad mit der Eigenschaft, dass $f_i \in I$.*
Algorithmus:

$(f_1, \ldots, f_n) = \mathbf{finduni}(d, \mathcal{M}_1, \ldots, \mathcal{M}_n)$

for $(i = 1, \ldots, n)$
    $v_1 \leftarrow e_1 \in K^d$
    $k \leftarrow 1$
    repeat
        $v \leftarrow \mathcal{M}_i v_k$
        $(w, p) \leftarrow$ **gaussreduce**$(v_1, \ldots, v_k, v)$
        if $(w = 0)$
            $f_i \leftarrow \sum_{j=1}^{k} p^{(j)} x_i^{j-1}$
        else
            $k \leftarrow k + 1$
            $v_k \leftarrow v$
    until $(w = 0)$
return $(f_1, \ldots, f_n)$.

- Da das Ideal nulldimensional ist, kann man hierzu auf den Algorithmus **multmat** zurückgreifen, der vollständig ohne Berechnung von Normalformen auskommt.
- Dieses eine Set von Multiplikationsmatrizen reicht aus, um alle n univariaten Polynome minimalen Grades zu finden.
- Zur Berechnung der univariaten Polynome benötigt man keine Ordnung.

Da man das Ergebnis, das der Algorithmus **finduni** liefert, für die Primärzerlegung eines nulldimensionalen Ideals benötigt, hat man somit nun neben der Eliminationsmethode ein weiteres Werkzeug zur Berechnung zur Verfügung.

Ein ähnliches Problem ergibt sich, wenn man polynomiale Gleichungssysteme lösen möchte, das heißt wenn man Variablen aus einem Ideal eliminieren möchte. Wie dieses Eliminieren von statten geht, sagt die Eliminationstheorie: Man berechnet eine lexikografische Gröbnerbasis F des Ideals I mit $x_1 > \ldots > x_n$ und weiss dann, dass $F \cap K[x_k, \ldots, x_n]$ eine Gröbnerbasis von $I \cap K[x_k, \ldots, x_n]$ ist. Nun ist aber die Berechnung einer lexikografischen Gröbnerbasis gerade über dem Körper $\mathbb{Q}$ meist sehr teuer, da die Koeffizienten sehr groß werden können. Rechnet man dagegen über einem endlichen Körper $\mathbb{Z}/p\mathbb{Z}$ mit einer Primzahl p, so können die Berechnungszeiten drastisch sinken (manchmal wird dann eine Berechnung überhaupt erst möglich).

Somit hat man folgende heuristische Idee: Berechne eine lexikografische Gröbnerbasis F_p von I über $\mathbb{Z}/p\mathbb{Z}$. Berechne damit ein $f_p \in \mathbb{Z}/p\mathbb{Z}[x_n]$, so dass $\langle f_p \rangle = I \cap \mathbb{Z}/p\mathbb{Z}[x_n]$. Sei etwa $f_p = \sum_{i=1}^{d} c_k x_n^k$. Ist $f \in \mathbb{Q}[x_n]$ das entsprechende Polynom bei der Rechnung über $\mathbb{Q}$, das heisst ist $\langle f \rangle = I \cap \mathbb{Q}[x_n]$, so hofft man, dass in f_p und f die gleichen Monome involviert sind und sich lediglich die Koeffizienten unterscheiden. Über $\mathbb{Q}$ muss man also nur noch die Monome $1, x_n, \ldots, x_n^d$ so mit Koeffizienten aus $\mathbb{Q}$ linear kombinieren, dass sie linear abhängig modulo I werden. Damit hat man es vermieden, in $\mathbb{Q}$ eine komplette lexikografische Gröbnerbasis ausrechnen zu müssen. Ob die Annahme, dass in f_p und f die gleichen Monome enthalten sind, richtig ist, kann man nur durch Nachrechnen feststellen. Erweist sie sich als falsch, so findet man eben keine geeigneten Koeffizienten, die die lineare Abhängigkeit erzielen. In diesem Fall kommt man dann mit der Idee nicht weiter und muss im Zweifelsfalle doch direkt eine lexikografische Basis ausrechnen.

Etwas allgemeiner als eben geschildert (das heisst nicht nur univariat) benötigt man also einen Algorithmus, der folgende Aufgabe meistert: Sei $I \subseteq P_n$ ein (nicht notwendig nulldimensionales) Ideal. Zu gegebenen $m_1, \ldots, m_s \in M_n$ finde Koeffizienten $c_1, \ldots, c_s \in K$, so dass gilt: Es ist $\sum c_i m_i \in I$ und $c_i = 0$ für alle i genau dann, wenn die Menge $\{m_1, \ldots, m_s\}$ linear unabhängig modulo I ist.

Die Vorgehensweise ist klar und unterscheidet sich nur in Details von dem bisher Dagewesenen: Die lineare Abhängigkeit wird wieder mittels eines definierenden Systems überprüft, man hat hier allerdings die gleichen Probleme wie im homogenen Fall: Da das Ideal nicht nulldimensional ist, gibt es kein endliches definierendes System, man kann auch keine Multiplikationsmatrizen verwenden. Wie auch schon im homogenen Fall wird man $\mathcal{L}_{\mathrm{NF}}$ als definierendes System benutzen, die Normalformen explizit erstellen und sich auf die (endlich vielen) interessanten Komponenten beschränken (vergleiche Algorithmus 6, Seite 57). Es besteht dennoch Hoffnung auf einen Geschwindigkeitsvorteil gegenüber der direkten Gröbnerbasisberechnung, da man hier nur wirklich die Polynome aufstellt, an denen man interessiert ist. Betreibt man Elimination, kommt es eben gerade nicht auf die komplette Basis an.

Satz 30. *Sei $I \subseteq P_n$ ein Ideal, sei $<$ eine Wohlordnung und sei $F \subset I$ eine Gröbnerbasis von I bezüglich dieser Ordnung. Ferner seien $m_1, \ldots, m_s \in M_n$. Dann berechnet der Algorithmus* **findcombination** *(Algorithmus 9) in höchstens s Schritten durch die Schleife ein $f = \sum_{i=1}^{s} c_i m_i \in I$ mit $c_i \in K$, so dass $f = 0$ genau dann, wenn die Menge $\{m_1, \ldots, m_s\}$ linear unabhängig modulo I ist.*

Beweis.
Auch hier gibt es nicht mehr viel zu beweisen. Die Aussage des Satzes folgt aus den Spezifikationen von **getvectorrep** und **gaussreduce** sowie den Eigenschaften definierender Systeme. □

Auch wenn der Algorithmus explizit als Parameter eine Wohlordnung erhält (diese braucht er, um Normalformen erstellen zu können), hängt das Ergebnis nicht von der Ordnung ab. Man kann sich eine solche aussuchen, für die die Berechnung von Normalformen schnell ist, oder aber eine solche, bezüglich der man schon eine Gröbnerbasis zur Hand hat.

Der Algorithmus ist so formuliert, dass er eine Menge von Monomen versucht linear zu kombinieren. Bei einer genauen Betrachtung stellt man fest, dass an keiner Stelle Eigenschaften von Monomen einfließen. Man kann genau diesen Algorithmus auch dafür verwenden, eine Menge von Polynomen modulo einem Ideal linear zu Null zu kombinieren, falls man dazu Lust verspürt. Da das oben angesprochene praktische Problem mit Monomen zufrieden ist (und da die Implementierung in SINGULAR aus programmiertechnischen Gründen auch nur diesen Fall vorsieht) wurde er hier explizit für Monome formuliert.

In SINGULAR wurde noch eine leicht abgewandelte Version implementiert, nachdem einige Beispiele aus Speicherplatzmangel während der Berechnung einer Normalform gescheitert waren. Hierzu wurde eine modifizierte Version des Algorithmus **nf** erstellt: Zu jedem der m_i führt man einen Permutationsvektor $p_i \in K^s$. Wann immer dann bei der Berechnung der Normalform von m_i in einem Reduktionsschritt das Leitmonom des aktuellen Polynoms mit

Algorithmus 9: findcombination.
Eingabe: *Seien* $m_1, \ldots, m_s \in M_n$*, sei* $F \subset P_n$ *eine Gröbnerbasis bezüglich der Wohlordnung* $<$ *von* $\langle F \rangle = I$.
Ausgabe: *Ein* $f = \sum_{i=1}^{s} c_i m_i \in I$ *mit* $c_i \in K$*, so dass* $f = 0$ *genau dann, wenn die Menge* $\{m_1, \ldots, m_s\}$ *linear unabhängig modulo I ist.*
Algorithmus:

$(f) = \textbf{findcombination}(m_1, \ldots, m_s, F, <)$

```
(d, v_1, ..., v_s) ← getvectorrep(m_1, ..., m_s)
i ← 1 ∈ ℕ
while (i ≤ s)
    (w, p) ← gaussreduce(v_1, ..., v_{i-1}, v_i)
    if (w = 0)
        f ← Σ_{k=1}^{i} p^{(k)} m_k
        i ← s + 1
    else
        i ← i + 1
return (f) •
```

---

einem der m_j übereinstimmt, so wird die Normalform-Berechnung unterbrochen und der Leitterm subtrahiert. Diese Aktion wird durch Subtraktion des Leitkoeffizienten an der Stelle j im Permutationsvektor gespeichert, und die Berechnung wird mit dem so modifizierten Polynom fortgesetzt. Nachdem dann die lineare Abhängigkeit gefunden worden ist, muss man mit Hilfe der Permutationsvektoren diese wieder in die Ausgangssituation zurückübersetzen. Die Implementierung hierzu findet man ab Seite 102.

„Wir wissen fast alles“, sagte der kleine, fette König.
„Wir wachen auf und liegen da und erheben uns und können schreiben
und infinitesimalrechnen und kompjutaprogrammieren und zurarbeitgehen
und geschäftsessen. Alles kein Problem.“
Axel Hacke, Kleiner König Dezember

Auswertung

Eines der Ziele der vorliegenden Arbeit war es, den FGLM-Algorithmus für das Computeralgebrasystem SINGULAR zu implementieren und die praktische Anwendbarkeit der FGLM-Methode zu überprüfen. In diesem Kapitel steht die Auswertung im Mittelpunkt, die den Nutzen des FGLM-Algorithmus untersuchen soll. Sämtliche in dieser Arbeit vorgestellten Algorithmen wurden in SINGULAR implementiert. Zu Beginn wird kurz erklärt, über welche Kommandos diese Befehle zu erreichen sind. Im nächsten Abschnitt wird motiviert, anhand welcher Beispiel die Frage nach dem Nutzen beantwortet werden soll. Die Ergebnisse der verschiedenen Tests werden dann in den verbleibenden Abschnitten diskutiert, wobei auch auf die Algorithmen aus Kapitel 6 eingegangen wird. Die verwendeten Beispiele finden sich ganz am Ende dieses Kapitels.

SINGULAR besitzt zwar eine eigene Interpretersprache, aus Geschwindigkeitsgründen wurde jedoch von Beginn an eine Realisierung in C/C++ im SINGULAR-Kern selbst angestrebt. Ergebnis dieser Bemühungen ist das neue SINGULAR-Kommando `fglm`, das eine Gröbnerbasis von einer Ordnung in eine andere transformiert. Die Syntax lautet

```
ideal = fglm(ring r, ideal i) .
```

Dabei ist `i` eine reduzierte Gröbnerbasis in dem Ring `r`, und als Ergebnis wird eine reduzierte Gröbnerbasis im aktuellen Ring zurückgegeben. Außerdem wurde in der Standardbibliothek `standard.lib` noch der Befehl

```
ideal = stdfglm(ideal i)
```

hinzugefügt. Dieser Befehl berechnet selbständig eine reduzierte Gröbnerbasis bezüglich der grad-revers-lexikografischen Ordnung, um sie dann mittels `fglm` in den aktuellen Ring zu transformieren.

In der zur Zeit aktuellen SINGULAR-Version 1.1.3 sind alle anderen in den vorangegangenen Kapiteln vorgestellten Algorithmen zwar ebenfalls direkt als C/C++-Kernfunktionen implementiert, an der Oberfläche sind sie aber nur über das `system`-Kommando erreichbar. Die Befehle lauten:

- `system("fglmhomog", ring r, ideal i)`:
  Dies ist der homogene FGLM-Algorithmus, wie er in Algorithmus 7 beschrieben ist. Das homogene Ideal `i` muss eine reduzierte Gröbnerbasis in dem Ring `r` sein. Zurückgegeben wird die reduzierte Gröbnerbasis im aktuellen Ring.
- `system("finduni", ideal i)`:
  Ist `i` eine reduzierte Gröbnerbasis im aktuellen Ring, so liefert dieses Kommando für jede Ringvariable ein univariates Polynom minimalen Grades, das in dem Ideal enthalten ist. Es entspricht somit dem Algorithmus **finduni**.
- `system("fastcomb", ideal i, poly p)`:
  Dies ist die Implementierung des Algorithmus **findcombination**. Die Liste der zu kombinierenden Monome $\{m_1, \ldots, m_s\}$ wird hier als Polynom $p = \sum m_i$ aus diesen Monomen

übergeben. Das Ideal `i` muss eine Gröbnerbasis im aktuellen Ring sein. Als Ergebnis erhält man ein Polynom $q = \sum c_i m_i$, so dass q im Ideal liegt.

- `system("comb", ideal i, poly p)`:
  Das Ergebnis dieses Befehls ist gleich dem von `fastcomb`, jedoch wird hier intern ein etwas anderer Weg gegangen (vergleiche auch Seite 62). Diese Version kommt im Allgemeinen mit weniger Speicher aus, ist dafür aber auch etwas langsamer.

Die Testumgebung

Bei der Auswertung stand die Frage im Mittelpunkt, ob der FGLM-Algorithmus eine effektive Alternative zum Buchberger-Algorithmus bei der Berechnung von Gröbnerbasen nulldimensionaler Ideale darstellt. Es wurde somit die Fähigkeit des FGLM-Algorithmus in den Vordergrund gestellt, Gröbnerbasen berechnen zu können. Seine Qualitäten als Transformations-Algorithmus, der Gröbnerbasen in verschiedene Ordnungen umrechnet, wurde hier nicht weiter untersucht, da der FGLM-Algorithmus zur Zeit der einzige Transformations-Algorithmus in SINGULAR ist. Dies wäre zum Beispiel bei einem Vergleich mit dem Gröbner-Walk-Algorithmus sinnvoll gewesen.

Wenn es gilt, zwei verschiedene Algorithmen zu vergleichen, sollte man berücksichtigen, dass die Effizienz eines Algorithmus sehr stark von der konkreten Implementierung abhängt. Ist ein Algorithmus in zwei verschiedenen Systemen realisiert, so können strukturelle Unterschiede zwischen diesen Systemen, wie etwa die Darstellung elementarer Datenstrukturen, die Laufzeiten stark beeinflussen und so das Ergebnis eines Vergleiches verfälschen. Dieser Effekt konnte im vorliegenden Fall vermieden werden, da sowohl der Buchberger- als auch der FGLM-Algorithmus im Kern von SINGULAR realisiert sind und so auf der gleichen internen Infrastruktur aufbauen. Ferner zählt der Buchberger-Algorithmus von SINGULAR zu den schnellsten Implementierungen, die zur Zeit zur Verfügung stehen, so dass gute Voraussetzungen für einen fairen Vergleich der beiden Algorithmen gegeben waren.

Wie schon erwähnt, hängt die Geschwindigkeit des Buchberger-Algorithmus sehr von der Ordnung ab, bezüglich der man die Basis berechnen möchte. Theoretische Untersuchungen haben ergeben, dass im Fall nulldimensionaler Ideale die grad-revers-lexikografische Ordnung die günstigste Wahl ist, wohingegen die lexikografische zu wesentlich aufwendigeren Berechnungen führt, die teilweise nicht mehr zu bewältigen sind. Dies ist auch aus der Praxis bekannt. Es besteht daher wenig Hoffnung, dass die FGLM-Methoden bei der Berechnung einer grad-revers-lexikografischen Ordnung einen Geschwindigkeitsvorteil gegenüber dem Buchberger-Algorithmus erzielen können. In der Praxis hat aber die lexikografische Ordnung eine größere Bedeutung, z.B. bei der Lösung polynomialer Gleichungssysteme, und ist schwieriger zu handhaben. Daher schien es angebracht, die Gröbnerbasis-Berechnung bezüglich dieser Ordnung durchzuführen.

Somit wurde als Testaufgabe folgendes Problem gestellt: Gegeben das Erzeugendensystem eines nulldimensionalen Ideals, berechne daraus die reduzierte Gröbnerbasis bezüglich der lexikografischen Ordnung. Im Falle des FGLM-Algorithmus wird dies bewerkstelligt, indem zunächst eine grad-revers-lexikografische Ordnung berechnet wird, die dann in die lexikografische transformiert werden soll.

Die für die Zeitvergleiche verwendeten Ideale finden sich am Ende dieses Kapitels. Zum Teil stammen sie aus der Originalarbeit über den FGLM-Algorithmus (Faugère et al. [11]) sowie aus dem *Handbook of Polynomial Systems* [4]. Es wurde sowohl über $\mathbb{Q}$ als auch über $\mathbb{Z}/32003\mathbb{Z}$ als Grundkörper gerechnet. Sämtliche in den Tabellen aufgeführten Zeiten sind in Sekunden angegeben. Dabei galt eine Berechnung als nicht durchführbar, wenn sie länger als 90 Minuten dauerte. Häufig terminierte ein Kommando auch aus Speichermangel, beide Fälle sind in den Tabellen mit einem „$\times$“ gekennzeichnet. Gerechnet wurde auf einem Pentium 90 mit 32 MByte Hauptspeicher unter Linux 1.2.13 und der gcc-Version 2.7.2. Entsprechend der Notation in SINGULAR bezeichne im Folgenden `dp` eine grad-revers-lexikografische und `lp` eine lexikografische Ordnung.

In Tabelle 1 sind die Daten der nulldimensionalen Beispiele aufgeführt, die Spalten enthalten der Reihenfolge nach:

- die Anzahl der Variablen
- die Vektorraumdimension
- die Anzahl der Elemente der reduzierten Gröbnerbasis bezüglich `dp` und `lp`.
- den maximalen Monomgrad der Elemente der Basis bezüglich `dp` und `lp`

Ob man über $\mathbb{Q}$ oder über $\mathbb{Z}/32003\mathbb{Z}$ rechnet, macht in den Daten bei diesen Beispielen keinen Unterschied, daher wird keine getrennte Tabelle für die beiden Fälle benötigt.

Beispiel	vars	vdim	$\lvert F_{dp} \rvert$	$\lvert F_{lp} \rvert$	grad_{dp}	grad_{lp}
reco1	2	63	4	3	11	39
cyclic7mod1	3	20	15	3	4	20
caprasse	4	56	26	7	7	20
cyclic5	5	70	20	11	8	15
isaac1	4	16	12	4	5	16
niermann1	3	54	16	3	9	54
cyclic5mod1	5	64	25	8	7	61
cyclic6	6	156	45	17	9	48
walk-s7	7	128	64	8	8	127
cassou-mod1	4	44	18	4	4	22
omdi	9	16	20	9	3	8
lamm1	2	56	7	2	12	56
dessin1	8	46	39	8	5	46
dessin2	10	42	45	10	4	42

Tabelle 1: Die Daten der verwendeten nulldimensionalen Beispiele.

Vergleich GB – Singular

Um ein Maß für die Qualität der FGLM-Implementierung in SINGULAR zu erhalten, sollte diese mit anderen Systemen verglichen werden. Hierzu wurde das Computeralgebrasystem GB (Faugère [10]) hinzugezogen. GB gilt als ein sehr effizientes System und besitzt wie auch SINGULAR eine äußerst schnelle Version des Buchberger-Algorithmus. Es wurde Version 3.0.765 verwendet.

Der FGLM-Algorithmus existiert in GB nur in einer spezialisierten Version, die lediglich eine Gröbnerbasistransformation von der Ordnung `dp` in die Ordnung `lp` erlaubt, andere Ordnungen werden nicht unterstützt. Unter diesen Voraussetzungen ergeben sich einige Vereinfachungen im Algorithmus. Außerdem weist die Architektur von GB einen grundlegenden Unterschied im Vergleich zu SINGULAR auf: Die Algorithmen existieren in GB in mehreren Ausführungen und sind für jede Ordnung und jeden Grundkörper einzeln implementiert. SINGULAR dagegen besteht aus universell geschriebenen Algorithmen, die keine speziellen Annahmen z.B. über die Ordnung machen.

Eine FGLM-Implementierung von M. Niermann für das System REDUCE wurde nicht zu einem Vergleich herangezogen, da sie in der Interpretersprache von REDUCE geschrieben ist und somit ein Vergleich mit in C/C++ geschriebenen Implementierungen nicht fair und wenig aussagekräftig wäre.

Tabelle 2 zeigt die Ergebnisse des Vergleiches zwischen GB und SINGULAR. Neben der Gesamtzeit, die die beiden Systeme für die Berechnung der lexikografischen Gröbnerbasis über $\mathbb{Q}$ benötigt haben, ist getrennt aufgeführt, wie diese sich aus den beiden Schritten „Berechnung der grad-revers-lexikografischen Basis“ und „Umwandlung in eine lexikografische Basis“ zusammensetzt. Da im ersten Schritt sowohl in SINGULAR als auch in GB der Buchberger-Algorithmus verwendet wird, hat man somit einen Bezugspunkt, an dem man sich bei einem Geschwindigkeitsvergleich der FGLM-Implementierungen orientieren kann.

	std$_{dp}$		fglm		std$_{dp}$+fglm	
Beispiel	SING.	GB	SING.	GB	SING.	GB
reco1	0	0,1	0,5	2,5	0,5	2,6
cyclic7mod1	0,1	0,3	0,6	1,6	0,7	1,9
caprasse	0,1	0,2	0,5	0,8	0,6	1,0
cyclic5	0,2	0,4	0,3	0,6	0,5	1,0
isaac1	0,6	0,8	5,7	33,5	6,3	34
niermann1	0,1	0,2	14,3	330	14,4	330
cyclic5mod1	4,2	4,3	41	795	45	800
cyclic6	16,4	11,5	13,9	19,5	30	31
walk-s7	0,5	0,8	6,1	18	6,6	18,8
cassou-mod1	132	137	71	674	203	811
omdi	3,6	6,9	0,7	0,4	4,3	7,3
lamm1	0,2	0,2	238	×	238	×
dessin1	18, 1	30	1529	×	1547	×
dessin2	29	76	778	×	807	×

Tabelle 2: Der FGLM-Algorithmus über $\mathbb{Q}$ in SINGULAR und GB.

Die Tabelle zeigt, dass sich die Ausführungszeiten der beiden Systeme im Buchberger-Algorithmus kaum unterscheiden. Die Zeiten für die Durchführung des FGLM-Algorithmus weisen jedoch teilweise signifikante Unterschiede zu Gunsten von SINGULAR auf; bei einigen Beispielen schafft GB die Umwandlung der Gröbnerbasis nicht.

Leider standen keine Details über die Implementierung des FGLM-Algorithmus in GB zur Verfügung, so dass an dieser Stelle keine Ursachen für diese Zeitdifferenzen ausgemacht werden können. Man kann lediglich feststellen, dass trotz der Spezialisierungen, die in GB vorgenommen wurden, die Implementierung des FGLM-Algorithmus in SINGULAR schneller ist als die in GB.

Vergleich FGLM – Buchberger

Als nächstes galt es, mit den verschiedenen in SINGULAR zur Verfügung stehenden Methoden, eine lexikografische Gröbnerbasis zu berechnen, um zu untersuchen, ob die FGLM-Methoden eine Alternative zum Buchberger-Algorithmus darstellen. Es kamen in Frage:

- `std` : Der gewöhnliche Buchberger-Algorithmus wird in SINGULAR über den Befehl `std` aufgerufen.

- `stdhilbert` : Seit Version 1.1.3 existiert das Kommando `stdhilbert` zur Gröbnerbasisberechnung. Dieses Kommando homogenisiert zunächst das Ideal bezüglich einer neuen Variablen, berechnet die Gröbnerbasis bezüglich der Ordnung `dp`, hiervon die Hilbertfunktion, dann die Gröbnerbasis in der gewünschten Ordnung und dehomogenisiert zu guter Letzt das Ergebnis zu einem Ideal im aktuellen Ring. Erstaunlicherweise ist dieser Weg trotz der Homogenisierung und der dafür notwendigen zusätzlichen Unbestimmten oftmals äußerst effizient. Bei einem Zeitvergleich mit dem FGLM-Algorithmus muss man jedoch berücksichtigen, dass `fglm` in jedem Fall eine reduzierte Gröbnerbasis berechnet, wohingegen die von `stdhilbert` berechnete im Allgemeinen nicht reduziert ist. Erzwingt man hier die Reduzierung, so geht der Geschwindigkeitsvorteil oftmals wieder verloren. Die Ausführungszeiten sind also immer vor diesem Hintergrund zu vergleichen.
- `stdfglm` : Wie oben erwähnt berechnet `stdfglm` zunächst eine Gröbnerbasis bezüglich der Ordnung `dp`, um sie dann in die Ordnung `lp` zu transformieren. Die in der folgenden Tabelle angegebenen Werte enthalten also auch die Zeit, die man zur Berechnung der Gröbnerbasis in `dp` benötigt.

Tabelle 3 (Seite 70) zeigt die Gesamtzeiten, die die verschiedenen Verfahren benötigten, um eine lexikografische Gröbnerbasis zu berechnen. Der Vollständigkeit halber sind hier auch noch einmal die Zeiten von GB aufgeführt.

Neben den Absolutzeiten ist noch ein relativer Geschwindigkeitsfaktor $f = t_r/t$ angegeben, wobei als Referenzzeit die Berechnungsdauer $t_r = t_{stdfglm}$ von `stdfglm` verwendet wird. Ist $f > 1$, so ist dieses Verfahren schneller als `stdfglm`, ist $f < 1$, so ist es langsamer. Die Faktoren stehen in der Tabelle als kleine Indizes an den Zeitwerten.

Alle Beispiele wurden für einen SINGULAR-internen Vergleich auch über $\mathbb{Z}/32003\mathbb{Z}$ gerechnet, die Ergebnisse zeigt Tabelle 4. Es sind lediglich die Beispiele aufgeführt, bei denen die Berechnung eine signifikant lange Zeit in Anspruch nimmt. Für den FGLM-Algorithmus sind die Zeiten für die Berechnung der Basis in `dp` und die Transformation nach `lp` auch noch einmal getrennt aufgeführt.

Beispiel	std_{dp}	fglm	stdfglm	lp	stdhilbert*
niermann1	0,1	0,3	0,4	9,7	1,2
cyclic5mod1	0,3	0,6	0,9	×	4,5
cyclic6	0,1	2,4	2,5	48	5,6
walk-s7	0,3	0,4	0,7	95	4,4
cassou-mod1	0,5	0,1	0,6	10	3,7
omdi	0,9	0	0,9	814	1050
dessin1	1,2	0,4	1,6	×	58
dessin2	2,0	0,4	2,4	3,6	532

* berechnet keine reduzierte Gröbnerbasis

Tabelle 4: Berechnung einer lexikografischen Basis über $\mathbb{Z}/32003\mathbb{Z}$ in SINGULAR.

Es zeigt sich, dass der FGLM-Algorithmus über $\mathbb{Q}$ in vielen der Beispiele, in denen der Buchberger-Algorithmus nicht zum Erfolg führt, die Berechnung einer reduzierten Gröbnerbasis in kurzer Zeit ermöglicht. Das Ideal „omdi“ zeigt dieses Verhalten am deutlichsten: Die Berechnung mit dem Buchberger-Algorithmus ist nicht durchführbar, aber mittels FGLM lässt sie

Beispiel	stdfglm	std_{lp}	stdhilbert*	GB
reco1	0,5	$0{,}1_{5{,}0}$	$0{,}1_{5{,}0}$	$2{,}6_{0{,}2}$
cyclic7mod1	0,7	$1{,}6_{0{,}4}$	$0{,}7_{1}$	$1{,}9_{0{,}4}$
caprasse	0,6	$0{,}4_{1{,}5}$	$0{,}4_{1{,}5}$	$1{,}0_{0{,}6}$
cyclic5	0,5	$0{,}9_{0{,}6}$	$0{,}3_{1{,}6}$	$1{,}0_{0{,}5}$
isaac1	6,3	$132_{<0{,}1}$	$1{,}7_{3{,}7}$	$34_{0{,}2}$
niermann1	14,4	$\times$	$6{,}2_{2{,}3}$	$330_{0{,}1}$
cyclic5mod1	45	$\times$	$23_{1{,}9}$	$800_{0{,}1}$
cyclic6	30	$\times$	$37_{0{,}8}$	$31_{1{,}0}$
walk-s7	6,6	$\times$	$4{,}6_{1{,}4}$	$18{,}8_{0{,}4}$
cassou-mod1	203	$\times$	$1090_{0{,}2}$	$811_{0{,}3}$
omdi	4,3	$\times$	$\times$	$7{,}3_{0{,}6}$
lamm1	238	$\times$	$33_{7{,}2}$	$\times$
dessin1	1550	$\times$	$69300_{<0{,}1}$	$\times$
dessin2	807	$\times$	$\times$	$\times$

* berechnet keine reduzierte Gröbnerbasis

Tabelle 3: Berechnung einer lexikografischen Basis über $\mathbb{Q}$ in SINGULAR und GB.

sich in einigen Sekunden erledigen. An diesem Beispiel scheitert sogar der Hilbert-getriebene Buchberger-Algorithmus, der ansonsten aber meist um einiges schneller als FGLM ist.

Wie ein Vergleich der Tabellen 3 und 4 zeigt, kann `stdhilbert` seine Geschwindigkeitsvorteile nur über $\mathbb{Q}$ ausspielen. Rechnet man in endlicher Charakteristik, so ist der FGLM-Algorithmus in den durchgeführten Beispielen schneller. Der Hilbert-getriebene Buchberger-Algorithmus beruht darauf, dass er mit Hilfe der Hilbertfunktion entscheiden kann, die Normalform eines S-Polynoms nicht berechnen zu müssen, sondern es einfach streichen zu können. Da die Normalformen-Berechnung in Charakteristik 0 teurer ist, wird dort durch das Vermeiden einer Berechnung der Geschwindigkeitsgewinn deutlicher.

Dieses Ergebnis soll nicht den Eindruck erwecken, dass der Buchberger-Algorithmus für nulldimensionale Ideale ausgedient hat und durch den FGLM-Algorithmus zu ersetzen ist. Es gibt genauso auch Beispiele, die der FGLM-Algorithmus nicht bewältigen kann, der Buchberger-Algorithmus aber sehrwohl meistert. Ein entscheidender Geschwindigkeitsfaktor ist die Vektorraumdimension des Ideals, die die Größe der Multiplikationsmatrizen und der Vektoren bestimmt. Wird diese zu groß, so kommt es zunächst zu Speicherplatzproblemen, die sich durch die Anzahl der Variablen, die ja die Anzahl der Matrizen bestimmt, noch verstärken. Diesem Effekt wurde in der Implementierung unter anderem dadurch versucht entgegenzuwirken, dass die Matrizen sich gemeinsame Spalten untereinander teilen und nur Einträge speichern, die von Null verschieden sind. Einzelheiten hierzu findet man auf den Seiten 48 und 89. Hat man das Speicherproblem entschärft, so bleibt der große Rechenaufwand bei Matrix/Vektor-Multiplikationen und bei der Gauss-Reduktion. Somit sind dem FGLM-Algorithmus allein durch die Vektorraumdimension Grenzen gesetzt.

Ziel dieser Zeitmessungen war es vielmehr zu überprüfen, ob der FGLM-Algorithmus als *Alternative* angesehen werden kann, die in einigen (und nicht in allen) Fällen einen Geschwindigkeitsvorteil gegenüber dem Buchberger-Algorithmus bringt bzw. eine Berechnung überhaupt erst ermöglicht. Im Hinblick auf die beiden ersten Spalten der Tabelle kann man die Frage positiv beantworten: Zumindest wenn es um die Berechnung der reduzierten Gröbnerbasis geht, stellt der FGLM-Algorithmus eine oft sehr effektive Alternative dar.

Die Verallgemeinerungen

Leider setzt der Algorithmus für homogene Ideale nicht die vom FGLM-Algorithmus gezeigten guten Leistungen fort. Er erweist sich in der Praxis oft als zu aufwendig und scheitert vor allem aus Speicherplatzmangel. Homogenisiert man etwa die oben benutzten nulldimensionalen Ideale, so lässt sich für keines der Beispiele in annehmbarer Zeit eine lexikografische Gröbnerbasis berechnen. Der Algorithmus für homogene Ideale konnte so nicht ganz die in ihn gesetzten Erwartungen erfüllen und stellt in vielen Fällen keine Alternative zum Buchberger-Algorithmus dar. Es gibt aber immerhin auch Beispiele, in denen er schneller ist.

Tabelle 5 zeigt einen Zeitvergleich mit **`std`** und **`stdhilbert`** bei der Berechnung einer lexikografischen Gröbnerbasis über $\mathbb{Q}$. Auch hier ist wieder getrennt aufgeführt, wie sich die Zeiten für den homogenen FGLM-Algorithmus unter den beiden Schritten aufteilt.

Beispiel	std_{dp}	$\text{fglm}_{hom.}$	$\text{std+fglm}_{hom.}$	lp	stdhilbert*
testh4	19,2	6,6	26	41	40
symmetric4-7	65	107	172	76	76
cyclic5-hom	0	×	×	0,8	0,8

* berechnet keine reduzierte Gröbnerbasis

Tabelle 5: Berechnung einer homogenen lexikografischen Basis über $\mathbb{Q}$ in SINGULAR.

Der Algorithmus ist so konstruiert, dass er mit Hilfe der Hilbertfunktion vorhersagen kann, welchen Grad das nächste Basispolynom haben wird. Daraufhin beginnt er, unter allen Monomen diesen Grades lineare Abhängigkeiten zu überprüfen und auf diese Weise das Basispolynom zu finden. Für jedes Monom muss dabei die Normalform berechnet werden. Wenn der zu betrachtende Grad steigt, wird die Anzahl der Monome schnell riesig groß. Dieser Effekt wird umso stärker, je mehr Variablen der Polynomring besitzt.

Dieses Verhalten resultiert zunächst in enormen Speicherplatzverbrauch, den man jedoch durch geschicktes Abspeichern der relevanten Daten in den Griff bekommen kann. Für jedes Monom muss aber zwangsläufig die Normalform berechnet werden, und dies verbraucht sehr viel Zeit. Eine genauere Untersuchung an einer Reihe von Beispielen hat ergeben, dass der Algorithmus die meiste Zeit mit der Berechnung der Normalformen und dem Erstellen der Vektoren beschäftigt ist. Die eigentliche Arbeit, nämlich das Auffinden der linearen Abhängigkeiten, nimmt dagegen nur einen kleinen Teil der Zeit in Anspruch.

So muss viel Aufwand getrieben werden, von dem im ungünstigen Fall ein Großteil überflüssig für die Bestimmung des Basispolynoms ist. Es lässt sich aber vorher nicht sagen, welches dieser Teil ist. Wenn die Gröbnerbasis so beschaffen ist, dass die Basispolynome über viele verschiedene Grade verteilt sind, wird dieser nachteilige Effekt besonders deutlich.

Es besteht die Möglichkeit, die Normalformen der Monome nicht in einem Rutsch zu berechnen, sondern immer nur dann, wenn ein neues Monom zum Kombinieren benötigt wird. Dieser Ansatz schien hier aber nicht praktikabel: Zum einen kann sich so in jedem Schritt die Vektorraumdimension ändern; zum anderen ist es in dieser Situation in SINGULAR erforderlich, zur Normalform-Berechnung einen Ringwechsel durchzuführen. Der administrative Aufwand, der damit verbunden ist, lässt wenig Hoffnung auf eine Geschwindigkeitssteigerung.

Zum Abschluss soll noch der Algorithmus **finduni** (Algorithmus 8) an einem Beispiel vorgeführt werden. Die Aufgabe bestand darin, die univariaten Polynome eines nulldimensionalen Ideals $I \subseteq \mathbb{Q}[x_1,\ldots,x_n]$ zu bestimmen. Wie in Kapitel 6 erwähnt, gelingt dies auch, indem man n lexikografische Gröbnerbasen zu den Ordnungen $<_i$ berechnet, die jeweils eines der univariaten Polynome enthalten. Hierfür ist es nicht notwendig, dass die Basis reduziert ist, so dass **stdhilbert** verwendet werden konnte. Eine Berechnung mit Hilfe von **std** wäre in den meisten der hier verwendeten Beispielen nicht möglich gewesen, wie ein Blick auf Tabelle 3 verdeutlicht.

Führt man die Berechnung z.B. an dem Ideal „cyclic6“ durch, so benötigt **finduni** hierfür 33 Sekunden, wohingegen die Berechnung mittels der Eliminationsmethode 186 Sekunden in Anspruch nimmt. Aber auch hier gilt wieder, dass das Verhältnis in anderen Beispielen genau umgekehrt sein kann.

Zusammenfassung

Aus all diesen Ergebnissen kann man schließen: Der FGLM-Algorithmus zieht seine Geschwindigkeitsvorteile größtenteils aus der Tatsache, dass er es durch theoretische Überlegungen erreicht, Normalformen zu bestimmen, ohne dafür den Algorithmus **nf** bzw. **lnf** verwenden zu müssen. Sobald die Voraussetzungen nicht mehr so geartet sind, dass man die Normalform theoretisch vorhersagen kann, geht den FGLM-Methoden einiges an Geschwindigkeit verloren. Unter diesem Aspekt erweist sich Lemma 24 als ein Schlüssel zum Erfolg.

Insgesamt konnte die von Faugère et al. [11] gestellte Behauptung, dass der FGLM-Algorithmus eine effektive Alternative zum Buchberger-Algorithmus darstellt, hier bestätigt werden. Mit der Implementierung steht nun in SINGULAR ein neues Werkzeug zur Verfügung, das bei der Gröbnerbasen-Berechnung nulldimensionaler Ideale zur Auswahl steht.

Die Beispiele

- **reco1:**

  Variablen: $x > y$

  $I = \langle\, \texttt{jacob}(x^5 + y^{11} + xy^9 + x^3y^9)\, \rangle$

- **cyclic7mod1:**

  Variablen: $a > b > c$

  $$I = \langle\, a^2bc + ab^2c + abc^2 + abc + ab + ac + bc,$$
  $$a^2b^2c + ab^2c^2 + a^2bc + abc + bc + a + c,$$
  $$a^2b^2c^2 + a^2b^2c + ab^2c + abc + ac + c + 1\,\rangle$$

- **caprasse:**

  Variablen: $x > y > z > t$

  $$I = \langle\, y^2z + 2xyt - 2x - z,\ 2yzt + xt^2 - x - 2z,$$
  $$-x^3z + 4xy^2z + 4x^2yt + 2y^3t + 4x^2 - 10y^2 + 4xz - 10yt + 2,$$
  $$-xz^3 + 4yz^2t + 4xzt^2 + 2yt^3 + 4xz + 4z^2 - 10yt - 10t^2 + 2\,\rangle$$

- **cyclic5:**

  Variablen: $a > b > c > d > e$

  $$I = \langle\, a + b + c + d + e,\ ab + bc + cd + ae + de,\ abc + bcd + abe + ade + cde,$$
  $$abcd + abce + abde + acde + bcde,\ abcde - 1\,\rangle$$

- **isaac1:**

  Variablen: $w > x > y > z$

  $$I = \langle\, 8w^2 + 5wx - 4wy + 2wz + 3w + 5x^2 + 2xy - 7xz - 7x + 7y^2$$
  $$-8yz - 7y + 7z^2 - 8z + 8,$$
  $$3w^2 - 5wx - 3wy - 6wz + 9w + 4x^2 + 2xy - 2xz + 7x + 9y^2$$
  $$+6yz + 5y + 7z^2 + 7z + 5,$$
  $$-2w^2 + 9wx + 9wy - 7wz - 4w + 8x^2 + 9xy - 3xz + 8x + 6y^2$$
  $$-7yz + 4y - 6z^2 + 8z + 2,$$
  $$7w^2 + 5wx + 3wy - 5wz - 5w + 2x^2 + 9xy - 7xz + 4x - 4y^2$$
  $$-5yz + 6y - 4z^2 - 9z + 2\,\rangle$$

- **niermann1:**

  Variablen: $x > y > z$

  $$I = \langle\, x^2 + xy^2z - 2xy + y^4 + y^2 + z^2,\ -x^3y^2 + xy^2z + xyz^3 - 2xy + y^4,$$
  $$-2x^2y + xy^4 + yz^4 - 3\,\rangle$$

- **cyclic5mod1:**

  Variablen: $a > b > c > d > e$

  $$I = \langle\, a + b + c + d + e,\ ab + bc + cd + ae + de,\ abc + bcd + abe + ade + cde,$$
  $$bcd + abce + abde + acde + bcde,\ abcde - 1\,\rangle$$

- **cyclic6:**

  Variablen: $a > b > c > d > e > f$

  $$\begin{aligned} I = \langle\, & a+b+c+d+e+f,\ ab+bc+cd+de+ef+af, \\ & abc+bcd+cde+def+aef+abf, \\ & abcd+bcde+cdef+adef+abef+abcf, \\ & abcde+bcdef+acdef+abdef+abcef+abcdf,\ abcdef-1\,\rangle \end{aligned}$$

- **walk-s7:**

  Variablen: $a > b > c > d > e > f > g$

  $$\begin{aligned} I = \langle\, & 2gb+2fc+2ed+a^2+a,\ 2gc+2fd+e^2+2ba+b, \\ & 2gd+2fe+2ca+c+b^2,\ 2ge+f^2+2da+d+2cb, \\ & 2gf+2ea+e+2db+c^2, \\ & g^2+2fa+f+2eb+2dc,\ 2ga+g+2fb+2ec+d^2\,\rangle \end{aligned}$$

- **cassou-mod1:**

  Variablen: $a > b > c > d$

  $$\begin{aligned} I = \langle\, & 15a^4bc^2+6a^4b^3+21a^4b^2c-144a^2b-8a^2b^2d-28a^2bcd-648a^2c \\ & \quad +36c^2d+9a^4c^3-120, \\ & 30b^3a^4c-32cd^2b-720ca^2b-24b^3a^2d-432b^2a^2+576db-576cd \\ & \quad +16ba^2c^2d+16c^2d^2+16d^2b^2+9b^4a^4+5184+39c^2a^4b^2+18c^3a^4b \\ & \quad -432c^2a^2+24c^3a^2d-16b^2a^2cd-240b, \\ & 216ca^2b-162c^2a^2-81b^2a^2+5184+1008db-1008cd+15b^2a^2cd \\ & \quad -15b^3a^2d-80cd^2b+40c^2d^2+40d^2b^2, \\ & 261+4ca^2b-3c^2a^2-4b^2a^2+22db-22cd\,\rangle \end{aligned}$$

- **omdi:**

  Variablen: $a > b > c > u > v > w > x > y > z$

  $$\begin{aligned} I = \langle\, & a+c+v+2x-1,\ ab+cu+2vw+2xy+2xz-2/3, \\ & ab^2+cu^2+2vw^2+2xy^2+2xz^2-2/5, \\ & ab^3+cu^3+2vw^3+2xy^3+2xz^3-2/7, \\ & ab^4+cu^4+2vw^4+2xy^4+2xz^4-2/9, \\ & vw^2+2xyz-1/9,\ vw^4+2xy^2z^2-1/25, \\ & vw^3+xyz^2+xy^2z-1/15,\ vw^4+xyz^3+xy^3z-1/21\,\rangle \end{aligned}$$

- **lamm1:**

  Variablen: $x > y$

  $$\begin{aligned} I = \langle\, & 45x^8+3x^7+39x^6+30x^5+13x^4+41x^3+5x^2+46x+7, \\ & 49x^7y+35xy^7+37xy^6+9y^7+4x^6+6y^6+27x^3y^2+20xy^4 \\ & \quad +31x^4+33x^2y+24x^2+49y+43\,\rangle \end{aligned}$$

- **dessin1:**

  Variablen: $a > b > u > v > w > x > y > z$

  $$\begin{aligned} I = \langle\, & 6zab + 10vax + 8yau - 162a^2u + 16uw + 14xb + 48aw, \\ & 15zau - 162a^2v - 312ab + 24aw + 27xu + 24yb + 18vay + 30vw + 84xa, \\ & -240a + 420z - 64v + 112y, \\ & 180za - 284va - 162a^2 + 60vy + 50ya + 70w + 55zu + 260x - 112b, \\ & 66za + 336y + 90x + 78vz - 1056a - 90u,\ 136z - 136, \\ & 4vaw + 2yab + 6bw - 162a^2b + 3xua, \\ & 28vaz + 192w + 128ya + 36xb + 36zb - 300au + 40yu - 648a^2 + 44vx \,\rangle \end{aligned}$$

- **dessin2:**

  Variablen: $a > b > c > d > u > v > w > x > y > z$

  $$\begin{aligned} I = \langle\, & 16aw + 18bv + 20cu,\ -80d + 180y + 855z,\ 7av + 8bu,\ 210z - 210, \\ & 40ay + 44bx + 48cw + 52dv + 280u,\ 27ax + 30bw + 33cv + 36du, \\ & 55az + 60by + 65cx + 70dw + 80u + 375v, \\ & 78bz + 84cy + 90dx - 170a + 102v + 480w,\ 136dz - 114c + 152x + 720y, \\ & 105cz + 112dy - 144b + 126w + 595x \,\rangle \end{aligned}$$

- **testh4:**

  Variablen: $u > v > w > x > y > z$

  $$\begin{aligned} I = \langle\, & xy + y^2 + uz - wz - xz + yz,\ x^2 - y^2 + uz - wz - xz + z^2,\ wy,\ wx - uz + yz, \\ & w^2 - y^2 + uz - z^2,\ vz - yz - z^2,\ vy - y^2 - wz + xz + yz + z^2, \\ & vx - y^2 + uz + yz - z^2,\ vw - y^2 + uz + yz - z^2,\ v^2 + y^2 + uz - xz - yz - z^2, \\ & uy + y^2 + yz,\ ux - y^2 + wz - xz - z^2,\ uw - y^2 + uz + yz - z^2, \\ & uv - y^2 - xz + yz,\ u^2 + yz \,\rangle^2 \end{aligned}$$

- **symmetric4-7:**

  Variablen: $a > b > c > d$

  $$I = \langle\, a^7 + b^7,\ b^7 + c^7,\ c^7 + d^7,\ a^6b + b^6c + c^6d + d^6a \,\rangle$$

- **cyclic5-hom:**

  Variablen: $a > b > c > d > e > t$

  $$\begin{aligned} I = \langle\, & a + b + c + d + e,\ ab + ae + bc + cd + de,\ \ abc + abe + ade + bcd + cde, \\ & abcd + abce + abde + acde + bcde,\ abcde - t^5 \,\rangle \end{aligned}$$

„Stil ist, was gefällt."
Zeitschrift Brigitte

Zur Implementierung

Ein Ziel der vorliegenden Arbeit bestand darin, den FGLM-Algorithmus sowie die in Kapitel 6 vorgestellten Erweiterungen für das Computeralgebrasystem SINGULAR zu implementieren. Es sollten effiziente Algorithmen daraus resultieren, da insbesondere der FGLM-Algorithmus als mögliche Alternative zum Buchberger-Algorithmus bereitgestellt werden sollte. Aus diesem Grund erfolgte die Implementierung als Kern-Funktionen in C/C++, und das Ergebnis dieser Bemühungen ist in Form des Quelltextes auf den folgenden Seiten zu finden.

Zur besseren Verständlichkeit ist der Programmtext durchgehend mit Kommentaren versehen, die die verschiedenen Klassen und Methoden erklären und Details zu deren Implementierung hervorheben. Die Bemerkungen sind nicht Teil des Quelltextes; sie sind typografisch von diesem abgehoben.

Die Algorithmen verteilen sich wie folgt über eine Reihe von Dateien:

- `fglmvec.h` / `fglmvec.cc` (Seite 78 bis 83)
  Eine Basisklasse, die von sämtlichen Algorithmen verwendet wird, ist die eines *Vektors* aus einem Vektorraum über dem dem Polynomring zugrundeliegenden Körper, in SINGULAR also ein Vektor mit `number`-Elementen. Dieser `fglmVector` wird hier zur Verfügung gestellt.
- `fglmgauss.h` / `fglmgauss.cc` (Seite 83 bis 85)
  In diesen Dateien ist die Klasse `gaussreducer` definiert, die verwendet wird, um Vektoren vom Typ `fglmVector` auf lineare Abhängigkeit zu testen. Sie ist auf den hier passenden Fall zugeschnitten, dass die Menge der linear unabhängigen Vektoren sukzessive erweitert wird.
- `fglm.h` / `fglm.cc` (Seite 85 bis 89)
  Hier sind die Verteilungsroutinen enthalten, die vom Interpreter aufgerufen werden. Sie führen Vorarbeiten und Konsistenz-Prüfungen durch, um schließlich den Algorithmus für nulldimensionale Ideale aufzurufen.
- `fglmzero.cc` (Seite 89 bis 98)
  Hier findet sich die Implementierung des FGLM-Algorithmus für nulldimensionale Ideale. Wie auch schon in Kapitel 5 ist der Algorithmus aufgeteilt in die Berechnung der Multiplikationsmatrizen und die Transformation einer Gröbnerbasis in eine neue Ordnung. Diese Aufteilung erwies sich als sehr nützlich, als der Algorithmus `finduni` implementiert werden sollte, da dieser auch mit Multiplikationsmatrizen arbeitet. Er ist ebenfalls in dieser Datei ausformuliert.
- `fglmhom.cc` (Seite 98 bis 102)
  Hier steht die Erweiterung des FGLM-Algorithmus auf homogene Ideale.
- `fglmcomb.cc` (Seite 102 bis 106)
  Die Ausformulierung des Algorithmus `findcombination` erfolgte, wie schon angedeutet, in zwei Varianten. Eine, die die von SINGULAR bereitgestellte Normalformen-Berechnung verwendet, sowie eine, die eine modifizierte Berechnung benutzt, die zwar eventuell ein wenig langsamer sein kann, dafür aber einen moderateren Speicherplatzverbrauch aufweist.

fglmvec.h

Ein zentrales Datenobjekt aller im Rahmen dieser Arbeit implementierter Algorithmen ist die Klasse `fglmVector`, die einen Vektor mit Elementen aus dem Grundkörper bereitstellt. Zwar gibt es in Singular bereits einen Vektor-Datentyp, dieser jedoch ist für Polynomeinträge vorgesehen. Nun sind die Körperelemente natürlich auch Polynome, aus Effizienzgründen wurde jedoch eine speziell ausgerichtete Implementierung bevorzugt. Dabei wurde ein sogenanntes *letter-envelope Modell* verwendet. Das heißt, dass die Vektor-Klasse selbst in zwei Bestandteile aufgeteilt ist. Zum einen in die *envelope-Klasse* `fglmVector`, die die Programmierschnittstelle bereitstellt. Der gesamte Zugriff auf den Vektor geschieht über Methoden, die von der konkreten Realisierung unabhängig sind. Insbesondere ist auch der elementweise Zugriff über Methoden geregelt. Die Klasse `fglmVectorRep` stellt die zugehörige *letter-Klasse* dar: Hier ist die konkrete Realisierung der Klasse „Vektor" verborgen. Durch diese Trennung bleiben Implementierungsdetails unsichtbar und austauschbar. Die envelope-Klasse greift über einen Zeiger auf die zugehörige letter-Klasse zu. Einhergehend mit diesem Konzept wurde ein Referenzmodell verwirklicht: Zusätzlich zu den Daten enthält die Klasse `fglmVectorRep` noch einen Referenzzähler, der die Anzahl der Objekte zählt, die dieses Repräsentationsobjekt referenzieren.

Aus dieser Trennung der Daten vom eigentlichen Objekt ergibt sich zwar eine zusätzliche Dereferenzierungsstufe beim Zugriff auf die Vektorelemente, dafür wird aber die Zuweisung eines Vektors extrem einfach: Statt alle Elemente kopieren zu müssen, wird lediglich der Zeiger auf die Envelope-Klasse kopiert und der Referenzzähler erhöht. Neben der Zeit- hat man so auch eine erhebliche Speicherplatzersparnis erzielt. Die Größe eines Vektors wird bei der Initialisierung angegeben, sie kann sich aber etwa durch eine Zuweisung auch durchaus während der Laufzeit wieder ändern, und die Arithmetik ist nicht darauf angewiesen, dass beide Operanden die gleiche Größe haben.

Hier zunächst die Definition der Klasse. Bemerkungen zu den einzelnen Methoden finden sich auch bei der Implementierung.

```
class fglmVector
{
protected:
    fglmVectorRep * rep;
    void makeUnique();
    fglmVector( fglmVectorRep * rep );
public:
    fglmVector();
    fglmVector( int size );
    fglmVector( int size, int basis );
    fglmVector( const fglmVector & v );
    ~fglmVector();
    int size() const;
    int numNonZeroElems() const;
```

Die Methode `nihilate` ist für die Gaussreduktion bestimmt: Sie multipliziert zwei Vektoren mit Skalaren, um sie anschliessend voneinander zu subtrahieren. Sie wird so aufgerufen, dass sich im Ergebnis eine bestimmte Komponente zu Null summiert hat.

```
    void nihilate( const number fac1, const number fac2, const fglmVector v );
    fglmVector & operator = ( const fglmVector & v );

    int operator == ( const fglmVector & );
    int operator != ( const fglmVector & );
    int isZero();
    int elemIsZero( int i );
    fglmVector & operator += ( const fglmVector & );
    fglmVector & operator -= ( const fglmVector & );
    fglmVector & operator *= ( const number & );
    fglmVector & operator /= ( const number & );
    friend fglmVector operator - ( const fglmVector & v );
    friend fglmVector operator + ( const fglmVector & lhs,
                                   const fglmVector & rhs );
    friend fglmVector operator - ( const fglmVector & lhs,
                                   const fglmVector & rhs );
    friend fglmVector operator * ( const fglmVector & v, const number n );
    friend fglmVector operator * ( const number n, const fglmVector & v );

    const number getconstelem( int i ) const;
    number & getelem( int i );
    void setelem( int i, number & n );
```

Die Methode `gcd` liefert den ggT aller Vektorelemente zurück, und `clearDenom` multipliziert alle Elemente mit dem kgV ihrer Nenner und liefert diesen zurück. Solche Methoden werden benötigt, da in allen Routinen versucht wird, die Koeffizienten nach Möglichkeit immer ganzzahlig zu halten.

```
    number gcd() const;
    number clearDenom();
};
```

fglmvec.cc

In der Klasse `fglmVectorRep` wurde ein Referenzmodell verwirklicht. Ein Element der Klasse `fglmVector` enthält lediglich einen Zeiger auf diese Referenzklasse, die die Daten enthält. Wird ein Vektor zugeweisen, so wird nur dieser Zeiger kopiert und in der Referenzklasse vermerkt, dass ein neuer Vektor diese Daten ebenfalls verwendet. Dies geschieht durch die Methode `copyObject`. Erst wenn die Daten eines Vektors verändert werden, die gleichzeitig von anderen Vektoren genutzt werden, muss man die Einträge tatsächlich kopieren (copy-on-write). Dies erledigt die Methode `clone`.

Wird ein Vektor gelöscht, so wird der Referenzzähler in der Repräsentationsklasse erniedrigt, und erst wenn dieser Zähler auf Null gelangt ist, wird auch die Repräsentation gelöscht. Diese Klasse wird nur intern verwendet.

```
class fglmVectorRep
{
private:
    int ref_count;
    int N;
    number * elems;
public:
    fglmVectorRep() : ref_count(1), N(0), elems(0) {}
    fglmVectorRep( int n, number * e ) : ref_count(1), N(n), elems(e) {}
    fglmVectorRep( int n ) : ref_count(1), N(n)
    {
        if ( N == 0 )
            elems= 0;
        else {
            elems= (number *)Alloc( N*sizeof( number ) );
            for ( int i= N-1; i >= 0; i-- )
                elems[i]= nInit( 0 );
        }
    }
    ~fglmVectorRep()
    {
        if ( N > 0 ) {
            for ( int i= N-1; i >= 0; i-- )
                nDelete( elems + i );
            Free( (ADDRESS)elems, N*sizeof( number ) );
        }
    }

    fglmVectorRep* clone() const
    {
        if ( N > 0 ) {
            number * elems_clone;
            elems_clone= (number *)Alloc( N*sizeof( number ) );
            for ( int i= N-1; i >= 0; i-- )
                elems_clone[i] = nCopy( elems[i] );
            return new fglmVectorRep( N, elems_clone );
        } else
            return new fglmVectorRep( N, 0 );
    }
    BOOLEAN deleteObject() { return --ref_count == 0; }
    fglmVectorRep * copyObject() { ref_count++; return this; }
    int refcount() const { return ref_count; }
```

Die so unscheinbar anmutende Methode `isUnique` ist äußerst wichtig, da sie feststellt, ob der Vektor mehrfach referenziert wird oder nicht.

```
    BOOLEAN isUnique() const { return ref_count == 1; }

    int size() const { return N; }
    int isZero() const
    {
        int k;
        for ( k= N; k > 0; k-- )
            if ( ! nIsZero( getconstelem( k ) ) )
                return 0;
        return 1;
    }
    int numNonZeroElems() const
    {
        int num = 0;
        int k;
        for ( k= N; k > 0; k-- )
            if ( ! nIsZero( getconstelem( k ) ) ) num++;
        return num;
    }
```

Bei den Zugriffsfunktionen auf die Vektorelemente ist zu beachten: Beim Setzen eines Elementes wird die Zahl nicht kopiert, die Zahl „gehört“ nun dem Vektor, dieser übernimmt auch das Löschen. Beim Auslesen wird ebenfalls nicht kopiert, dies muss man gegebenenfalls von Hand erledigen.

```
    void setelem( int i, number n )
    {
        nDelete( elems + i-1 );
        elems[i-1]= n;
    }
    number & getelem( int i )
    {
        return elems[i-1];
    }
    const number getconstelem( int i) const
    {
        return elems[i-1];
    }
    friend class fglmVector;
};
```

Für die Vektorklasse gibt es vier verschiedene Konstruktoren:
`fglmVector( fglmVectorRep * r )`: Dieser Konstruktor findet nur internen Gebrauch, da die Klasse `fglmVectorRep` nach außen nicht sichtbar ist.
`fglmVector()`: Der default-Konstruktor erzeugt einen leeren Vektor der Länge Null.

`fglmVector( int size )`: Dieser Aufruf erzeugt einen Vektor der Länge `size`, die Einträge sind mit Null initialisiert.
`fglmVector( int size, int basis )`: Dies erzeugt einen Vektor der Länge `size`. In der Komponente `basis` steht eine Eins, sonst ist dieser Vektor mit Nullen initialsiert.

```
fglmVector::fglmVector( fglmVectorRep * r ) : rep(r) {}
fglmVector::fglmVector() : rep( new fglmVectorRep() ) {}
fglmVector::fglmVector( int size ) : rep( new fglmVectorRep( size ) ) {}
fglmVector::fglmVector( int size, int basis ) : rep( new fglmVectorRep( size ) )
{
    rep->setelem( basis, nInit( 1 ) );
}
fglmVector::fglmVector( const fglmVector & v )
{
    rep= v.rep->copyObject();
}
fglmVector::~fglmVector()
{
    if ( rep->deleteObject() )
        delete rep;
}
```

Die Methode `makeUnique` sorgt dafür, dass der Vektor gegebenenfalls kopiert wird, falls er mehrmals referenziert wird und man schreibend zugreifen will.

```
void
fglmVector::makeUnique()
{
    if ( rep->refcount() != 1 ) {
        rep->deleteObject();
        rep= rep->clone();
    }
}

int
fglmVector::size() const
{
    return rep->size();
}

int
fglmVector::numNonZeroElems() const
{
    return rep->numNonZeroElems();
}
```

Aus Effizienzgründen machen die nachfolgenden Arithmetikroutinen meist eine Unterscheidung, ob der Vektor mehrfach referenziert wird oder nicht.

```
void
fglmVector::nihilate( const number fac1, const number fac2,
                      const fglmVector v )
{
    int i;
    int vsize= v.size();
    number term1, term2;
    if ( rep->isUnique() ) {
        for ( i= vsize; i > 0; i-- ) {
            term1= nMult( fac1, rep->getconstelem( i ) );
            term2= nMult( fac2, v.rep->getconstelem( i ) );
            rep->setelem( i, nSub( term1, term2 ) );
            nDelete( &term1 );
            nDelete( &term2 );
        }
        for ( i= rep->size(); i > vsize; i-- ) {
            rep->setelem( i, nMult( fac1, rep->getconstelem( i ) ) );
        }
    }
    else
    {
        number* newelems;
        newelems= (number *)Alloc( rep->size()*sizeof( number ) );
        for ( i= vsize; i > 0; i-- ) {
            term1= nMult( fac1, rep->getconstelem( i ) );
            term2= nMult( fac2, v.rep->getconstelem( i ) );
            newelems[i-1]= nSub( term1, term2 );
            nDelete( &term1 );
            nDelete( &term2 );
        }
        for ( i= rep->size(); i > vsize; i-- ) {
            newelems[i-1]= nMult( fac1, rep->getconstelem( i ) );
        }
        rep->deleteObject();
        rep= new fglmVectorRep( rep->size(), newelems );
    }
}

fglmVector &
fglmVector::operator = ( const fglmVector & v )
{
    if ( this != &v ) {
        if ( rep->deleteObject() )
            delete rep;
        rep = v.rep->copyObject();
    }
    return *this;
}
```

```
int
fglmVector::operator == ( const fglmVector & v )
{
    if ( rep->size() == v.rep->size() ) {
        if ( rep == v.rep ) return 1;
        else {
            int i;
            for ( i= rep->size(); i > 0; i-- )
                if ( ! nEqual( rep->getconstelem( i ),
                               v.rep->getconstelem( i ) ) )
                    return 0;
            return 1;
        }
    }
    return 0;
}

int
fglmVector::operator != ( const fglmVector & v )
{
    return !( *this == v );
}

int
fglmVector::isZero()
{
    return rep->isZero();
}

int
fglmVector::elemIsZero( int i )
{
    return nIsZero( rep->getconstelem( i ) );
}

fglmVector &
fglmVector::operator += ( const fglmVector & v )
{
    int i;
    if ( rep->isUnique() ) {
        for ( i= rep->size(); i > 0; i-- )
            rep->setelem( i, nAdd( rep->getconstelem( i ),
                                   v.rep->getconstelem( i ) ) );
    }
    else
    {
        int n = rep->size();
        number* newelems;
        newelems= (number *)Alloc( n*sizeof( number ) );
        for ( i= n; i > 0; i-- )
            newelems[i-1]= nAdd( rep->getconstelem( i ),
                                 v.rep->getconstelem( i ) );
        rep->deleteObject();
        rep= new fglmVectorRep( n, newelems );
    }
    return *this;
}

fglmVector &
fglmVector::operator -= ( const fglmVector & v )
{
    int i;
    if ( rep->isUnique() ) {
        for ( i= rep->size(); i > 0; i-- )
            rep->setelem( i, nSub( rep->getconstelem( i ),
                                   v.rep->getconstelem( i ) ) );
    }
    else
    {
        int n = rep->size();
        number* newelems;
        newelems= (number *)Alloc( n*sizeof( number ) );
        for ( i= n; i > 0; i-- )
            newelems[i-1]= nSub( rep->getconstelem( i ),
                                 v.rep->getconstelem( i ) );
        rep->deleteObject();
        rep= new fglmVectorRep( n, newelems );
    }
    return *this;
}

fglmVector &
fglmVector::operator *= ( const number & n )
{
    int s= rep->size();
    int i;
    if ( ! rep->isUnique() ) {
        number * temp;
        temp= (number *)Alloc( s*sizeof( number ) );
        for ( i= s; i > 0; i-- )
            temp[i-1]= nMult( rep->getconstelem( i ), n );
        rep->deleteObject();
        rep= new fglmVectorRep( s, temp );
    }
    else
```

```
    {
        for (i= s; i > 0; i-- )
            rep->setelem( i, nMult( rep->getconstelem( i ), n ) );
    }
    return *this;
}

fglmVector &
fglmVector::operator /= ( const number & n )
{
    int s= rep->size();
    int i;
    if ( ! rep->isUnique() ) {
        number * temp;
        temp= (number *)Alloc( s*sizeof( number ) );
        for ( i= s; i > 0; i-- ) {
            temp[i-1]= nDiv( rep->getconstelem( i ), n );
            nNormalize( temp[i-1] );
        }
        rep->deleteObject();
        rep= new fglmVectorRep( s, temp );
    }
    else
    {
        for (i= s; i > 0; i-- ) {
            rep->setelem( i, nDiv( rep->getconstelem( i ), n ) );
            nNormalize( rep->getelem( i ) );
        }
    }
    return *this;
}

fglmVector
operator - ( const fglmVector & v )
{
    fglmVector temp( v.size() );
    int i;
    number n ;
    for ( i= v.size(); i > 0; i-- ) {
        n= nCopy( v.getconstelem( i ) );
        n= nNeg( n );
        temp.setelem( i, n );
    }
    return temp;
}

fglmVector
operator + ( const fglmVector & lhs, const fglmVector & rhs )
{
    fglmVector temp= lhs;
    temp+= rhs;
    return temp;
}

fglmVector
operator - ( const fglmVector & lhs, const fglmVector & rhs )
{
    fglmVector temp= lhs;
    temp-= rhs;
    return temp;
}

fglmVector
operator * ( const fglmVector & v, const number n )
{
    fglmVector temp= v;
    temp*= n;
    return temp;
}

fglmVector
operator * ( const number n, const fglmVector & v )
{
    fglmVector temp= v;
    temp*= n;
    return temp;
}

number &
fglmVector::getelem( int i )
{
    makeUnique();
    return rep->getelem( i );
}

const number
fglmVector::getconstelem( int i ) const
{
    return rep->getconstelem( i );
}

void
fglmVector::setelem( int i, number & n )
{
    makeUnique();
    rep->setelem( i, n );
```

```
    n= nNULL;
}

number
fglmVector::gcd() const
{
    int i= rep->size();
    BOOLEAN found = FALSE;
    BOOLEAN gcdIsOne = FALSE;
    number theGcd;
    number current;
    while( i > 0 && ! found ) {
        current= rep->getconstelem( i );
        if ( ! nIsZero( current ) ) {
            theGcd= nCopy( current );
            found= TRUE;
            if ( ! nGreaterZero( theGcd ) ) {
                theGcd= nNeg( theGcd );
            }
            if ( nIsOne( theGcd ) ) gcdIsOne= TRUE;
        }
        i--;
    }
    if ( found ) {
        while ( i > 0 && ! gcdIsOne ) {
            current= rep->getconstelem( i );
            if ( ! nIsZero( current ) ) {
                number temp= nGcd( theGcd, current );
                nDelete( &theGcd );
                theGcd= temp;
                if ( nIsOne( theGcd ) ) gcdIsOne= TRUE;
            }
            i--;
        }
    }
    else
        theGcd= nInit( 0 );
    return theGcd;
}

number
fglmVector::clearDenom()
{
    number theLcm = nInit( 1 );
    number current;
    BOOLEAN isZero = TRUE;
    int i;
    for ( i= size(); i > 0; i-- ) {
        if ( ! nIsZero( rep->getconstelem(i) ) ) {
            isZero= FALSE;
            number temp= nLcm( theLcm, rep->getconstelem( i ) );
            nDelete( &theLcm );
            theLcm= temp;
        }
    }
    if ( isZero ) {
        nDelete( &theLcm );
        theLcm= nInit( 0 );
    }
    else {
        if ( ! nIsOne( theLcm ) ) {
            *this *= theLcm;
            for ( i= size(); i > 0; i-- ) {
                nNormalize( rep->getelem( i ) );
            }
        }
    }
    return theLcm;
}
```

fglmgauss.h

Alle im Rahmen dieser Arbeit vorgestellten Algorithmen haben gemeinsam, dass sie Gaussreduktion anwenden (vergleiche Seite 49). Daher lag es nahe, für diese Aufgabe eine eigene Klasse zu implementieren, die Klasse `gaussReducer`. Diese testet eine Menge von Vektoren auf lineare Abhängigkeit und gibt gegebenenfalls die Koeffizienten der Abhängigkeit zurück. Hier wurde die Funktionalität noch genauer den Erfordernissen angepasst, dass die auf lineare Abhängigkeit zu testende Menge in jedem Schritt um genau einen Vektor wächst. Wie schon auf Seite 49 angedeutet, ist es in dieser Situation sinnvoll, sich die Arbeit, die man in vorherigen Schritten geleistet hat, zu merken und auszunutzen. Das heißt, dass man die bisherigen Vektoren in oberer Dreiecksgestalt belässt, um mit ihnen in Zukunft effizienter kombinieren zu können. Die entsprechenden Spaltenvertauschungen werden dabei in einem gesonderten Vektor vermerkt.

Im Unterschied zum klassischen Gauss-Algorithmus werden die Diagonalelemente nicht auf Eins normiert. Dies hat den Vorteil, dass im Laufe der Rechnung keine Brüche auftreten können.

```
class gaussReducer
{
private:
    int size;
    int max;
    gaussElem * elems;
    BOOLEAN * isPivot;
    int * perm;
```

```
    fglmVector v;
    fglmVector p;
    number pdenom;
public:
    gaussReducer( int dimen );
    ~gaussReducer();
```

Die Methode **reduce** versucht den übergebenen Vektor mittels der zur Zeit in der Klasse gespeicherten Vektoren linear zu Null zu kombinieren. Wenn dies geglückt ist, so wird **TRUE** zurückgegeben, andernfalls **FALSE**. Falls dies nicht möglich ist, so kann man mittels **store** diesen Vektor in der Klasse abspeichern lassen. Er steht dann für zukünftiges Kombinieren zur Verfügung. Falls die Linearkombination jedoch gelungen ist, so liefert die Methode **getDependence** einen Vektor zurück, der genau die Koeffizienten enthält, die die lineare Abhängigkeit herbeiführen.

```
    BOOLEAN reduce( fglmVector v );
    void store();
    fglmVector getDependence();
};
```

fglmgauss.cc

Die Klasse **gaussReducer** führt ein (gegebenenfalls wachsendes) Array von Objekten des Typs **gaussElem**. Diese Klasse enthält jeweils einen Vektor mit dem zugehörigen Permutationsvektor. Außerdem ist dort das Pivot-Element gespeichert.

```
class gaussElem
{
public:
    fglmVector v;
    fglmVector p;
    number pdenom;
    number fac;
    gaussElem( const fglmVector newv, const fglmVector newp,
               number & newpdenom, number & newfac ) :
        v( newv ), p( newp ), pdenom( newpdenom ), fac( newfac )
    {
        newpdenom= NULL;
        newfac= NULL;
    }
    ~gaussElem()
    {
        nDelete( & pdenom );
        nDelete( & fac );
    }
};
```

```
gaussReducer::gaussReducer( int dimen )
{
    int k;
    size= 0;
    max= dimen;
    elems= (gaussElem *)Alloc( (max+1)*sizeof( gaussElem ) );
    isPivot= (BOOLEAN *)Alloc( (max+1)*sizeof( BOOLEAN ) );
    for ( k= max; k > 0; k-- )
            isPivot[k]= FALSE;
    perm= (int *)Alloc( (max+1)*sizeof( int ) );
}

gaussReducer::~gaussReducer()
{
    int k;
    for ( k= size; k > 0; k-- )
        elems[k].~gaussElem();
    Free( (ADDRESS)elems, (max+1)*sizeof( gaussElem ) );
    Free( (ADDRESS)isPivot, (max+1)*sizeof( BOOLEAN ) );
    Free( (ADDRESS)perm, (max+1)*sizeof( int ) );
}

BOOLEAN
gaussReducer::reduce( fglmVector thev )
{
    number fac1, fac2;
    number temp;
    v= thev;
    p= fglmVector( size + 1, size + 1 );
    pdenom= nInit( 1 );
    number vdenom = v.clearDenom();
    if ( ! nIsOne( vdenom ) && ! nIsZero( vdenom ) ) {
        p.setelem( p.size(), vdenom );
    }
    else {
        nDelete( & vdenom );
    }
    number gcd = v.gcd();
    if ( ! nIsOne( gcd ) && ! nIsZero( gcd ) ) {
        v /= gcd;
        number temp= nMult( pdenom, gcd );
        nDelete( & pdenom );
        pdenom= temp;
    }
    nDelete( & gcd );

    int k;
    for ( k= 1; k <= size; k++ ) {
```

```
        if ( ! v.elemIsZero( perm[k] ) ) {
            fac1= elems[k].fac;
            fac2= nCopy( v.getconstelem( perm[k] ) );
            v.nihilate( fac1, fac2, elems[k].v );
            fac1= nMult( fac1, elems[k].pdenom );
            temp= nMult( fac2, pdenom );
            nDelete( & fac2 );
            fac2= temp;
            p.nihilate( fac1, fac2, elems[k].p );
            temp= nMult( pdenom, elems[k].pdenom );
            nDelete( & pdenom );
            pdenom= temp;

            nDelete( & fac1 );
            nDelete( & fac2 );
            number gcd = v.gcd();
            if ( ! nIsOne( gcd ) && ! nIsZero( gcd ) ) {
                v/= gcd;
                number temp = nMult( pdenom, gcd );
                nDelete( & pdenom );
                pdenom= temp;
            }
            nDelete( & gcd );
            gcd= p.gcd();
            temp= nGcd( pdenom, gcd );
            nDelete( & gcd );
            gcd= temp;
            if ( ! nIsZero( gcd ) && ! nIsOne( gcd ) ) {
                p/= gcd;
                temp= nDiv( pdenom, gcd );
                nDelete( & pdenom );
                pdenom= temp;
                nNormalize( pdenom );
            }
            nDelete( & gcd );
        }
    }
    return ( v.isZero() );
}

void
gaussReducer::store()
{
    number fac;

    size++;
    int k= 1;
    while ( nIsZero(v.getconstelem(k)) || isPivot[k] ) {
        k++;
    }
    number pivot= v.getconstelem( k );
    int pivotcol = k;
    k++;
    while ( k <= max ) {
        if ( ! nIsZero( v.getconstelem(k) ) && ! isPivot[k] ) {
            if ( nGreater( v.getconstelem( k ), pivot ) ) {
                pivot= v.getconstelem( k );
                pivotcol= k;
            }
        }
        k++;
    }
    isPivot[ pivotcol ]= TRUE;
    perm[size]= pivotcol;

    pivot= nCopy( v.getconstelem( pivotcol ) );
    elems[size].gaussElem( v, p, pdenom, pivot );
}

fglmVector
gaussReducer::getDependence()
{
    nDelete( & pdenom );
    fglmVector result = p;
    p= fglmVector();
    return ( result );
}
```

fglm.h

Die Klassen **`fglmSelem`** und **`fglmDelem`** speichern die relevanten Informationen der „Kandidaten“, die bei der Berechnung der Multiplikationsmatrizen bzw. der Gröbnerbasis benötigt werden (vergleiche Algorithmus 4 und 5).

```
class fglmSelem
{
public:
    int numVars;
    poly monom;
    int * divisors;
    fglmSelem( poly p, int var );

    void cleanup();
    BOOLEAN isBasisOrEdge() const { return ( (divisors[0] == numVars) ?
```

```
                         TRUE : FALSE ); }
    void newDivisor( int var ) { divisors[ ++divisors[0] ]= var; }
};

class fglmDelem
{
public:
    int insertions;
    poly monom;
    fglmVector v;
    int var;
    fglmDelem( poly & m, fglmVector mv, int v );

    void cleanup();
    BOOLEAN isBasisOrEdge() const { return ( (insertions == 0) ?
                         TRUE : FALSE ); }
    void newDivisor() { insertions--; }
};
```

Über die Funktion `fglmzero` wird der nulldimensionale FGLM-Algorithmus angesprochen. Die Parameter haben die folgende Bedeutung:

- `sourceRingHdl` und `destRingHdl` geben den Quell- und Zielring an.
- `sourceIdeal` ist eine reduzierte Gröbnerbasis im Quellring, in `destIdeal` wird die reduzierte Gröbnerbasis im Zielring zurückgegeben.
- der Schalter `switchBack` gibt an, ob nach der Rechnung in den Quellring zurückgeschaltet werden soll.
- `deleteIdeal` gibt an, ob das Quellideal nach erfolgter Berechnung gelöscht werden soll.

Die beiden Schalter werden benötigt, wenn man in einem `qring` rechnet. Die Funktion überprüft nicht, ob das Ideal wirklich nulldimensional ist. Wenn das Ideal minimal, aber nicht reduziert ist, wird `FALSE` zurückgegeben, ansonsten `TRUE`.

Vom Interpreter aus wird die Routine `fglmProc` aufgerufen, die die nötigen Überprüfungen vornimmt, um den FGLM-Algorithmus zu starten. Diese gibt `TRUE` zurück, wenn ein Fehler aufgetreten ist.

```
BOOLEAN
fglmzero( idhdl sourceRingHdl, ideal & sourceIdeal, idhdl destRingHdl,
      ideal & destideal,
      BOOLEAN switchBack = TRUE, BOOLEAN deleteIdeal = FALSE );

BOOLEAN
fglmProc( leftv result, leftv first, leftv second );

ideal
FindUnivariateWrapper( ideal source );
```

fglm.cc

```
enum FglmState{
    FglmOk,
    FglmHasOne,
    FglmNoIdeal,
    FglmNotReduced,
    FglmNotZeroDim,
    FglmIncompatibleRings
};
```

Die Routine `fglmUpdatesource` wird benötigt, wenn die Berechnung in Singular in einem `qring` stattfindet. Dann muss zur Vorbereitung das Ideal um gewisse Erzeuger des Quotientenideals erweitert werden, bevor die Gröbnerbasenberechnung stattfinden kann. Am Ende der Berechnung müssen dann wiederum gewisse Erzeuger aus dem Ergebnis herausgenommen werden, um ein korrektes Ergebnis im `qring` zu erhalten. Dies erledigt `fglmUpdateresult`.

```
ideal fglmUpdatesource( const ideal sourceIdeal )
{
    // assumes that currRing == sourceRing
    int k, l, offset;
    BOOLEAN found;
    ideal newSource= idInit( IDELEMS( sourceIdeal )
                         + IDELEMS( currQuotient ), 1 );
    for ( k= IDELEMS( sourceIdeal )-1; k >=0; k-- )
        (newSource->m)[k]= pCopy( (sourceIdeal->m)[k] );
    offset= IDELEMS( sourceIdeal );
    for ( l= IDELEMS( currQuotient )-1; l >= 0; l-- ) {
        if ( (currQuotient->m)[l] != NULL ) {
            found= FALSE;
            for ( k= IDELEMS( sourceIdeal )-1;
                 (k >= 0) && (found == FALSE); k-- ) {
                if ( pDivisibleBy( (sourceIdeal->m)[k], (currQuotient->m)[l] ) )
                    found= TRUE;
                if ( ! found ) {
                    (newSource->m)[offset]= pCopy( (currQuotient->m)[l] );
                    offset++;
                }
            }
        }
    }
    idSkipZeroes( newSource );
    return newSource;
}

void
fglmUpdateresult( ideal & result )
```

```
{
    // assumes that currRing == destRing
    int k, l;
    BOOLEAN found;
    for ( k= IDELEMS( result )-1; k >=0; k-- ) {
        if ( (result->m)[k] != NULL ) {
            found= FALSE;
            for ( l= IDELEMS( currQuotient )-1; (l >= 0)
                  && ( found == FALSE ); l-- )
            if ( pDivisibleBy( (currQuotient->m)[l], (result->m)[k] ) )
                found= TRUE;
            if ( found ) pDelete( & ((result->m)[k]) );
        }
    }
    idSkipZeroes( result );
}
```

Die folgende Routine überprüft, ob die beiden Ringe, zwischen denen der FGLM-Algorithmus arbeiten soll, kompatibel sind, d.h. :

- Beide Grundkörper müssen die gleiche Charakteristik haben.
- Beide Ringe müssen die gleiche Anzahl Variablen und Parameter haben, und Variablen- sowie Parameternamen müssen bis auf Permutationen in beiden Ringen übereinstimmen.
- Die Ordnung muss jeweils eine Wohlordnung sein.
- Entweder beide Ringe sind `qring`e, oder nicht. Im ersteren Fall müssen die Quotientenideale übereinstimmen.

Wenn die Ringe kompatibel sind, wird in **`vperm`** die Permutation der Variablennamen zurückgegeben.

```
FglmState
fglmConsistency( idhdl sringHdl, idhdl dringHdl, int * vperm )
{
    // Should be called with currRing= IDRING( sringHdl );
    int k;
    FglmState state= FglmOk;
    ring dring = IDRING( dringHdl );
    ring sring = IDRING( sringHdl );

    if ( sring->ch != dring->ch ) {
        WerrorS( "rings must have same characteristic" );
        state= FglmIncompatibleRings;
    }
    if ( (sring->OrdSgn != 1) || (dring->OrdSgn != 1) ) {
        WerrorS( "only works for global orderings" );
        state= FglmIncompatibleRings;
    }
    if ( sring->N != dring->N ) {
        WerrorS( "rings must have same number of variables" );
        state= FglmIncompatibleRings;
    }
    if ( sring->P != dring->P ) {
        WerrorS( "rings must have same number of parameters" );
        state= FglmIncompatibleRings;
    }
    if ( state != FglmOk ) return state;
    int nvar = sring->N;
    int npar = sring->P;
    int * pperm;
    if ( npar > 0 )
        pperm= (int *)Alloc0( (npar+1)*sizeof( int ) );
    else
        pperm= NULL;
    maFindPerm( sring->names, nvar, sring->parameter, npar, dring->names,
                nvar, dring->parameter, npar, vperm, pperm );
    for ( k= nvar; (k > 0) && (state == FglmOk); k-- ) {
        if ( vperm[k] <= 0 ) {
            WerrorS( "variable names do not agree" );
            state= FglmIncompatibleRings;
        }
    }
    for ( k= npar-1; (k >= 0) && (state == FglmOk); k-- ) {
        if ( pperm[k] >= 0 ) {
            WerrorS( "paramater names do not agree" );
            state= FglmIncompatibleRings;
        }
    }
    Free( (ADDRESS)pperm, (npar+1)*sizeof( int ) );
    if ( state != FglmOk ) return state;
    if ( sring->qideal != NULL ) {
        if ( dring->qideal == NULL ) {
            Werror( "%s is a qring, current ring not", sringHdl->id );
            return FglmIncompatibleRings;
        }
        rSetHdl( dringHdl, TRUE );
        nSetMap( sring->ch, sring->parameter, npar, sring->minpoly );
        ideal sqind = idInit( IDELEMS( sring->qideal ), 1 );
        for ( k= IDELEMS( sring->qideal )-1; k >= 0; k-- )
            (sqind->m)[k]= pPermPoly( (sring->qideal->m)[k], vperm, nvar );
        ideal sqindred = kNF( dring->qideal, NULL, sqind );
        if ( ! idIs0( sqindred ) ) {
            WerrorS( "the quotients do not agree" );
            state= FglmIncompatibleRings;
        }
        idDelete( & sqind );
        idDelete( & sqindred );
        rSetHdl( sringHdl, TRUE );
```

```
        if ( state != FglmOk ) return state;
        int * dsvperm = (int *)Alloc0( (nvar+1)*sizeof( int ) );
        maFindPerm( dring->names, nvar, NULL, 0,
                    sring->names, nvar, NULL, 0, dsvperm, NULL );
        nSetMap( dring->ch, dring->parameter, npar, dring->minpoly );
        ideal dqins = idInit( IDELEMS( dring->qideal ), 1 );
        for ( k= IDELEMS( dring->qideal )-1; k >= 0; k-- )
            (dqins->m)[k]= pPermPoly( (dring->qideal->m)[k], dsvperm, nvar );
        ideal dqinsred = kNF( sring->qideal, NULL, dqins );
        if ( ! idIs0( dqinsred ) ) {
            WerrorS( "the quotients do not agree" );
            state= FglmIncompatibleRings;
        }
        idDelete( & dqins );
        idDelete( & dqinsred );
        Free( (ADDRESS)dsvperm, (nvar+1)*sizeof( int ) );
        if ( state != FglmOk ) return state;
    }
    else {
        if ( dring->qideal != NULL ) {
            Werror( "current ring is a qring, %s not", sringHdl->id )
            return FglmIncompatibleRings;
        }
    }
    return FglmOk;
}
```

So wie die Ringe auf Konsistenz überprüft werden müssen, muss auch sichergestellt werden, dass das Ideal nulldimensional ist. Dies wird von der folgenden Routine erledigt. Sie macht auch einen ersten Check, ob die Gröbnerbasis reduziert ist, indem sie die Leitterme auf gegenseitige Teilbarkeit testet. Eine komplette Untersuchung, ob die Basis reduziert ist, würde an dieser Stelle zu viel Zeit in Anspruch nehmen. Außerdem gibt es im Algorithmus `multmat` ein Stelle, an der man ohne zusätzlichen Aufwand feststellt, ob die Basis reduziert ist oder nicht. Siehe hierzu Seite 94.

```
FglmState
fglmIdealcheck( const ideal theIdeal )
{
    FglmState state = FglmOk;
    int power;
    int k;
    BOOLEAN * purePowers = (BOOLEAN *)Alloc( pVariables*sizeof( BOOLEAN ) );
    for ( k= pVariables-1; k >= 0; k-- )
        purePowers[k]= FALSE;

    for ( k= IDELEMS( theIdeal ) - 1; (state == FglmOk) && (k >= 0); k-- ) {
        poly p = (theIdeal->m)[k];
        if ( pIsConstant( p ) ) state= FglmHasOne;
        else if ( (power= pIsPurePower( p )) > 0 ) {
            if ( purePowers[power-1] == TRUE  ) state= FglmNotReduced;
            else purePowers[power-1]= TRUE;
        }
        for ( int l = IDELEMS( theIdeal ) - 1; state == FglmOk && l >= 0; l-- )
            if ( (k != l) && pDivisibleBy( p, (theIdeal->m)[l] ) )
                state= FglmNotReduced;
    }
    if ( state == FglmOk ) {
        for ( k= pVariables-1 ; (state == FglmOk) && (k >= 0); k-- )
            if ( purePowers[k] == FALSE ) state= FglmNotZeroDim;
    }
    Free( (ADDRESS)purePowers, pVariables*sizeof( BOOLEAN ) );
    return state;
}
```

Die Routine `fglmProc` handelt die Berechnung einer nulldimensionalen Gröbnerbasis ab. Hier werden die nötigen Überprüfungen veranlasst, der `qring`-Fall abgefangen und schließlich `fglmzero` aufgerufen.

```
BOOLEAN
fglmProc( leftv result, leftv first, leftv second )
{
    FglmState state = FglmOk;

    idhdl destRingHdl = currRingHdl;
    ring destRing = currRing;
    ideal destIdeal = NULL;
    idhdl sourceRingHdl = (idhdl)first->data;
    rSetHdl( sourceRingHdl, TRUE );
    ring sourceRing = currRing;

    int * vperm = (int *)Alloc0( (pVariables+1)*sizeof( int ) );
    state= fglmConsistency( sourceRingHdl, destRingHdl, vperm );
    Free( (ADDRESS)vperm, (pVariables+1)*sizeof(int) );

    if ( state == FglmOk ) {
        idhdl ih = currRing->idroot->get( second->Name(), myynest );
        if ( (ih != NULL) && (IDTYP(ih)==IDEAL_CMD) ) {
            ideal sourceIdeal;
            if ( currQuotient != NULL )
                sourceIdeal= fglmUpdatesource( IDIDEAL( ih ) );
            else
                sourceIdeal = IDIDEAL( ih );
            state= fglmIdealcheck( sourceIdeal );
            if ( state == FglmOk ) {
                assumeStdFlag( (leftv)ih );
                if ( fglmzero( sourceRingHdl, sourceIdeal,
```

```
                              destRingHdl, destIdeal,
                              FALSE, (currQuotient != NULL) ) == FALSE )
                    state= FglmNotReduced;
                }
            }
            else state= FglmNoIdeal;
    }
    if ( currRingHdl != destRingHdl ) rSetHdl( destRingHdl, TRUE );
    switch (state) {
        case FglmOk:
            if ( currQuotient != NULL ) fglmUpdateresult( destIdeal );
            break;
        case FglmHasOne:
            destIdeal= idInit(1,1);
            (destIdeal->m)[0]= pOne();
            state= FglmOk;
            break;
        case FglmIncompatibleRings:
            Werror( "ring %s and current ring are incompatible",
                    first->Name() );
            destIdeal= idInit(0,0);
            break;
        case FglmNoIdeal:
            Werror( "Can't find ideal %s in ring %s", second->Name(),
                    first->Name() );
            destIdeal= idInit(0,0);
            break;
        case FglmNotZeroDim:
            Werror( "The ideal %s has to be 0-dimensional", second->Name() );
            destIdeal= idInit(0,0);
            break;
        case FglmNotReduced:
            Werror( "The ideal %s has to be reduced", second->Name() );
            destIdeal= idInit(0,0);
            break;
        default:
            destIdeal= idInit(1,1);
    }
    result->rtyp = IDEAL_CMD;
    result->data= (void *)destIdeal;
    setFlag( result, FLAG_STD );
    if ( state == FglmOk )
    return FALSE;
    else
    return TRUE;
}
```

fglmzero.cc

Die Klasse **idealFunctionals** ist die Darstellung der Multiplikationsmatrizen. Sie ist so ausgelegt, dass die Größe der Matrizen während der Berechnung noch wachsen kann. Außerdem werden nur Einträge (**matElem**) ungleich Null abgespeichert, und Spalten (**matHeader**), die in mehreren Matrizen auftauchen, werden mehrfach referenziert. Der Eintrag **owner** in der Klasse **matHeader** gibt an, welche Matrix die entsprechende Spalte besitzt und sich für die Freigabe des Speichers verantwortlich zeichnet.

```
struct matElem
{
    int row;
    number elem;
};

struct matHeader
{
    int size;
    BOOLEAN owner;
    matElem * elems;
};

class idealFunctionals
{
private:
    int _block;
    int _max;
    int _size;
    int _nfunc;
    int * currentSize;
    matHeader ** func;
    matHeader * grow( int var );
public:
    idealFunctionals( int blockSize, int numFuncs );
    ~idealFunctionals();
```

Die Verwendung der **idealFunctionals** gliedert sich in zwei Teile: Die Erstellungsphase und die Benutzungsphase. Während der Erstellung können Spalten über die Funktion **addCols** hinzugefügt werden, wobei die Matrizen automatisch mitwachsen (über die private Methode **grow**). Ist diese Phase abgeschlossen, so ruft man **endofConstruction** auf, wodurch die Größe der Matrizen festgelegt wird.

```
    int dimen() const { return _size; }
    void endofConstruction();
    void map( ring source );
    void insertCols( int * divisors, int to );
    void insertCols( int * divisors, const fglmVector to );
    fglmVector addCols( const int var, int basisSize,
```

```
                        const fglmVector v ) const;
    fglmVector multiply( const fglmVector v, int var ) const;
};

idealFunctionals::idealFunctionals( int blockSize, int numFuncs )
{
    int k;
    _block= blockSize;
    _max= _block;
    _size= 0;
    _nfunc= numFuncs;

    currentSize= (int *)Alloc( _nfunc*sizeof( int ) );
    for ( k= _nfunc-1; k >= 0; k-- )
        currentSize[k]= 0;

    func= (matHeader **)Alloc( _nfunc*sizeof( matHeader * ) );
    for ( k= _nfunc-1; k >= 0; k-- )
        func[k]= (matHeader *)Alloc( _max*sizeof( matHeader ) );
}

idealFunctionals::~idealFunctionals()
{
    int k;
    int l;
    int row;
    matHeader * colp;
    matElem * elemp;
    for ( k= _nfunc-1; k >= 0; k-- ) {
        for ( l= _size-1, colp= func[k]; l >= 0; l--, colp++ ) {
            if ( ( colp->owner == TRUE ) && ( colp->size > 0 ) ) {
                for ( row= colp->size-1, elemp= colp->elems; row >= 0;
                      row--, elemp++ )
                    nDelete( & elemp->elem );
                Free( (ADDRESS)colp->elems, colp->size*sizeof( matElem ) );
            }
        }
        Free( (ADDRESS)func[k], _max*sizeof( matHeader ) );
    }
    Free( (ADDRESS)func, _nfunc*sizeof( matHeader * ) );
    Free( (ADDRESS)currentSize, _nfunc*sizeof( int ) );
}

void
idealFunctionals::endofConstruction()
{
    _size= currentSize[0];
}
```

Die Methode `map` wird benötigt, wenn die Reihenfolge der Ringvariablen in den beiden Ringen nicht übereinstimmt. Nach einem Ringwechsel werden die entsprechenden Permutationen hier durchgeführt.

```
void
idealFunctionals::map( ring source )
{
    int var, col, row;
    matHeader * colp;
    matElem * elemp;
    number newelem;

    int * perm = (int *)Alloc0( (_nfunc+1)*sizeof( int ) );
    maFindPerm( source->names, source->N, NULL, 0,
                currRing->names, currRing->N, NULL, 0, perm, NULL );
    nSetMap( source->ch, source->parameter, source->P, source->minpoly );

    matHeader ** temp = (matHeader **)Alloc( _nfunc*sizeof( matHeader * ));
    for ( var= 0; var < _nfunc; var ++ ) {
        for ( col= 0, colp= func[var]; col < _size; col++, colp++ ) {
            if ( colp->owner == TRUE ) {
                for ( row= colp->size-1, elemp= colp->elems; row >= 0;
                      row--, elemp++ ) {
                    newelem= nMap( elemp->elem );
                    nDelete( & elemp->elem );
                    elemp->elem= newelem;
                }
            }
        }
        temp[ perm[var+1]-1 ]= func[var];
    }
    Free( (ADDRESS)func, _nfunc*sizeof( matHeader * ) );
    Free( (ADDRESS)perm, (_nfunc+1)*sizeof( int ) );
    func= temp;
}

matHeader *
idealFunctionals::grow( int var )
{
    if ( currentSize[var-1] == _max ) {
    int k;
    for ( k= _nfunc; k > 0; k-- )
        func[k-1]= (matHeader *)
                   ReAlloc( func[k-1], _max*sizeof( matHeader ),
                            (_max + _block)*sizeof( matHeader ) );
        _max+= _block;
    }
```

```
    currentSize[var-1]++;
    return func[var-1] + currentSize[var-1] - 1;
}

void
idealFunctionals::insertCols( int * divisors, int to )
{
    int k;
    BOOLEAN owner = TRUE;
    matElem * elems = (matElem *)Alloc( sizeof( matElem ) );
    elems->row= to;
    elems->elem= nInit( 1 );
    for ( k= divisors[0]; k > 0; k-- ) {
        matHeader * colp = grow( divisors[k] );
        colp->size= 1;
        colp->elems= elems;
        colp->owner= owner;
        owner= FALSE;
    }
}

void
idealFunctionals::insertCols( int * divisors, const fglmVector to )
{
    int k, l;
    int numElems= to.numNonZeroElems();
    matElem * elems;
    matElem * elemp;
    BOOLEAN owner = TRUE;
    if ( numElems > 0 ) {
        elems= (matElem *)Alloc( numElems * sizeof( matElem ) );
        for ( k= 1, l= 1, elemp= elems; k <= numElems; k++, elemp++ ) {
            while ( nIsZero( to.getconstelem(l) ) ) l++;
            elemp->row= l;
            elemp->elem= nCopy( to.getconstelem( l ) );
            l++;
        }
    }
    else
    elems= NULL;
    for ( k= divisors[0]; k > 0; k-- ) {
        matHeader * colp = grow( divisors[k] );
        colp->size= numElems;
        colp->elems= elems;
        colp->owner= owner;
        owner= FALSE;
    }
}
```

Während der Erstellung der Multiplikationsmatrizen ist es schon notwendig, gewisse Spalten der bis hier aufgebauten Matrizen aufzuaddieren (vergleiche Zeile 23 im Algorithmus 4). Dies kann mit der Methode `addCols` geschehen.

Die Methode `multiply` stellt die Matrix-Vektor-Multiplikation dar, die aufzurufen ist, nachdem die Matrizen vollständig erstellt sind.

```
fglmVector
idealFunctionals::addCols( const int var, int basisSize,
                           const fglmVector v ) const
{
    fglmVector result( basisSize );
    matHeader * colp;
    matElem * elemp;
    number factor, temp;
    int k, l;
    int vsize = v.size();

    for ( k= 1, colp= func[var-1]; k <= vsize; k++, colp++ ) {
        factor= v.getconstelem( k );
        if ( ! nIsZero( factor ) ) {
            for ( l= colp->size-1, elemp= colp->elems; l >= 0;
                  l--, elemp++ ) {
                temp= nMult( factor, elemp->elem );
                number newelem= nAdd( result.getconstelem( elemp->row ),
                                      temp );
                nDelete( & temp );
                nNormalize( newelem );
                result.setelem( elemp->row, newelem );
            }
        }
    }
    return result;
}

fglmVector
idealFunctionals::multiply( const fglmVector v, int var ) const
{
    fglmVector result( _size );
    matHeader * colp;
    matElem * elemp;
    number factor, temp;
    int k, l;
    for ( k= 1, colp= func[var-1]; k <= _size; k++, colp++ ) {
        factor= v.getconstelem( k );
        if ( ! nIsZero( factor ) ) {
            for ( l= colp->size-1, elemp= colp->elems; l >= 0;
                  l--, elemp++ ) {
```

```
                temp= nMult( factor, elemp->elem );
                number newelem= nAdd( result.getconstelem( elemp->row ),
                                      temp );
                nDelete( & temp );
                nNormalize( newelem );
                result.setelem( elemp->row, newelem );
            }
        }
    }
    return result;
}
```

Der folgende Abschnitt beschäftigt sich mit der Berechnung der Multiplikationsmatrizen. In der Klasse **borderElem** werden wichtige Informationen über Monome aus dem Rand R gespeichert, die benötigt werden, um Zeile 23 im Algorithmus 4 durchführen zu können.

Wie oben schon erwähnt, enthält die Klasse **fglmSelem** die Information über die „Kandidaten", die noch zu bearbeiten sind.

```
class borderElem
{
public:
    poly monom;
    fglmVector nf;
    borderElem() : monom(NULL), nf() {}
    borderElem( poly p, fglmVector n ) : monom( p ), nf( n ) {}
    ~borderElem() { pDelete1( &monom ); }
};

fglmSelem::fglmSelem( poly p, int var ) : numVars( 0 ), monom( p )
{
    for ( int k = pVariables; k > 0; k-- )
        if ( pGetExp( monom, k ) > 0 )
            numVars++;
    divisors= (int *)Alloc( (numVars+1)*sizeof( int ) );
    divisors[0]= 0;
    newDivisor( var );
}

void
fglmSelem::cleanup()
{
    Free( (ADDRESS)divisors, (numVars+1)*sizeof( int ) );
}
```

Die Klasse **fglmSdata** stellt die nötige Funktionalität zur Verfügung, die zur Berechnung der Matrizen vohanden sein muss. Hier werden die Kandidaten verwaltet, die Basis- und Randmonome gespeichert und diverse Informationen bereitgestellt.

Die Menge der noch zu bearbeitenden Kandidaten wird als doppelt verzeigerte, sortierte Liste realisiert. Hierzu wird eine Template-Listenklasse verwendet, die von **factory** zur Verfügung gestellt wird.

```
class fglmSdata
{
private:
    ideal theIdeal;
    int idelems;

    int basisBS;
    int basisMax;
    int basisSize;
    polyset basis;

    int borderBS;
    int borderMax;
    int borderSize;
    borderElem * border;

    List<fglmSelem> nlist;
    BOOLEAN _state;
public:
    fglmSdata( const ideal thisIdeal );
    ~fglmSdata();

    BOOLEAN state() const { return _state; };
    int getBasisSize() const { return basisSize; };
    int newBasisElem( poly & p );
    void newBorderElem( poly & m, fglmVector v );
    BOOLEAN candidatesLeft() const {
        return ( nlist.isEmpty() ? FALSE : TRUE ); }
    fglmSelem nextCandidate();
    void updateCandidates();
    int getEdgeNumber( const poly m ) const;
    poly getSpanPoly( int number ) const {
        return pCopy( (theIdeal->m)[number-1] ); }
    fglmVector getVectorRep( const poly m );
    fglmVector getBorderDiv( const poly m, int & var ) const;
};

fglmSdata::fglmSdata( const ideal thisIdeal )
{
    theIdeal= thisIdeal;
    idelems= IDELEMS( theIdeal );

    basisBS= 100;
    basisMax= basisBS;
```

```
    basisSize= 0;
    basis= (polyset)Alloc( basisMax*sizeof( poly ) );

    borderBS= 100;
    borderMax= borderBS;
    borderSize= 0;
    border= (borderElem *)Alloc( borderMax*sizeof( borderElem ) );
    _state= TRUE;
}

fglmSdata::~fglmSdata()
{
    for ( int k = basisSize; k > 0; k-- )
        pDelete1( basis + k );
    Free( (ADDRESS)basis, basisMax*sizeof( poly ) );
    for ( int l = borderSize; l > 0; l-- )
        border[l].~borderElem();
    Free( (ADDRESS)border, borderMax*sizeof( borderElem ) );
}
```

Die Routinen **newBasisElem** und **newBorderElem** werden aufgerufen, wenn ein neues Basis- bzw. Randmonom gefunden worden ist. Im Falle eines Randmonoms wird auch gleich noch der entsprechende Vektor gespeichert.

```
int
fglmSdata::newBasisElem( poly & m )
{
    basisSize++;
    if ( basisSize == basisMax ) {
        basis= (polyset)ReAlloc( basis, basisMax*sizeof( poly ),
                                 (basisMax + basisBS)*sizeof( poly ) );
        basisMax+= basisBS;
    }
    basis[basisSize]= m;
    m= NULL;
    return basisSize;
}

void
fglmSdata::newBorderElem( poly & m, fglmVector v )
{
    borderSize++;
    if ( borderSize == borderMax ) {
        border= (borderElem *)
        ReAlloc( border, borderMax*sizeof( borderElem ),
                 (borderMax + borderBS)*sizeof( borderElem ) );
        borderMax+= borderBS;
    }
    border[borderSize].borderElem( m, v );
    m= NULL;
}

fglmSelem
fglmSdata::nextCandidate()
{
    fglmSelem result = nlist.getFirst();
    nlist.removeFirst();
    return result;
}
```

Die folgende Routine wird verwendet, um neue Elemente in die Liste der noch zu bearbeitenden Kandidaten einzusetzen. Diese Liste ist zu jeder Zeit sortiert, und das Einsetzen erfolgt in aufsteigender Reihenfolge.

```
void
fglmSdata::updateCandidates()
{
    ListIterator<fglmSelem> list = nlist;
    poly m = basis[basisSize];
    poly newmonom = NULL;
    int k = pVariables;
    BOOLEAN done = FALSE;
    int state = 0;
    while ( k >= 1 ) {
        newmonom = pCopy( m );
        pGetExp( newmonom, k )++;
        pSetm( newmonom );
        done= FALSE;
        while ( list.hasItem() && (done == FALSE) ) {
            if ( (state= pComp( list.getItem().monom, newmonom )) < 0 )
                list++;
            else done= TRUE;
        }
        if ( done == FALSE ) {
            nlist.append( fglmSelem( newmonom, k ) );
            break;
        }
        if ( state == 0 ) {
            list.getItem().newDivisor( k );
            pDelete1( &newmonom );
        }
        else {
            list.insert( fglmSelem( newmonom, k ) );
        }
        k--;
    }
```

```
    while ( --k >= 1 ) {
        newmonom= pCopy( m );
        pGetExp( newmonom, k )++;
        pSetm( newmonom );
        nlist.append( fglmSelem( newmonom, k ) );
    }
}

int
fglmSdata::getEdgeNumber( const poly m ) const
{
    for ( int k = idelems; k > 0; k-- )
        if ( pEqual( m, (theIdeal->m)[k-1] ) )
            return k;
    return 0;
}
```

Die Routine **getVectorRep** liefert einen Vektor zurück, der die Vektorraumrepräsentation in der aktuellen Basis des übergebenen Polynoms darstellt.

```
fglmVector
fglmSdata::getVectorRep( const poly p )
{
    fglmVector temp( basisSize );
    poly m = p;
    int num = basisSize;
    while ( m != NULL ) {
        int comp = pComp( m, basis[num] );
        if ( comp == 0 ) {
            number newelem = nCopy( pGetCoeff( m ) );
            temp.setelem( num, newelem );
            num--;
            m= pIter( m );
        }
        else {
            if ( comp < 0 ) {
                num--;
            }
            else {
```

Dies ist die oben angesprochene Stelle, an der man feststellen kann, ob die Gröbnerbasis reduziert war oder nicht. Wird nämlich ein Monom des Polynoms nicht in der Basis gefunden, so ist dies ein sicheres Zeichen dafür, dass die Basis nicht reduziert war.

```
                _state= FALSE;
                return temp;
            }
        }
    }
    return temp;
}

fglmVector
fglmSdata::getBorderDiv( const poly m, int & var ) const
{
    int num = borderSize;
    while ( num > 0 ) {
        poly temp = border[num].monom;
        if ( pDivisibleBy( temp, m ) ) {
            var = pVariables;
            while ( var > 0 ) {
            if ( (pGetExp( m, var ) - pGetExp( temp, var )) == 1 )
                return border[num].nf;
            var--;
            }
        }
        num--;
    }
    return fglmVector();
}
```

Die Routine **CalculateFunctionals** entspricht dem Algorithmus **multmats**. Sie berechnet die Multiplikationsmatrizen des nulldimensionalen Ideals **i** und gibt diese in **l** zurück.

```
BOOLEAN
CalculateFunctionals( const ideal & theIdeal, idealFunctionals & l )
{
    fglmSdata data( theIdeal );

    poly one = pOne();
    data.newBasisElem( one );
    data.updateCandidates();

    STICKYPROT(".");
    while ( data.candidatesLeft() == TRUE ) {
        fglmSelem candidate = data.nextCandidate();
        if ( candidate.isBasisOrEdge() == TRUE ) {
            int edge = data.getEdgeNumber( candidate.monom );
            if ( edge != 0 ) {
                poly nf = data.getSpanPoly( edge );
                pNorm( nf );
                pDelete1( &nf );
                nf= pNeg( nf );
                fglmVector nfv = data.getVectorRep( nf );
                l.insertCols( candidate.divisors, nfv );
                data.newBorderElem( candidate.monom, nfv );
                pDelete( &nf );
                STICKYPROT( "+" );
```

```
            }
            else {
                int basis= data.newBasisElem( candidate.monom );
                data.updateCandidates();
                l.insertCols( candidate.divisors, basis );
                STICKYPROT( "." );
            }
        }
        else {
            int var = 0;
            fglmVector temp = data.getBorderDiv( candidate.monom, var );
            fglmVector nfv = l.addCols( var, data.getBasisSize(), temp );
            data.newBorderElem( candidate.monom, nfv );
            l.insertCols( candidate.divisors, nfv );
            STICKYPROT( "-" );
        }
        candidate.cleanup();
    }
    l.endofConstruction();
    STICKYPROT2( "\nvdim= %i\n", data.getBasisSize() );
    return ( data.state() );
}
```

Nachdem nun die Multiplikationsmatrizen erstellt sind, folgt die Berechnung der Gröbnerbasis. Die beiden Algorithmen haben eine sehr ähnliche Struktur und unterscheiden sich teilweise nur in kleinen Details. Dennoch hat es sich gelohnt, sie nicht zusammenzufassen. Zum einen aus Geschwindigkeitsgründen, zum anderen, da so der Algorithmus zur Bestimmung der Matrizen auch von `finduni` verwendet werden konnte.

Da die einzelnen Routinen sich ähnlich sind, lohnt es sich, auch die Kommentare von oben zu konsultieren.

```
fglmDelem::fglmDelem( poly & m, fglmVector mv, int v ) :
    insertions( 0 ), v( mv ), var( v )
{
    monom= m;
    m= NULL;
    for ( int k = pVariables; k > 0; k-- )
    if ( pGetExp( monom, k ) > 0 )
        insertions++;
    newDivisor();
}

void
fglmDelem::cleanup()
{
    if ( monom != NULL ) {
        pDelete1( &monom );
    }
}

class fglmDdata
{
private:
    int dimen;
    gaussReducer gauss;
    int basisSize;
    polyset basis;

    int groebnerBS;
    int groebnerSize;
    ideal destId;

    List<fglmDelem> nlist;
public:
    fglmDdata( int dimension );
    ~fglmDdata();

    int getBasisSize() const { return basisSize; }
    BOOLEAN candidatesLeft() const {
        return ( nlist.isEmpty() ? FALSE : TRUE ); }
    fglmDelem nextCandidate();
    void newBasisElem( poly & m );
    void updateCandidates( poly m, const fglmVector v );
    void newGroebnerPoly( poly & m );
    BOOLEAN reduce( fglmVector v );
    ideal buildIdeal()
    {
        idSkipZeroes( destId );
        return destId;
    }
};

fglmDdata::fglmDdata( int dimension ) : gauss( dimension )
{
    int k;
    dimen= dimension;
    basisSize= 0;
    basis= (polyset)Alloc( (dimen+1)*sizeof( poly ) );
    groebnerBS= 16;
    groebnerSize= 0;
    destId= idInit( groebnerBS, 1 );
}

fglmDdata::~fglmDdata()
{
    int k;
```

```
    for ( k= dimen; k > 0; k-- )
        pDelete1( basis + k );
    Free( (ADDRESS)basis, (dimen+1)*sizeof( poly ) );
}

fglmDelem
fglmDdata::nextCandidate()
{
    fglmDelem result = nlist.getFirst();
    nlist.removeFirst();
    return result;
}

void
fglmDdata::newBasisElem( poly & m )
{
    basisSize++;
    basis[basisSize]= m;
    m= NULL;
    gauss.store();
}

void
fglmDdata::updateCandidates( poly m, const fglmVector v )
{
    ListIterator<fglmDelem> list = nlist;
    poly newmonom = NULL;
    int k = pVariables;
    BOOLEAN done = FALSE;
    int state = 0;
    while ( k >= 1 ) {
        newmonom = pCopy( m );
        pGetExp( newmonom, k )++;
        pSetm( newmonom );
        done= FALSE;
        while ( list.hasItem() && (done == FALSE) ) {
            if ( (state= pComp( list.getItem().monom, newmonom )) < 0 )
                list++;
            else done= TRUE;
        }
        if ( done == FALSE ) {
            nlist.append( fglmDelem( newmonom, v, k ) );
            break;
        }
        if ( state == 0 ) {
            list.getItem().newDivisor();
            pDelete1( & newmonom );
        }
        else {
            list.insert( fglmDelem( newmonom, v, k ) );
        }
        k--;
    }
    while ( --k >= 1 ) {
        newmonom= pCopy( m );
        pGetExp( newmonom, k )++;
        pSetm( newmonom );
        nlist.append( fglmDelem( newmonom, v, k ) );
    }
}

BOOLEAN
fglmDdata::reduce( fglmVector v )
{
    return gauss.reduce( v );
}
```

Wenn eine lineare Abhängigkeit von `gauss` festgestellt worden ist, so wird in dieser Routine das entsprechende Basispolynom erstellt. Hier wird nochmals gekürzt und auf eine für Singular normierte Form gebracht.

```
void
fglmDdata::newGroebnerPoly( poly & m )
{
    fglmVector p = gauss.getDependence();
    int k;
    poly result = m;
    poly temp = result;
    m= NULL;
    if ( nGetChar() > 0 ) {
        number lead = nCopy( p.getconstelem( basisSize+1 ) );
        p /= lead;
        nDelete( & lead );
    }
    if ( nGetChar() == 0 ) {
        number gcd= p.gcd();
        if ( ! nIsOne( gcd ) )
            p /= gcd;
        nDelete( & gcd );
    }
    pSetCoeff( result, nCopy( p.getconstelem( basisSize+1 ) ) );
    for ( k= basisSize; k > 0; k-- ) {
        if ( ! nIsZero( p.getconstelem( k ) ) ) {
            temp->next= pCopy( basis[k] );
            pIter( temp );
            pSetCoeff( temp, nCopy( p.getconstelem( k ) ) );
```

```
        }
    }
    pSetm( result );
    if ( ! nGreaterZero( pGetCoeff( result ) ) ) result= pNeg( result );
    if ( groebnerSize == IDELEMS( destId ) ) {
        pEnlargeSet( & destId->m, IDELEMS( destId ), groebnerBS );
        IDELEMS( destId )+= groebnerBS;
    }
    (destId->m)[groebnerSize]= result;
    groebnerSize++;
}
```

Hier kommt nun der FGLM-Algorithmus, der aus den in `l` übergebenen Multiplikationsmatrizen die reduzierte Gröbnerbasis berechnet. Wie auf Seite 51 beschrieben, nimmt er den ersten Durchlauf durch die Schleife vorweg und setzt z.B. die Liste der Kandidaten entsprechend.

```
ideal
GroebnerViaFunctionals( const idealFunctionals & l )
{
    fglmDdata data( l.dimen() );

    poly one = pOne();
    data.updateCandidates( one, fglmVector(l.dimen(), 1) );
    number nOne = nInit( 1 );
    data.reduce( fglmVector( l.dimen(), 1 ) );
    data.newBasisElem( one );
    STICKYPROT( "." );
    while ( data.candidatesLeft() == TRUE ) {
        fglmDelem candidate = data.nextCandidate();
        if ( candidate.isBasisOrEdge() == TRUE ) {
            fglmVector v = l.multiply( candidate.v, candidate.var );
            if ( data.reduce( v ) == TRUE ) {
                data.newGroebnerPoly( candidate.monom );
                STICKYPROT( "+" );
            }
            else {
                data.updateCandidates( candidate.monom, v );
                data.newBasisElem( candidate.monom );
                STICKYPROT( "." );
            }
        }
        else {
            STICKYPROT( "-" );
            candidate.cleanup();
        }
    }
    STICKYPROT( "\n" );
    return ( data.buildIdeal() );
}
```

Dies ist die Implementierung des Algorithmus `finduni`. Sie berechnet alle univariaten Polynome des durch die Multiplikationsmatrizen angegeben Ideals. Sie verwendet ebenfalls wieder die Klasse `gaussReducer`.

```
ideal
FindUnivariatePolys( const idealFunctionals & l )
{
    fglmVector v;
    fglmVector p;
    ideal destIdeal = idInit( pVariables, 1 );

    int i;
    BOOLEAN isZero;
    for ( i= 1; i <= pVariables; i++ ) {
        gaussReducer gauss( l.dimen() );
        isZero= FALSE;
        v= fglmVector( l.dimen(), 1 );
        while ( isZero == FALSE ) {
            if ( (isZero= gauss.reduce( v )) == TRUE ) {
                p= gauss.getDependence();
                number gcd= p.gcd();
                if ( ! nIsOne( gcd ) ) {
                    p /= gcd;
                }
                nDelete( & gcd );
                int k;
                poly temp = NULL;
                poly result;
                for ( k= p.size(); k > 0; k-- ) {
                    number n = nCopy( p.getconstelem( k ) );
                    if ( ! nIsZero( n ) ) {
                        if ( temp == NULL ) {
                            result= pOne();
                            temp= result;
                        }
                        else {
                            temp->next= pOne();
                            pIter( temp );
                        }
                        pSetCoeff( temp, n );
                        pSetExp( temp, i, k-1 );
                        pSetm( temp );
                    }
                }
                if ( ! nGreaterZero( pGetCoeff( result ) ) )
                    result= pNeg( result );
```

```
                (destIdeal->m)[i-1]= result;
            }
            else {
                gauss.store();
                v= l.multiply( v, i );
            }
        }
    }
    return destIdeal;
}

BOOLEAN
fglmzero( idhdl sourceRingHdl, ideal & sourceIdeal,
          idhdl destRingHdl, ideal & destIdeal,
          BOOLEAN switchBack, BOOLEAN deleteIdeal )
{
    idhdl initialRingHdl = currRingHdl;
    BOOLEAN fglmok;

    if ( currRingHdl != sourceRingHdl )
        rSetHdl( sourceRingHdl, TRUE );
    idealFunctionals L( 100, pVariables );
    fglmok = CalculateFunctionals( sourceIdeal, L );
    if ( deleteIdeal == TRUE )
        idDelete( & sourceIdeal );
    rSetHdl( destRingHdl, TRUE );
    if ( fglmok == TRUE ) {
        L.map( IDRING( sourceRingHdl ) );
        destIdeal= GroebnerViaFunctionals( L );
    }
    if ( (switchBack == TRUE) && (currRingHdl != initialRingHdl) )
        rSetHdl( initialRingHdl, TRUE );
    return fglmok;
}

ideal
FindUnivariateWrapper( ideal source )
{
    ideal destIdeal;
    BOOLEAN fglmok;

    idealFunctionals L( 100, pVariables );
    fglmok = CalculateFunctionals( source, L );
    if ( fglmok == TRUE ) {
        destIdeal= FindUnivariatePolys( L );
    }
    return destIdeal;
}
```

fglmhom.cc

Der homogene Algorithmus gliedert sich in drei Abschnitte: Alle erforderlichen Monome eines bestimmten Grades finden, von diesen die Normalform berechnen und unter den so erhaltenen Vektoren die linearen Abhängigkeiten aufspüren. Genauso sind auch die Funktionen aufgeteilt.

```
struct doublepoly
{
    poly sm;
    poly dm;
};

class homogElem
{
public:
    doublepoly mon;
    fglmVector v;
    fglmVector dv;
    int basis;
    int destbasis;
    BOOLEAN inDest;
    homogElem() : v(), dv(), basis(0), destbasis(0), inDest(FALSE) {}
    homogElem( poly m, int b, BOOLEAN ind ) :
        basis(b), inDest(ind)
    {
        mon.dm= m;
        mon.sm= NULL;
    }
};

struct homogData
{
    ideal sourceIdeal;
    doublepoly * sourceHeads;
    int numSourceHeads;
    ideal destIdeal;
    int numDestPolys;
    homogElem * monlist;
    int monlistmax;
    int monlistblock;
    int numMonoms;
    int basisSize;
    int overall;
    int numberofdestbasismonoms;
};
```

Über **hfglmNextdegree** wird festgestellt, in welchem Grad nach den nächsten Basispolynomen gesucht werden muss.

```
int
hfglmNextdegree( intvec * source, ideal current, int & deg )
{
    int numelems;
    intvec * newhilb = hHstdSeries( current, NULL, currQuotient );

    loop {
        if ( deg < newhilb->length() ) {
            if ( deg < source->length() )
                numelems= (*newhilb)[deg]-(*source)[deg];
            else
                numelems= (*newhilb)[deg];
        }
        else {
            if ( deg < source->length() )
                numelems= -(*source)[deg];
            else {
                deg= 0;
                return 0;
            }
        }
        if ( numelems != 0 )
            return numelems;
        deg++;
    }
    delete newhilb;
}
```

Die Funktion **generateMonoms** erzeugt alle Monome eines gewissen Grades. Dabei werden gleich solche aussortiert, die sich schon in dem bis hier gefundenen Ideal befinden.

```
void
generateMonoms( poly m, int var, int deg, homogData * dat )
{
    if ( var == pVariables ) {
        BOOLEAN inSource = FALSE;
        BOOLEAN inDest = FALSE;
        poly mon = pCopy( m );
        pSetExp( mon, var, deg );
        pSetm( mon );
        ++dat->overall;
        int i;
        for ( i= dat->numSourceHeads - 1;
              (i >= 0) && (inSource==FALSE); i-- ) {
            if ( pDivisibleBy( dat->sourceHeads[i].dm, mon ) ) {
                inSource= TRUE;
            }
        }
        for ( i= dat->numDestPolys - 1; (i >= 0) && (inDest==FALSE); i-- ) {
            if ( pDivisibleBy( (dat->destIdeal->m)[i], mon ) ) {
                inDest= TRUE;
            }
        }
        if ( (!inSource) || (!inDest) ) {
            int basis = 0;
            if ( !inSource )
                basis= ++(dat->basisSize);
            if ( !inDest )
                ++dat->numberofdestbasismonoms;
            if ( dat->numMonoms == dat->monlistmax ) {
                dat->monlist= (homogElem * )
                                ReAlloc( dat->monlist,
                                         (dat->monlistmax)*sizeof( homogElem ),
                                         (dat->monlistmax+dat->monlistblock)
                                           * sizeof( homogElem ) );
                int k;
                for ( k= dat->monlistmax;
                      k < (dat->monlistmax+dat->monlistblock); k++ )
                    dat->monlist[k].homogElem();
                dat->monlistmax+= dat->monlistblock;
            }
            dat->monlist[dat->numMonoms]= homogElem( mon, basis, inDest );
            dat->numMonoms++;
            if ( inSource && ! inDest ) PROT( "\\" );
            if ( ! inSource && inDest ) PROT( "/" );
            if ( ! inSource && ! inDest ) PROT( "." );
        }
        else {
            pDelete( & mon );
        }
        return;
    }
    else {
        poly newm = pCopy( m );
        while ( deg >= 0 ) {
            generateMonoms( newm, var+1, deg, dat );
            pGetExp( newm, var )++;
            pSetm( newm );
            deg--;
        }
        pDelete( & newm );
    }
    return;
```

```
}

void
mapMonoms( ring oldRing, homogData & dat )
{
    int * vperm = (int *)Alloc( (currRing->N + 1)*sizeof(int) );
    maFindPerm( oldRing->names, oldRing->N, NULL, 0,
                currRing->names, currRing->N, NULL, 0, vperm, NULL );
    nSetMap( oldRing->ch, oldRing->parameter, oldRing->P, oldRing->minpoly );
    int s;
    for ( s= dat.numMonoms - 1; s >= 0; s-- ) {
        dat.monlist[s].mon.sm= pPermPoly( dat.monlist[s].mon.dm,
                                          vperm, currRing->N, NULL, 0 );
    }
}
```

Hier wird nun die Normalform der einzelnen Monome berechnet und gleich in ihre Vektorraumdarstellung transformiert.

```
void
getVectorRep( homogData & dat )
{
    int s;
    for ( s= 0;  s < dat.numMonoms; s++ ) {
        if ( dat.monlist[s].inDest == FALSE ) {
            fglmVector v;
            if ( dat.monlist[s].basis == 0 ) {
                v= fglmVector( dat.basisSize );
                poly nf = kNF( dat.sourceIdeal, NULL, dat.monlist[s].mon.sm );
                poly temp = nf;
                while (temp != NULL ) {
                    int t;
                    for ( t= dat.numMonoms - 1; t >= 0; t-- ) {
                        if ( dat.monlist[t].basis > 0 ) {
                            if ( pEqual( dat.monlist[t].mon.sm, temp ) ) {
                                number coeff= nCopy( pGetCoeff( temp ) );
                                v.setelem( dat.monlist[t].basis, coeff );
                            }
                        }
                    }
                    temp= pIter(temp);
                }
                pDelete( & nf );
            }
            else {
                v= fglmVector( dat.basisSize, dat.monlist[s].basis );
            }
            dat.monlist[s].v= v;
        }
    }
}

void
remapVectors( ring oldring, homogData & dat )
{
    nSetMap( oldring->ch, oldring->parameter, oldring->P, oldring->minpoly );
    int s;
    for ( s= dat.numMonoms - 1; s >= 0; s-- ) {
        if ( dat.monlist[s].inDest == FALSE ) {
            int k;
            fglmVector newv( dat.basisSize );
            for ( k= dat.basisSize; k > 0; k-- ){
                number newnum= nMap( dat.monlist[s].v.getelem( k ) );
                newv.setelem( k, newnum );
            }
            dat.monlist[s].dv= newv;
        }
    }
}
```

Und schließlich werden in `gaussreduce` die linearen Abhängigkeiten gesucht. Hier wird wieder die Klasse `gaussreducer` verwendet.

```
void
gaussreduce( homogData & dat, int maxnum, int BS )
{
    int s;
    int found= 0;

    int destbasisSize = 0;
    gaussReducer gauss( dat.basisSize );

    for ( s= 0; (s < dat.numMonoms) && (found < maxnum); s++ ) {
        if ( dat.monlist[s].inDest == FALSE ) {
            if ( gauss.reduce( dat.monlist[s].dv ) == FALSE ) {
                destbasisSize++;
                dat.monlist[s].destbasis= destbasisSize;
                gauss.store();
            }
            else {
                fglmVector p= gauss.getDependence();
                poly result = pCopy( dat.monlist[s].mon.dm );
                pSetCoeff( result, nCopy( p.getconstelem( p.size() ) ) );
                int l = 0;
                int k;
                for ( k= 1; k < p.size(); k++ ) {
                    if ( ! p.elemIsZero( k ) ) {
                        while ( dat.monlist[l].destbasis != k )
```

```
                    l++;
                    poly temp = pCopy( dat.monlist[l].mon.dm );
                    pSetCoeff( temp, nCopy( p.getconstelem( k ) ) );
                    result= pAdd( result, temp );
                }
            }
            if ( ! nGreaterZero( pGetCoeff( result ) ) )
                result= pNeg( result );
            PROT( "+" );
            (dat.destIdeal->m)[dat.numDestPolys]= result;
            dat.numDestPolys++;
            if ( IDELEMS(dat.destIdeal) == dat.numDestPolys ) {
                pEnlargeSet( & dat.destIdeal->m,
                IDELEMS( dat.destIdeal ), BS );
                IDELEMS( dat.destIdeal )+= BS;
            }
        }
    }
  }
}
```

Die Routine **fglmhomog** entpricht, wie der Name schon verrät, der Erweiterung des FGLM-Algorithmus auf homogene Ideale. Nachdem die Hilbertfunktion des Eingabeideals berechnet worden ist, werden in einer Schleife gradweise die Basispolynome gefunden.

```
BOOLEAN
fglmhomog( idhdl sourceRingHdl, ideal sourceIdeal,
           idhdl destRingHdl, ideal & destIdeal )
{
#define groebnerBS 16
    int numGBelems;
    int deg = 0;

    homogData dat;

    rSetHdl( sourceRingHdl, TRUE );
    ring sourceRing = currRing;

    intvec * hilb = hHstdSeries( sourceIdeal, NULL, currQuotient );
    int s;
    dat.sourceIdeal= sourceIdeal;
    dat.sourceHeads= (doublepoly *)
    Alloc( IDELEMS( sourceIdeal ) * sizeof( doublepoly ) );
    for ( s= IDELEMS( sourceIdeal ) - 1; s >= 0; s-- ) {
        dat.sourceHeads[s].sm= pHead( (sourceIdeal->m)[s] );
    }
    dat.numSourceHeads= IDELEMS( sourceIdeal );
    rSetHdl( destRingHdl, TRUE );
    ring destRing = currRing;

    int * vperm = (int *)Alloc( (sourceRing->N + 1)*sizeof(int) );
    maFindPerm( sourceRing->names, sourceRing->N, NULL, 0,
                currRing->names, currRing->N, NULL, 0, vperm, NULL );
    nSetMap( sourceRing->ch, sourceRing->parameter,
             sourceRing->P, sourceRing->minpoly );
    for ( s= IDELEMS( sourceIdeal ) - 1; s >= 0; s-- ) {
        dat.sourceHeads[s].dm= pPermPoly( dat.sourceHeads[s].sm,
                                          vperm, sourceRing->N, NULL, 0 );
    }

    dat.destIdeal= idInit( groebnerBS, 1 );
    dat.numDestPolys= 0;

    while ( (numGBelems= hfglmNextdegree( hilb, dat.destIdeal, deg )) != 0 ) {
        int num = 0;
        dat.monlistblock= 512;
        dat.monlistmax= dat.monlistblock;
        dat.monlist= (homogElem *)Alloc( dat.monlistmax*sizeof( homogElem ) );
        int j;
        for ( j= dat.monlistmax - 1; j >= 0; j-- ) dat.monlist[j].homogElem();
        dat.numMonoms= 0;
        dat.basisSize= 0;
        dat.overall= 0;
        dat.numberofdestbasismonoms= 0;

        poly start= pOne();
        generateMonoms( start, 1, deg, &dat );
        pDelete( & start );

        rSetHdl( sourceRingHdl, TRUE );
        mapMonoms( destRing, dat );
        getVectorRep( dat );

        rSetHdl( destRingHdl, TRUE );
        remapVectors( sourceRing, dat );

        gaussreduce( dat, numGBelems, groebnerBS );

        Free( (ADDRESS)dat.monlist, dat.monlistmax*sizeof( homogElem ) );
    }
    destIdeal= dat.destIdeal;
    idSkipZeroes( destIdeal );
    return TRUE;
}

ideal
```

```
fglmhomProc(leftv first, leftv second)
{
    idhdl dest= currRingHdl;
    ideal result;
    rSetHdl( (idhdl)first->data, TRUE );
    idhdl ih= currRing->idroot->get( second->name, myynest );
    ASSERT( ih!=NULL, "Error: Can't find ideal in ring");
    rSetHdl( dest, TRUE );

    ideal i= IDIDEAL(ih);
    fglmhomog( (idhdl)first->data, i, dest, result );

    return( result );
}
```

fglmcomb.cc

Zu guter Letzt noch der Algorithmus `findcombination`. Wie schon erwähnt, existiert er in zwei Versionen. Einmal wird die gewöhnliche Normalformen-Berechnung verwendet, einmal eine modifizierte Version.

```
void
fglmEliminateMonomials( poly * pptr, fglmVector & v,
                        polyset monomials, int numMonoms )
{
    poly temp = *pptr;
    poly pretemp = NULL;
    int point = 0;
    int state;

    while ( (temp != NULL) && (point < numMonoms) ) {
        state= pComp( temp, monomials[point] );
        if ( state == 0 ) {
            poly todelete;
            if ( pretemp == NULL ) {
                todelete = temp;
                *pptr= pIter( *pptr );
                temp= *pptr;
            }
            else {
                todelete= temp;
                temp= pIter( temp );
                pretemp->next= temp;
            }
            pGetCoeff( todelete )= nNeg( pGetCoeff( todelete ) );
            number newelem = nAdd( pGetCoeff( todelete ),
                                   v.getconstelem( point+1 ) );
                v.setelem( point+1, newelem );
                nDelete( & pGetCoeff( todelete ) );
                pFree1( todelete );
                point++;
            }
            else if ( state < 0 )
                point++;
            else {
                pretemp= temp;
                temp= pIter( temp );
            }
        }
}

BOOLEAN
fglmReductionStep( poly * pptr, ideal source, int * w )
{
    if ( *pptr == NULL ) return FALSE;
    int k;
    int best = 0;
    for ( k= IDELEMS( source ) - 1; k >= 0; k-- ) {
        if ( pDivisibleBy( (source->m)[k], *pptr ) ) {
            if ( best == 0 ) {
                best= k + 1;
            }
            else {
                if ( w[k] < w[best-1] ) {
                    best= k + 1;
                }
            }
        }
    }
    if ( best > 0 ) {
        poly p2 = (source->m)[best-1];
        int i, diff;

        poly m = pOne();
        for ( i= pVariables; i > 0; i-- ) {
            diff= pGetExp( *pptr, i ) - pGetExp( p2, i );
            pSetExp( m, i, diff );
        }
        pSetm( m );
        number n1 = nCopy( pGetCoeff( *pptr ) );
        number n2 = pGetCoeff( p2 );

        p2= pCopy( p2 );
        pDelete1( pptr );
        pDelete1( & p2 );
```

```
        p2= pMult( m, p2 );

        number temp = nDiv( n1, n2 );
        nNormalize( temp );
        nDelete( & n1 );
        n1= temp;
        n1= nNeg( n1 );
        pMultN( p2, n1 );
        nDelete( & n1 );
        *pptr= pAdd( *pptr, p2 );
    }
    return ( (best > 0) );
}

void
fglmReduce( poly * pptr, fglmVector & v, polyset m,
            int numMonoms, ideal source, int * w )
{
    BOOLEAN reduced = FALSE;
    reduced= fglmReductionStep( pptr, source, w );
    while ( reduced == TRUE ) {
        fglmEliminateMonomials( pptr, v, m, numMonoms );
        reduced= fglmReductionStep( pptr, source, w );
    }
    STICKYPROT( "<" );
    poly temp = * pptr;
    if ( temp != NULL ) {
        while ( pNext( temp ) != NULL ) {
            STICKYPROT( ">" );
            reduced= fglmReductionStep( & pNext( temp ), source, w );
            while ( reduced == TRUE ) {
                fglmEliminateMonomials( & pNext( temp ), v, m, numMonoms );
                reduced= fglmReductionStep( & pNext( temp ), source, w );
            }
            if ( pNext( temp ) != NULL ) {
                temp= pIter( temp );
            }
        }
    }
}

poly
fglmNewLinearCombination( ideal source, poly monset )
{
    polyset m = NULL;
    polyset nf = NULL;
    fglmVector * mv = NULL;
    fglmVector * v = NULL;
    polyset basis = NULL;
    int basisSize = 0;
    int basisBS = 16;
    int basisMax = basisBS;

    int * weights = NULL;
    int * lengthes = NULL;
    int * order = NULL;

    int numMonoms = 0;
    int k;

    numMonoms= pLength( monset );
    STICKYPROT2( "%i monoms\n", numMonoms );

    m= (polyset)Alloc( numMonoms * sizeof( poly ) );
    poly temp= monset;
    for ( k= 0; k < numMonoms; k++ ) {
        m[k]= pOne();
        pSetExpV( m[k], temp->exp );
        pSetm( m[k] );
        temp= pIter( temp );
    }

    nf= (polyset)Alloc( numMonoms * sizeof( poly ) );
    mv= (fglmVector *)Alloc( numMonoms * sizeof( fglmVector ) );

    v= (fglmVector *)Alloc( numMonoms * sizeof( fglmVector ) );
    basisMax= basisBS;
    basis= (polyset)Alloc( basisMax * sizeof( poly ) );

    weights= (int *)Alloc( IDELEMS( source ) * sizeof( int ) );
    STICKYPROT( "weights: " );
    for ( k= 0; k < IDELEMS( source ); k++ ) {
        poly temp= (source->m)[k];
        int w = 0;
        while ( temp != NULL ) {
            w+= nSize( pGetCoeff( temp ) );
            temp= pIter( temp );
        }
        weights[k]= w;
        STICKYPROT2( "%i ", w );
    }
    STICKYPROT( "\n" );
    lengthes= (int *)Alloc( numMonoms * sizeof( int ) );
    order= (int *)Alloc( numMonoms * sizeof( int ) );
```

Nachdem die Normalformen auf die spezielle Art berechnet worden sind, werden die für die Vektorraumdarstellung nötigen Monome herausgesucht.

```
for ( k= 0; k < numMonoms; k++ ) {
    STICKYPROT( "#" );
    poly current = pCopy( m[k] );
    fglmVector currV( numMonoms, k+1 );

    fglmReduce( & current, currV, m, numMonoms, source, weights );
    poly temp = current;
    int b;
    while ( temp != NULL ) {
        BOOLEAN found = FALSE;
        for ( b= 0; (b < basisSize) && (found == FALSE); b++ ) {
            if ( pEqual( temp, basis[b] ) ) {
                found= TRUE;
            }
        }
        if ( found == FALSE ) {
            if ( basisSize == basisMax ) {
                basis= (polyset)
                         ReAlloc( basis, basisMax * sizeof( poly ),
                                  (basisMax + basisBS ) * sizeof( poly ) );
                basisMax+= basisBS;
            }
            basis[basisSize]= pOne();
            pSetExpV( basis[basisSize], temp->exp );
            pSetm( basis[basisSize] );
            basisSize++;
        }
        temp= pIter( temp );
    }
    nf[k]= current;
    mv[k].fglmVector( currV );
    STICKYPROT( "\n" );
}
```

Ab hier werden die Vektorraumrepräsentationen gefunden.

```
for ( k= 0; k < numMonoms; k++ ) {
    STICKYPROT( "." );
    v[k].fglmVector( basisSize );
    poly mon= nf[k];
    while ( mon != NULL ) {
        BOOLEAN found = FALSE;
        int b = 0;
        while ( found == FALSE ) {
            if ( pEqual( mon, basis[b] ) ) {
                found= TRUE;
            }
            else {
                b++;
            }
        }
        number coeff = nCopy( pGetCoeff( mon ) );
        v[k].setelem( b+1, coeff );
        mon= pIter( mon );
    }
    pDelete( nf + k );
}
Free( (ADDRESS)nf, numMonoms * sizeof( poly ) );
Free( (ADDRESS)weights, IDELEMS( source ) * sizeof( int ) );

STICKYPROT2( "\nbasis size: %i\n", basisSize );
gaussReducer gauss( basisSize );
BOOLEAN isZero = FALSE;
fglmVector p;

STICKYPROT( "sizes: " );
for ( k= 0; k < numMonoms; k++ ) {
    lengthes[k]= v[k].numNonZeroElems();
    STICKYPROT2( "%i ", lengthes[k] );
}
STICKYPROT( "\n" );
```

Und dies ist schließlich der Teil, in dem auf lineare Abhängigkeit getestet wird.

```
int act = 0;
while ( isZero == FALSE ) {
    int best = 0;
    for ( k= numMonoms - 1; k >= 0; k-- ) {
        if ( lengthes[k] > 0 ) {
            if ( best == 0 ) {
                best= k+1;
            }
            else {
                if ( lengthes[k] < lengthes[best-1] ) {
                    best= k+1;
                }
            }
        }
    }
    lengthes[best-1]= 0;
    order[act]= best-1;
    STICKYPROT2( " (%i) ", best );
    if ( ( isZero= gauss.reduce( v[best-1] )) == TRUE ) {
        p= gauss.getDependence();
    }
```

```
    else {
        STICKYPROT( "+" );
        gauss.store();
        act++;
    }
}
poly result = NULL;
if ( isZero == TRUE ) {
    number gcd = p.gcd();
    if ( ! nIsZero( gcd ) && ! ( nIsOne( gcd ) ) ) {
        p/= gcd;
    }
    nDelete( & gcd );
    fglmVector temp( numMonoms );
    for ( k= 0; k < p.size(); k++ ) {
        if ( ! p.elemIsZero( k+1 ) ) {
            temp+= p.getconstelem( k+1 ) * mv[order[k]];
        }
    }
    number denom = temp.clearDenom();
    nDelete( & denom );
    gcd = temp.gcd();
    if ( ! nIsZero( gcd ) && ! nIsOne( gcd ) ) {
        temp/= gcd;
    }
    nDelete( & gcd );

    poly sum = NULL;
    for ( k= 1; k <= numMonoms; k++ ) {
        if ( ! temp.elemIsZero( k ) ) {
            if ( result == NULL ) {
                result= pCopy( m[k-1] );
                sum= result;
            }
            else {
                sum->next= pCopy( m[k-1] );
                pIter( sum );
            }
            pSetCoeff( sum, nCopy( temp.getconstelem( k ) ) );
        }
    }
    pContent( result );
    if ( ! nGreaterZero( pGetCoeff( result ) ) ) result= pNeg( result );
}
Free( (ADDRESS)lengthes, numMonoms * sizeof( int ) );
Free( (ADDRESS)order, numMonoms * sizeof( int ) );
for ( k= 0; k < numMonoms; k++ )
    v[k].~fglmVector();
    Free( (ADDRESS)v, numMonoms * sizeof( fglmVector ) );
    for ( k= 0; k < basisSize; k++ )
        pDelete( basis + k );
    Free( (ADDRESS)basis, basisMax * sizeof( poly ) );
    for ( k= 0; k < numMonoms; k++ )
        mv[k].~fglmVector();
    Free( (ADDRESS)mv, numMonoms * sizeof( fglmVector ) );

    for ( k= 0; k < numMonoms; k++ )
        pDelete( m + k );
    Free( (ADDRESS)m, numMonoms * sizeof( poly ) );

    STICKYPROT( "\n" );
    return result;
}
```

Zum Schluß folgt noch die Version, die die gewöhnliche Normalformen-Berechnung verwendet. Sie ist vom Aufbau sehr ähnlich zu obiger Funktion, so dass die Kommentare nicht nochmals angegeben sind.

```
poly
fglmLinearCombination( ideal source, poly monset )
{
    int k;
    poly temp;

    polyset m;
    polyset nf;
    fglmVector * v;

    polyset basis;
    int basisSize = 0;
    int basisBS = 16;
    int basisMax;
    int numMonoms = 0;
    temp = monset;
    while ( temp != NULL ) {
        numMonoms++;
        temp= pIter( temp );
    }
    m= (polyset)Alloc( numMonoms * sizeof( poly ) );
    nf= (polyset)Alloc( numMonoms * sizeof( poly ) );
    v= (fglmVector *)Alloc( numMonoms * sizeof( fglmVector ) );
    basisMax= basisBS;
    basis= (polyset)Alloc( basisMax * sizeof( poly ) );

    temp= monset;
    for ( k= 0; k < numMonoms; k++ ) {
```

```
        poly mon= pHead( temp );
        if ( ! nIsOne( pGetCoeff( mon ) ) ) {
            nDelete( & pGetCoeff( mon ) );
            pSetCoeff( mon, nInit( 1 ) );
        }
        STICKYPROT( "(" );
        nf[k]= kNF( source, currRing->qideal, mon );
        STICKYPROT( ")" );

        STICKYPROT( "[" );
        poly sm = nf[k];
        while ( sm != NULL ) {
            BOOLEAN found = FALSE;
            int b;
            for ( b= 0; (b < basisSize) && (found == FALSE); b++ )
                if ( pEqual( sm, basis[b] ) ) found= TRUE;
            if ( found == FALSE ) {
                if ( basisSize == basisMax ) {
                    basis= (polyset)
                             ReAlloc( basis, basisMax * sizeof( poly ),
                                      (basisMax + basisBS ) * sizeof( poly ) );
                    basisMax+= basisBS;
                }
                basis[basisSize]= pHead( sm );
                nDelete( & pGetCoeff( basis[basisSize] ) );
                pSetCoeff( basis[basisSize], nInit( 1 ) );
                basisSize++;
            }
            sm= pIter( sm );
        }
        STICKYPROT( "]" );
        m[k]= mon;
        temp= pIter( temp );
    }
    STICKYPROT2( "(%i)", basisSize );
    for ( k= 0; k < numMonoms; k++ ) {
        v[k].fglmVector( basisSize );
        STICKYPROT( "(+" );
        poly mon= nf[k];
        while ( mon != NULL ) {
            BOOLEAN found = FALSE;
            int b= 0;
            while ( found == FALSE ) {
                if ( pEqual( mon, basis[b] ) )
                    found= TRUE;
                else
                    b++;
            }
            number coeff = nCopy( pGetCoeff( mon ) );
            v[k].setelem( b+1, coeff );
            mon= pIter( mon );
        }
        STICKYPROT( ")" );
    }
    gaussReducer gauss( basisSize );
    BOOLEAN isZero = FALSE;
    fglmVector p;
    for ( k= 0; (k < numMonoms) && (isZero == FALSE); k++ ) {
        STICKYPROT( "(-" );
        if ( ( isZero= gauss.reduce( v[k] )) == TRUE )
            p= gauss.getDependence();
        else
            gauss.store();
        STICKYPROT( ")" );
    }
    poly comb = NULL;
    if ( isZero == TRUE ) {
        number gcd = p.gcd();
        if ( ! nIsZero( gcd ) && ! ( nIsOne( gcd ) ) )
            p/= gcd;
        nDelete( & gcd );
        for ( k= 1; k <= p.size(); k++ ) {
            if ( ! p.elemIsZero( k ) ) {
                poly temp = pCopy( m[k-1] );
                pSetCoeff( temp, nCopy( p.getconstelem( k ) ) );
                comb= pAdd( comb, temp );
            }
        }
        if ( ! nGreaterZero( pGetCoeff( comb ) ) ) comb= pNeg( comb );
    }

    for ( k= 0; k < numMonoms; k++ ) {
        pDelete( m + k );
        pDelete( nf + k );
    }
    Free( (ADDRESS)m, numMonoms * sizeof( poly ) );
    Free( (ADDRESS)nf, numMonoms * sizeof( poly ) );
    for ( k= numMonoms - 1; k >= 0; k-- ) v[k].~fglmVector();
    Free( (ADDRESS)v, numMonoms * sizeof( fglmVector ) );
    for ( k= 0; k < basisSize; k++ )
        pDelete( basis + k );
    Free( (ADDRESS)basis, basisMax * sizeof( poly ) );
    STICKYPROT( "\n" );
    return comb;
}
```

Nachwort

Nun ist es also geschafft. Der FGLM-Algorithmus ist erfolgreich implementiert. Durch die Realisierung direkt im Kern von SINGULAR konnte eine faire Umgebung geschaffen werden, um einen Vergleich der beiden verschiedenen Ansätze Buchberger- und FGLM-Algorithmus durchzuführen. Wie die Zeitmessungen ergaben, erweist sich der FGLM-Algorithmus durchaus als Alternative zum herkömmlichen Buchberger-Algorithmus, vor allem, wenn es um die Berechnung einer reduzierten Basis geht. Somit steht in SINGULAR nun ein neues Werkzeug zur Verfügung, das zum Berechnen von Gröbnerbasen eingesetzt werden kann.

Das Danksagen

Mein erster und besonderer Dank geht an Prof. Dr. G. Pfister für die interessante Themenstellung, die gute Betreuung und die zahlreichen Anregungen, Tips und Hinweise, nicht zuletzt zu den Algorithmen aus Kapitel 6, deren Ausarbeitung viel Spaß gemacht hat. Ebenso gilt mein Dank Prof. Dr. G. M. Greuel, der mich in ähnlicher Weise unterstützt hat.

Für die seelische und moralische Unterstützung und vor allem für das intensive Korrekturlesen kann ich Christiane Schlemme nicht genug danken. Wer weiß, was ohne sie aus mir geworden wäre.

Michael Niermann hat mich zu Beginn meiner Arbeit sehr kompetent in die FGLM-Welt eingeführt und mir so den Einstieg wesentlich erleichtert. Hans Schönemann danke ich für das geduldige Beantworten meiner SINGULAR-Fragen.

Große Unterstützung, nicht zuletzt in Form eines optimalen Arbeitsklimas, habe ich durch meine Zimmergenossen Rüdiger Stobbe, Joachim Rech und Olaf Bachmann genossen. Ich war immer sehr froh darüber, das Privileg zu haben, in diesem Zimmer sitzen zu dürfen. Olaf möchte ich an dieser Stelle auch nochmals für das Korrekturlesen und die hilfreiche konstruktive Kritik danken. Jens Schmidt danke ich für die Unterstützung vor allem während der letzten hektischen Tage meiner Arbeit und natürlich für `sns.el`.

Alle Mitglieder der Arbeitsgruppe AGAG an dieser Stelle aufzuführen, würde sicherlich den Rahmen sprengen. Dennoch sei hier gesagt, dass ich mich sehr wohl gefühlt habe, und auch ihnen gilt mein Dank.

Und was würde ich ohne Jones machen!

Sämtliche noch in dieser Arbeit existierende Schreibfehler können auf die neue deutsche Rechtschreibung zurückgeführt werden, die hier verwendet wurde.

Tim Wichmann, im November 1997

Literaturverzeichnis

[1] B. Amrhein, O. Gloor und W. Küchlin. On the walk. Preprint, 27. Januar 1997.

[2] Thomas Becker und Volker Weispfenning. *Gröbner Bases*. Springer-Verlag New York, Inc., 1993.

[3] A. M. Bigatti, P. Conti, L. Robbiano und C. Traverso. A Divide and Conquer algorithm for Hilbert-Poincaré series, multiplicity and dimension of monomial ideals. In *Proceedings of AAECC-10*, Nummer 673 in Lecture Notes in Comp.Sci., Seite 76–88. Springer Verlag, 1993.

[4] D. Bini und B. Mourrain. Handbook of polynomial systems. Im Internet zu finden unter `http://www.inria.fr/safir/POL/`.

[5] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Dissertation, Universität Innsbruck, 1965.

[6] M. Caboara, G. De Dominicis und L. Robbiano. Multigraded Hilbert functions and Buchberger algorithm. Editor Y. N. Lakshman, *Proc. ISSAC '96*, Seite 72–78, New York, 1996. ACM Press.

[7] S. Collart, M. Kalkbrenner und D. Mall. Converting bases with the gröbner walk. *Journal of Symbolic Computations*, 1997.

[8] David Cox, John Little und Donal O'Shea. *Ideals, Varieties, and Algorithms*, volume II. Springer-Verlag, New York-Berlin-Heidelberg, 1992.

[9] David Cox, John Little und Donal O'Shea. *Using Algebraic Geometry*. Springer-Verlag, New York-Berlin-Heidelberg, 1997.

[10] J. C. Faugère. GB: State of GB + tutorial. In *Proceedings of the POSSO Workshop on Software*, 1995.

[11] J. C. Faugère, P. Gianni, D. Lazard und T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *Journal of Symbolic Computation*, 1989.

[12] G.-M. Greuel, G. Pfister und H. Schönemann. Singular Reference Manual. In *Reports On Computer Algebra*, Nummer 12. Zentrum für Computeralgebra, Universität Kaiserslautern, Mai 1997.

[13] M. G. Marinari, H. M. Möller und T. Mora. Gröbner bases of ideals given by dual bases. In *Proc. ISAAC 91*, 55-63, 1991. ACM.

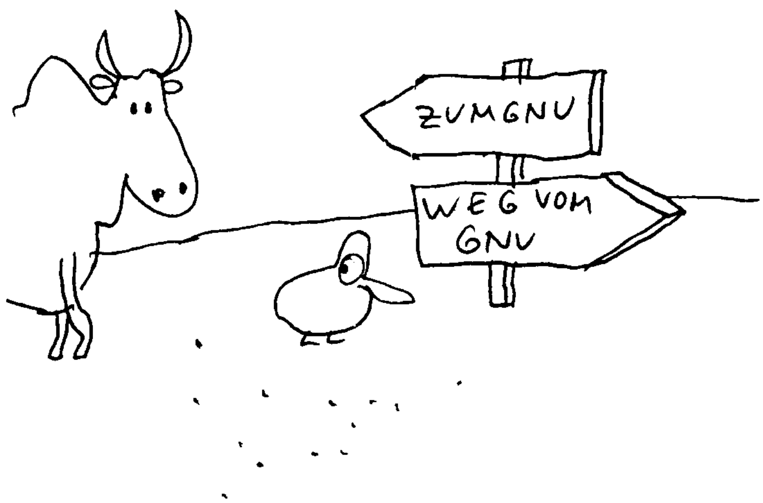
[14] M. G. Marinari, H. M. Möller und T. Mora. Gröber bases of ideals defined by functionals with an application to ideals of projective points. *AAECC*, 1993.

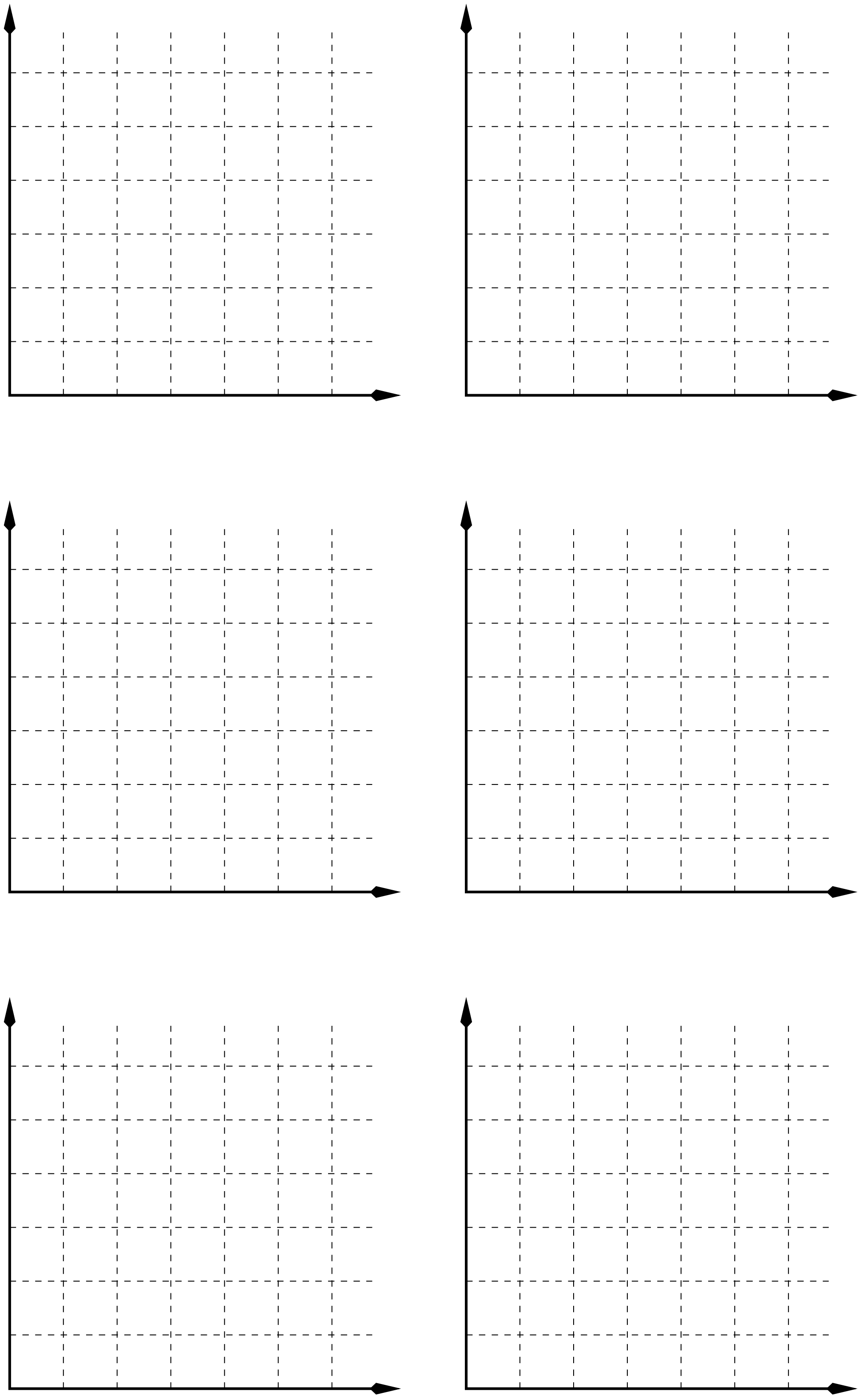
[15] Michael Niermann. Computing gröbner bases of arbitrary ideals by change of term orders. Preprint.

[16] L. Robbiano. Termordering on the polynomial ring. In B. F. Caviness, editor, *EUROCAL 1985 European Conference on Computer Algebra*, volume II, Seite 513–517. Springer LNCS 204, 1985.

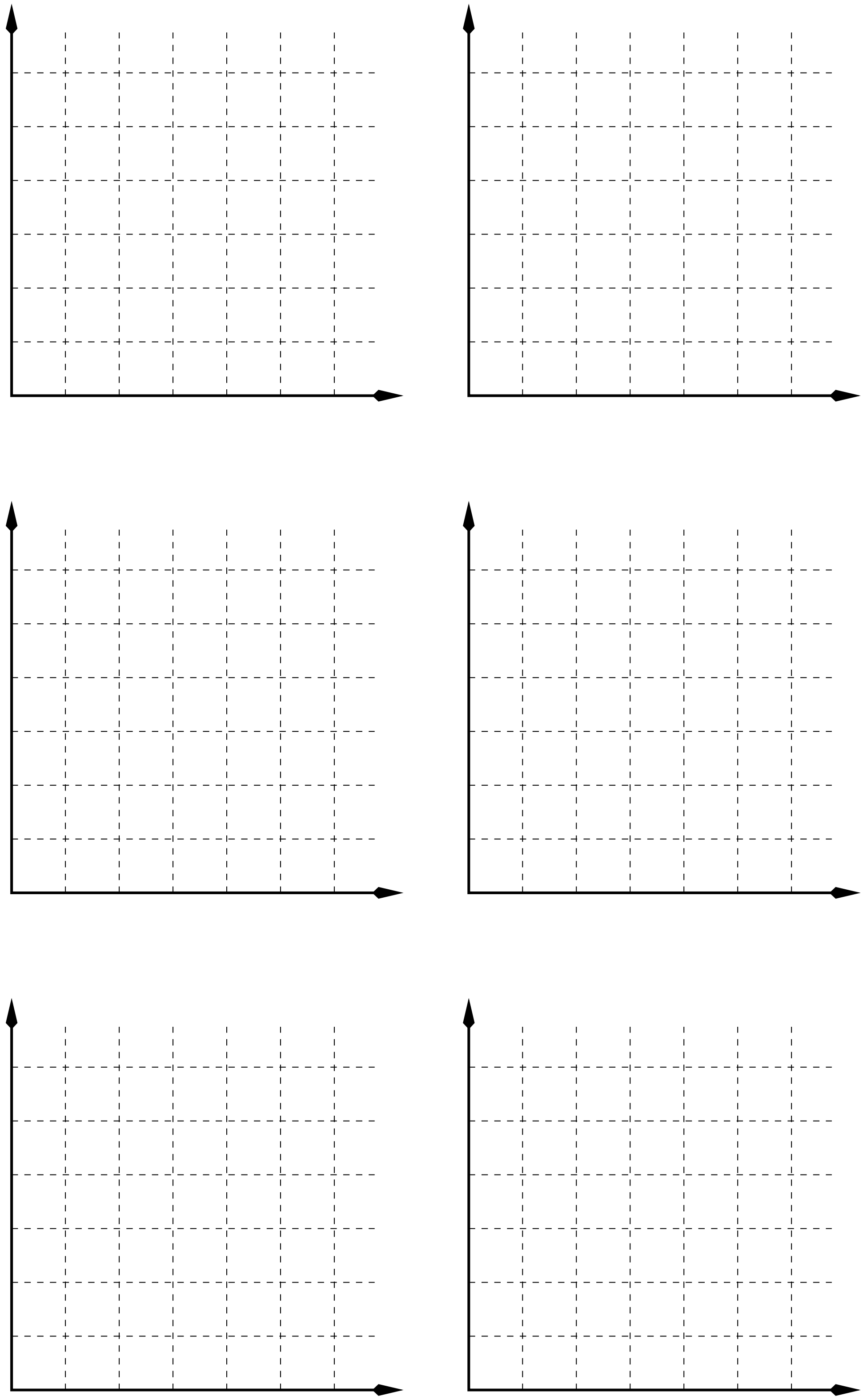
Ferner wurden verwendet:

- Donald E. Knuth, *The TEX-Book*, 14. Auflage, 1986
- Stanley B. Lippman, *C++-Primer*, zweite Auflage, 1991
- Bjarne Stroustrup, *The C++ Programming Language*, zweite Auflage, 1991
- Lothar Fischer, *Max Ernst*
- Tex Rubinowitz, *Auf falbem Laube*
- DUDEN, *Rechtschreibung der deutschen Sprache*, 21. Auflage, Band 1.
- *GNU Emacs Manual*
- *GNU Emacs Lisp Reference Manual*





Blanko Newton-Diagramme zum Heraustrennen oder Kopieren.



Blanko Newton-Diagramme zum Heraustrennen oder Kopieren.