Wiedemann-Algorithmus zur Lösung dünnbesetzter Gleichungssysteme über $\mathbb{F}_p$

Diplomarbeit von Pia Busch

Saarbrücken, Juni 1994

Ich versichere hiermit an Eides statt, daß ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Einzelfall unter Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Saarbrücken, den 30.06.1994

Pia Busch

Mein besonderer Dank gilt Herrn Prof. Buchmann für dieses interessante Thema und Herrn Zayer für die Betreuung.
Weiterhin danke ich Thomas Denny, Thomas Papanikolaou, Andreas Müller u.v.a.m. für ihre zahlreichen Anregungen und Alexander Bach, Olaf Groß, Thomas Sosnowski für ihre Hilfe beim Korrekturlesen dieser Arbeit. Ich danke meinem Freund Jean-Claude Klotzki für seine Geduld in der Endphase meines Studiums, sowie meinen Eltern und den Eltern meines Freundes für ihre Unterstützung während meines Studiums.

Inhaltsverzeichnis

Einleitung

In den letzten Jahren hat die Kryptographie durch die weltweite Vernetzung sicherheitskritischer Rechnernetze (Banken, Militär) stark an Bedeutung gewonnen. Daher rückten Verschlüsselungsverfahren auf der Grundlage der Faktorisierung großer Zahlen in den Blickpunkt des allgemeinen, öffentlichen Interesses. Bekannte Verfahren sind RSA und Diffie-Hellmann-Schema. Sie haben inzwischen in weiteren Bereichen Anwendung gefunden, weil sie eine hohe Datensicherheit bei einfacher, effizienter Implementierung bieten.

Bei verschiedenen Verfahren zur Lösung des Faktorisierungsproblems ist die Lösung linearer Gleichungssystemen ein bedeutendes algorithmisches Hilfsmittel. Gleichungssysteme sind u.a. zu lösen in QS (Quadratic Sieve) und bei der Verwendung des NFS (Number Field Sieve).
Ebenso führen viele mathematische Modelle aus dem Bereich der Natur- und Ingenieurwissenschaften zum Problem der Lösung linearer Gleichungssysteme. Aufgrund dessen wurde die Behandlung dieser Problemstellung eine wichtige Aufgabe der elektronischen Datenverarbeitung. Die Lösung von Gleichungssystemen wurde dadurch für viele Bereiche der technischen Entwicklung und der naturwissenschaftlichen Forschung zu einem Kernproblem, dessen Wichtigkeit durch die Realisierung spezieller Hardwarearchitekturen (Vektorrechner) und die ununterbrochene Folge von Veröffentlichungen unterstrichen wird.

Vor diesem Hintergrund wurden verschiedene effiziente Algorithmen zur Lösung reell-zahliger linearer Gleichungssysteme entwickelt, ebenso Algorithmen die eine besondere Struktur der Gleichungssysteme effizient ausnutzen. (Bandmatrix, Dreieckform).

Neben der Lösung von Gleichungssystemen über $\mathbb{R}$ werden mehr und mehr Verfahren entwickelt, um Gleichungssysteme über endlichen Körpern zu lösen. Zunächst versuchte man die Algorithmen über $\mathbb{R}$ und $\mathbb{C}$ auf endliche Körper zu übertragen. Daran anknüpfend suchte man Verfahren, die eigens über endlichen Körpern entwickelt wurden.

Thema dieser Arbeit ist es nun einen solchen vorzustellen. Er wurde von Douglas H. Wiedemann 1986 in [Wie86] veröffentlicht. Vom Author dieses Papers erhielt er auch seinen Namen, *Wiedemann-Algorithmus*.

Zur Gliederung:

In Kapitel 1 führen wir eine Vielzahl von Begriffen ein, die wir im Verlauf der Arbeit immer wieder benötigen. Sie entstammen weitgehend der linearen Algebra. Der Abschluß diese Kapitels bildet die Beschäftigung mit der Lösbarkeit von Gleichungssystemen.

Kapitel 2 beginnen wir mit der Vorstellung verschiedener Lösungsverfahren. Dann gehen wir auf den Wiedemann-Algorithmus mit mehreren Varianten ein. Hierfür benötigen wir ein Hilfsmittel, den Berlekamp-Massey-Algorithmus. Danach gehen wir zu Beschreibung der eigentlichen Algorithmen über. Wir beginnen mit dem einfachsten Fall, der Lösung linearer Gleichungssysteme mit regulärer Koeffizientenmatrix und erweitern diesen dann für beliebige quadratische Matrizen und Rechteckmatrizen.

In Kapitel 3 beschäftigen wir uns mit den zugrundeliegenden Arithmetiken und ihren Datenstrukturen. Besonderen Wert legen wir dabei auf die Beschreibung der Matrixstruktur.

Abschließen werden wir diese Arbeit mit Kapitel 4. Es enthält unsere Testergebnissen und einem Ausblick auf Themen, die im Zusammenhang mit dieser Arbeit stehen, aber unbehandelt bleiben mußten, da sie über den Rahmen dieser Arbeit hinausgehen.

Kapitel 1

Mathematische Grundlagen

Bevor wir zu den eigentlichen Algorithmen übergehen können, benötigen wir eine Vielzahl von Begriffen, die wir jetzt einführen werden. Definitionen und Sätze sind aus [Rei91] [Rei92] entnommen. Beweise werden wir in diesem Kapitel nicht führen, deshalb verweisen auf die entsprechende Literatur [Lam78] [Hun74] [Lan84].

1.1. Definition (Körper)
Sei K eine Menge, $+, *$ Verknüpfungen auf K, eine algebraische Struktur $(K, +, *)$ heißt Körper, wenn für

- *für $(K, +), (K - \{0\}, *)$ das Assoziativgesetz gilt,*
- *für $(K, +)(K - \{0\} *)$ ein neutrales Element existiert,*
- *für $(K, +)(K - \{0\} *)$ jedes Element ein Inverses besitzt,*
- *für $(K, +)(K - \{0\} *)$ die Verknüpfung kommutativ ist,*
- *die Verknüpfungen $+, *$ distributiv sind*

1.2. Definition (Dimension, Betrag eines Körpers)
Sei K ein Körper. Wir bezeichen mit Dimension oder Betrag von K die Anzahl der Elemente in K.

Speziell in dieser Arbeit verwenden wir den Körper $\mathbb{F}_p$, der isomorph zu $\mathbb{Z}/p\mathbb{Z}$ ist. Die Abbildung, die jedem Element aus $\mathbb{Z}$ einer Restklasse aus $\mathbb{Z}/p\mathbb{Z}$ zuordnet, nennen wir Modulo-Funktion. Sie ist folgendermaßen definiert:

$$\mathrm{mod} : \mathbb{Z} \longrightarrow \mathbb{Z}/p\mathbb{Z} : a \bmod (p) = \{z \in \mathbb{Z} : z = a + l * p, l \in \mathbb{Z}\}$$

Wir werden im Folgenden keinen Unterschied zwischen einem Repräsentaten a und der von a repäsentierten Menge machen.

Addition und Multiplikation sind mittels der entsprechenden Verknüpfungen in $\mathbb{Z}$ definiert. Wir identifizieren den Körper $\mathbb{F}_p$ mit $\mathbb{Z}/p\mathbb{Z}$.

$$+ : \mathbb{F}_p \times \mathbb{F}_p \longrightarrow \mathbb{F}_p : (a, b) \longrightarrow a + b \bmod p \tag{1.1}$$

$$* : \mathbb{F}_p \times \mathbb{F}_p \longrightarrow \mathbb{F}_p : (a, b) \longrightarrow a * b \bmod p \tag{1.2}$$

1.3. Beispiel (Verknüpfungstafel des Körpers $\mathbb{F}_7$)

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

*	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

1.1 Polynomring

1.4. Definition (Polynom)
Sei K ein Körper, seien $a_i \in K$ für $i = 1, \cdots, \infty$ so bezeichnen wir $f(X) = \sum_{i=0}^{\infty} a_i * x^i$ als Polynom in X über K. K heißt Koeffizientenkörper, X Unbestimmte.

Auf der Menge der Polynome läßt sich eine Ordnungsrelation einführen.

1.5. Definition (Grad eines Polynoms)
Sei $f(X)$ Polynom, die größte natürliche Zahl i für die $a_i \neq 0$ heißt grad(f(X)) oder deg(f(x)). Statt Grad sagen wir auch Gradfunktion.

Wir können auf der Menge der Polynome verschiedene Verknüpfungen einführen.

1.6. Definition (Verknüpfungen des Polynomrings)
Sei $K[X]$ die Menge der Polynome über dem Körper K, Polynomring genannt. Seien $f, g \in K[X]$, $f = \sum_{i=0}^{\infty} a_i * x^i$, $g = \sum_{k=0}^{\infty} b_k * x^k$. Sei r=grad(f(X)) und s=grad(g(X)), so sind Polynomaddition und -multiplikation folgendermaßen definiert:

$$+ : K[X] \times K[X] \longrightarrow K[X] : (f, g) \longrightarrow \sum_i (a_i + b_i) * x^i \tag{1.3}$$

$$* : K[X] \times K[X] \longrightarrow K[X] : (f, g) \longrightarrow \sum_i^{r+s} \sum_k^{i} (a_k * b_{i-k}) * x^i \tag{1.4}$$

1.2 Vektorräume

1.7. Definition (Vektorraum, Linearraum)
Sei $(M, +_M)$ eine kommutative Gruppe, sei $(K; +_K, *)$ ein Körper und "$*$" eine Verknüpfung auf M bezüglich K, dann heißt $(M; +, K, *)$ Vektorraum oder Linearraum über K, wenn für alle $a, b \in M$ und $\alpha, \beta \in K$ gilt:

- $\alpha * (a +_M b) = \alpha * a +_M \alpha * b$
- $(\alpha +_K \beta) * a = \alpha * a +_M \beta * a$
- $(\alpha * \beta) * a = \alpha * (\beta * a)$
- $1 * a = a$

Die Elemente dieses Vektorraumes nennen wir Vektoren.

Im Verlauf dieser Arbeit benötigen wir nur der Spezialfall $M = K^n$, $n \in \mathbb{N}$. Daher definieren wir die Verknüpfungen direkt für K^n.

1.8. Definition (Vektoraddition)
Sei $a, b \in K^n$, e_i für $i = 1, \cdots, n$ die Einheitsvektoren des K^n, so definieren wir:

$$+ : K^n \times K^n \longrightarrow K^n : a + b = \sum_{i=0}^{n} a_i * e_i + \sum_{i=0}^{n} b_i * e_i = \sum_{i=0}^{n} (a_i + b_i) * e_i \tag{1.5}$$

Es gibt noch weitere Abbildungen zwischen der Menge der Vektoren und dem Koeffizientenkörper K. Eine wichtige ist die Bildung des Skalarprodukts.

1.9. Definition (Skalarprodukt)
Seien $a, b \in K^n$ so nennen wir

$$<,>: K^n \times K^n \longrightarrow K :< a, b > = < \sum_{i=0}^{n} a_i * e_i, \sum_{i=0}^{n} b_i * e_i > = \sum_{i=0}^{n} (a_i * b_i) \quad (1.6)$$

Skalarprodukt von a und b.

1.3 lineare Abbildungen

1.10. Definition (Linearkombination)
Sei $T \subseteq K^n, T \neq \emptyset$. Sei $f = \sum_{x \in T} a_x * x$, $a_x \in K$ bis auf endliche viele $a_x = 0$. x durchlaufe ganz T, so nennen wir f eine Linearkombination von Elementen aus T.

1.11. Satz
Die Menge aller Linearkombinationen aus Elementen von T bilden ein Unterraum des Vektorraumes K^n.

1.12. Definition (Erzeugendensystem)
Sei K^n ein K-Vektorraum, dann heißt eine Teilmenge E ein Erzeugendensystem von K^n, wenn die Menge der Linearkombinationen $\{\sum a_x * x, x \in E, a_x \in K\} = K^n$ ist. Zum Erzeugnis der Menge E sagen wir auch Span(E).

1.13. Definition (linear abhängig)
Sei K^n ein K-Vektorraum. Eine Teilmenge T von K^n heißt linear unabhängig über K, wenn aus $\sum_{a_x} a_x * x = 0$ stets $a_x = 0$ für alle a_x folgt. Sonst heißt T linear abhängig über K.
Speziell: $\emptyset$ ist per Definition linear unabhängig.

1.14. Definition (Basis)
Sei K^n ein K-Vektorraum. Ein linear unabhängiges Erzeugendensystem von K^n heißt Basis von K^n.

1.15. Definition (Dimension eines Vektorraumes)
Sei V ein K-Vektorraum, $E \subseteq V$ Basis, so heißt die Ordung der Basis E Dimension des Vektorraumes V. Wir schreiben kurz $\dim(V/K)$.

1.16. Definition (lineare Abbildung)
Seien V, W K-Vektorräume, dann heißt $f : V \longrightarrow W$ Vektorraumhomomorphismus oder lineare Abbildung, wenn für alle $a, b \in V$ und $\alpha \in K$ gilt:

$$f(a + b) = f(a) + f(b) \tag{1.7}$$

$$f(\alpha * a) = \alpha * f(a) \tag{1.8}$$

1.17. Bemerkung
Wir nennen eine Verknüpfung linear, wenn sie den Eigenschaften der linearen Abbildung genügt.

Da jeder Vektorraum eine Basis besitzt, sind lineare Abbildungen bereits eindeutig bestimmt durch die Bildelemente der Basis. Für endliche Vektorräume über K gibt es aufgrund dieser Eigenschaft eine enge Beziehung zwischen linearer Abbildung und Matrizen.

Betrachten wir zwei endlich dimensionale Vektorräume V, W über K mit den Dimensionen n b.z.w. m, und geben wir als Basen $B = \{b_1, \cdots, b_n\}$ von V und $C = \{c_1, \cdots, c_m\}$ von W vor, so existiert zu n nicht notwendigerweise verschiedene Elemente $x_1, \cdots, x_n$ aus W genau eine Abbildung $f : V \longrightarrow W$ mit $f(b_i) = x_i$ $(i = 1, \cdots, n)$. Da die x_i bezüglich C eine eindeutige Darstellung $x_i = \sum_{k=1}^{m} a_{ik} * c_k$ besitzen, ergibt sich

$$f(b_i) = \sum_{k=1}^{m} a_{ik} * c_k \quad (i \in \{1, \cdots, n\}).$$

Bezüglich der Basen B, C ist jede lineare Abbildung bereits eindeutig bestimmt durch ein System nicht notwendigerweise verschiedener Körperelemente α_{ik} $(i \in \{1, \cdots, n\}, k \in \{1, \cdots, m\})$. Dieses System bezeichnen wir als Matrix.

1.18. Definition (Matrix)
Sei K ein Körper und α_{ik} $(i \in \{1, \cdots, n\}, k \in \{1, \cdots, m\})$ seien nicht notwendigerweise verschiedene Elemente aus K, dann heißt

$$(\alpha_{ik}) = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1m} \\ \vdots & & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nm} \end{pmatrix}$$

Matrix über K vom Type(n,m) oder kurz $n \times m$-Matrix über K.
i heißt Zeilenindex, k Spaltenindex. Die Anzahl der Spalten ist m, die Anzahl der Zeilen ist n.

Da auch zu jeder $n \times m$-Matrix eine lineare Abbildung $f : V \longrightarrow W$ existiert, ergibt sich

1.19. Satz
Sind V, W endlich dimensionale K-Vektorräume, so läßt sich bezüglich der Basen B und C jeder linearen Abbildung $f : V \longrightarrow W$ bijektiv eine $n \times m$-Matrix über K zuordnen.

Bezeichen wir die Menge aller linearen Abbildungen von V nach W mit $L(V, W)$ und die Menge aller $n \times m$-Matrizen mit $M_{n,m}(K)$ so gibt es eine bijektive Abbildung $l : L(V, W) \longrightarrow M_{n,m}(K)$. Auf $L(V, W)$ läßt sich durch

$$f + g : V \longrightarrow W : x \mapsto f(x) + g(x) \tag{1.9}$$

$$\alpha * f : V \longrightarrow W : x \mapsto \alpha * f(x) \tag{1.10}$$

eine Vektoraumstruktur erklären. Da sich den linearen Abbildungen $f + g$ und $\alpha * f$ Matrizen zuordnen lassen, definieren wir

1.20. Definition (Matrixaddition)
Sei $A = (\alpha_{ik})$ und $B = (\beta_{ik})$ $i = 1, \cdots, n,\ k = 1, \cdots, m$

$$A + B : M_{n,m}(K) \times M_{n,m}(K) \longrightarrow M_{n,m}(K) : (\alpha_{ik}) + (\beta_{ik}) = (\alpha_{ik} + \beta_{ik}) \tag{1.11}$$

heißt Matrixaddition

und

1.21. Definition (Skalarmultiplikation)
Sei $A = (\alpha_{ik})$, $i = 1, \cdots, n,\ k = 1, \cdots, m$ eine $n \times m$ Matrix über K. $\alpha \in K$

$$\alpha * A : K \times M_{n,m}(K) \longrightarrow M_{n,m}(K) : \alpha * (\alpha_{ik}) \mapsto (\alpha * \alpha_{ik}) \tag{1.12}$$

nennen wir Multiplikation mit einen Körperelement oder Skalarmultiplikation.

Aus der Hintereinanderausführung von linearen Abbildungen läßt sich die Matrixmultiplikation ableiten.

1.22. Definition (Matrixmultiplikation)
Seien $f : V_1 \longrightarrow V_2, g : V_2 \longrightarrow V_3$ lineare Abbildungen, so ist die Hintereinanderausführung $g \circ f : V_1 \longrightarrow V_3$ eine lineare Abbildung. Geben wir bezüglich der Basen B_1, B_2, B_3 die entsprechenden Matrizen an, so entspricht $f(x) = (\alpha_{ik}) * x$, $g(x) = (\beta_{kl} * x)$ und $g \circ f(x) = (\alpha_{ik}) * (\beta_{kl}) * x = \sum_{k=1}^{m} \alpha_{ik}\beta_{kl} * x$. Diese Verknüpfung zweier Matrizen nennen wir Matrixmultiplikation.

Die Multiplikation von Vektor und Matrix läßt sich auf die Matrixmultiplikation zurückführen, indem wir den Vektor v als $m \times 1$ b.z.w. $1 \times n$-Matrix auffassen, je nachdem ob wir von rechts oder links den Vektor an eine $m \times n$ Matrix multiplizieren.

Betrachten wir nun die Zeilen oder Spalten einer Matrix als Vektoren.

1.23. Definition (Rang einer Matrix)
Sei $A \in M_{n,m}(K)$, seien $a_1, \cdots, a_n$ die Zeilen von A als Vektoren, dann ist die Anzahl der linear unabhängigen Vektoren gleich dem Rang von A, kurz rang(A).

Ist rang(A)=min(n,m), so hat A Maximalrang. Ist dabei $n = m$, so sagen wir auch A ist regulär, sonst heißt A singulär.

Wir benötigen noch den Begriff der Determinante einer Matrix. Es ist nur sinnvoll sie für quadratische Matrizen zu definieren.

Zur Definition benötigen wir den Begriff *Signum einer Permutation.*
Läßt sich in einer Permutation $P = \begin{pmatrix} 1 & \cdots & n \\ i_1 & \cdots & i_n \end{pmatrix}$ die Reihenfolge (1 ...n) durch eine Anzahl k von Vertauschungen je zweier benachbarter Elemente in die Reihenfolge $(i_1 \cdots i_n)$ überführen, so heißt $(-1)^k$ Signum von P oder kurz sign(P).

1.24. Definition (transponierte Matrix)
Sei A ein $n \times m$-Matrix. Sei A^T eine $m \times n$-Matrix, die durch

$$trans : M_{n,m}(K) \longrightarrow M_{m,n}(K) : z_i \mapsto s_i \tag{1.13}$$

aus A hervorgeht, so heißt A^T transponierte Matrix zu A. z_i bezeichne die i-te Zeile und s_i die i-te Spalte von A.

1.25. Definition (Determinante eine Matrix)
Sei $A = (\alpha_{ik}) \in K^{n,n}$, dann heißt

$$\begin{vmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{vmatrix} = \sum sign(P) * \alpha_{i_1 1} * \alpha_{i_2 2} * \cdots * \alpha_{i_n n}$$

mit $P = \begin{pmatrix} 1 & \cdots & n \\ i_1 & \cdots & i_n \end{pmatrix}$

n-reihige Determinante (Det.) von A, kurz det(A).

Wir geben noch einige Eigenschaften der Determinanten an.

Die Determinante der Matrix A ist ungleich Null, wenn der Rang der Matrix maximal ist, d.h. die Matrix regulär ist.
Die Determinante von A^T ist gleich der Determinante von A.

1.26. Satz (Dimensionsformel für Vektorräume)
Seien V, W endlich dimensionale K-Vektorräume, $V \subseteq W$ *dann gilt:*

$$dim(W) = dim(V) * dim(W/V).$$

W/V bezeichen wir als Restklassenraum. Er entsteht durch den Homomorphismus $f : W \longrightarrow W/V$*, der alle Erzeugnisse der Vektoren aus V auf Null abbildet. Diese Abbildung nennen wir kanonischer Homomorphismus von W in seinen Restklassenraum W/V.*

1.4 Lösbarkeit von Gleichungssystemen

1.27. Definition (Lineares Gleichungssystem)
Sei K ein Körper, a_{ij} *und* b_i *aus K für* $i = 1, \ldots, m$, $j = 1, \ldots, n$
Ein System von m Gleichungen und n Unbestimmten $x_1, \ldots, x_n$

$$\begin{array}{ccccccccc} a_{11} * x_1 & + & a_{12} * x_2 & + & \ldots & + & a_{1n} * x_n & = & b_1 \\ a_{21} * x_1 & + & a_{22} * x_2 & + & \ldots & + & a_{2n} * x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & \vdots & \\ a_{m1} * x_1 & + & a_{m2} * x_2 & + & \ldots & + & a_{mn} * x_n & = & b_m \end{array} \qquad (*)$$

heißt lineares Gleichungssystem.

1.28. Bemerkung
Zur Vereinfachung der Schreibweise werden wir im Folgenden die Matrizenschreibweise verwenden. Für obiges Gleichungssystem schreiben wir kurz

$$A * x = b$$

A ist eine $m \times n$-Matrix mit den Koeffizienten a_{ij}, b ein Vektor aus K^m und x der Vektor der Unbestimmten aus K^n.

Wir werden uns nun mit der Lösbarkeit dieser Systeme beschäftigen. Wann ist $A * x = b$ lösbar, wieviele Lösungen gibt es? Die nachfolgenden Sätze geben darüber Auskunft.

1.29. Satz
Sei $A \in K^{n,m}$, $x \in K^n$, $b \in K^m$ dann gilt:
Das Gleichungssystem $A * x = b$ ist genau dann lösbar, wenn b linear abhängig von $a_1, \ldots, a_m$, den Spaltenvektoren von A, ist, d.h. es gibt ein $a_i \in K$, $a_i \neq 0$, sodaß $b = \sum_{i=1}^{m} c_i * a_i$.

1.30. Satz
Sei $A \in K^{m,n}$, $x \in K^n$ und $b \in K^m$ dann gilt:

1. *Die Lösungmenge des linearen Gleichungssystems $A*x = b$ ist ein Vektorraum der Dimension n - r. (r der Rang A)*
2. *Das Gleichungssystem $A * x = b$ hat genau eine eindeutig bestimmte Lösung, wenn der $rang(A) = r = n = m$ ist.*

Fassen wir kurz zusammen:

- für reguläre $n \times n$-Matrizen gibt es immer eine Lösung. Sie ist eindeutig bestimmt.
- für beliebige $n \times m$-Matrizen kann es keine oder mehrere Lösungen geben. Liegt b im Vektorraum, der durch die Spaltenvektoren von A erzeugt wird, so gibt es mindestens eine Lösung, sonst keine. Die Anzahl der Lösungen ist abhängig von Rang der Matrix. Sei $r = rang(A)$. Ist $n < m$, so gibt es $m - r$ Lösungen, ist $m > n$, so gibt es $n - r$ Lösungen.

Kapitel 2

Der Wiedemann-Algorithmus

Douglas H. Wiedemann stellt in seinem Artikel [Wie86] verschiedene Algorithmen zur Lösung linearer Gleichungssysteme vor. Sie sind speziell über endlichen Körpern $\mathbb{F}_p$ entwickelt worden. Zur Lösung des Gleichungssystems $A * x = b$ berechnet er das Minimalpolynom $f_A(x)$, dann ist $\sum_{i=1}^{d} f_i * A^{i-1} * b$, $\text{grad}(f) = d$, eine Lösung. Die f_i bezeichnen die Koeffizienten von f.

Wir werden uns zunächst mit einigen Lösungsansätzen beschäftigen und dann zur Beschreibung der Algorithmen übergehen. Wir beschreiben dazu zuerst den Berlekamp-Massey-Algorithmus. Er wird in jeder Variante des Wiedemann-Algorithmus benötigt. Danach erst gehen wir auf die eigentlichen Algorithmen ein. Wir beginnen mit der Version für reguläre Matrizen, beschreiben einen deterministischen sowie einen probabilistischen Ansatz, und erweitern diese Schritt für Schritt für allgemeine quadratische Matrizen und Rechteckmatrizen.

2.1 Lösungsansätze

Wir betrachten zunächst Laufzeit- und Platzverhalten des Gaussalgorithmus.
Sei $K = \mathbb{Z}/p\mathbb{Z}$, p Primzahl, seien $A \in K^{n,m}$ $n, m \in \mathbb{N}$, $b \in K^n$, sei w die Anzahl der Einträge ungleich Null der Matrix A, so liefert der Gaussalgorithmus folgende Ergebnisse:

1. Laufzeit
   Die Laufzeit der Gausselimination ist mit $O(n * m * \max(n, m))$ nur von der Größe der Matrix abhängig. Insbesondere ist sie unabhängig von der Anzahl w der Einträge ungleich Null der Matrix A.

2. Speicherplatz
   Bei der Lösung von Gleichungssystemen benötigt die Gausselimination immer $O(n * m)$ Speichereinheiten. Ist die Matrix dünnbesetzt liefert dies keine Verbesserung in der Speicherplatzkomplexität.

Diese Analyse verdeutlicht, das Verfahren benötigt werden, die die Dünnbesetztheit der Gleichungssysteme ausnutzt, um bessere Laufzeit und Speicherplatzabschätzungen zu erhalten. Hier setzt Wiedemann mit seinem Verfahren an.

Das Verfahren beruht auf der Idee, das Problem der Lösung eines linearen Gleichungssystems auf das Problem, der Bestimmung eines Minimalpolynoms zur Matrix A zurückzuführen. Dieses Verfahren ist sinnvoll, da es seit mehren Jahren ein effizientes Verfahren zur Bestimmung eines Minimalpolynoms existiert [Mas69]. Wir führen zunächst den Begriff des Minimalpolynoms ein.

2.1. Definition (Minimalpolynom)
Sei $A \in K^{n,n}$, $f \in K[X]$, $d = \mathrm{grad}(f)$
f^A *heißt Minimalpolynom zu* A*, wenn gilt:*

1. $f^A(A) = f_0 + f_1 * A + f_2 * A^2 + \cdots + f_d * A^d = 0$
2. f^A *ist Polynom kleinsten positiven Grades mit obiger Eigenschaft*

Nun möchten wir auf die Grundidee des Wiedemann-Algorithmus zurückkommen.

Wiedemann schlägt vor das Minimalpolynom $f^A(x)$ zu berechnen und daraus eine Lösung durch die Formel $-\frac{1}{f_0} * \sum_{i=1}^{d} f_i * A^{i-1} * b$ zu berechnen.

Jetzt etwas formaler:

2.2. Satz
Sei $A \in K^{n,n}, x, b \in K^n$*, K Körper, sei f das Minimalpolynom zur Matrix A, so ist*

$$-\frac{1}{f_0} * \sum_{i=1}^{d} f_i * A^{i-1} * b \tag{2.1}$$

eine Lösung zu $A * x = b$.

Beweis:

$$\begin{aligned}
A * x &= A * (-\tfrac{1}{f_0} * \textstyle\sum_{i=1}^{n} f_i * A^{i-1} * b) \\
&= -\tfrac{1}{f_0} * \textstyle\sum_{i=1}^{n} f_i * A^{i} * b \\
&= -\tfrac{1}{f_0} * (\textstyle\sum_{i=0}^{n} f_i * A^{i} * b - f_0 * A^0 * b) && (A^0 = I) \\
&= -\tfrac{1}{f_0} * (\quad 0 \quad - f_0) * b && (f \textit{ ist Minimalpolynom von } A) \\
&= b
\end{aligned}$$

■

Wir wissen jetzt, wie das gesuchte Polynom aussehen soll, gibt es jedoch ein Polynom, das obige Eigenschaft erfüllt.

2.3. Satz
Sei $A \in K^{n,n}$ beliebig, so ist die Menge der Polynome die

$$f(A) = \sum_{i=0}^{d} f_i * A^i = 0$$

erfüllen, nicht leer.

Beweis:
Wir führen den Beweis auf die Definition des Charakteristischen Polynoms zur Matrix A zurück.

2.4. Definition (Charakteristisches Polynom)
Sei $A \in K^{n,n}$. Das charakteristische Polynom zur Matrix A ist das Polynom $f_A(X) = det(X * I_n - A)$ über K. Es hat Grad n.

Wie man leicht sieht ist $f_A(A) = det(A * I_n - A) = det(O_n) = 0$ ■

Aufgrund der Gradfunktion auf dem Polynomring existiert ein Polynom kleinsten Grades, das diese Eigenschaft erfüllt.
Normieren wir das Polynom durch Multiplikation eines invertierbaren Elements (Einheit) des Polynomringes, sodaß der Koeffizient $f_0 = 1$, so ist das Polynom eindeutig bestimmt.

2.2 Der Berlekamp-Massey Algorithmus zur Berechnung des Minimalpolynoms

Im vorangehenden Abschnitt haben wir gesehen, daß es für jede beliebige Matrix A ein Minimalpolynom f existiert mit $f(A) = 0$, jetzt möchten wir einen Algorithmus vorstellen der uns dieses Polynom berechnet.

Wie wir schon gesehen haben, benötigen wir Polynome f die folgende Eigenschaft haben:

$$u * f(A) * b = 0$$

Sei $f = \sum_{i=0}^{\mathrm{grad}(f)} f_i * X^i$, so gilt:

$$\begin{aligned} u * f(A) * b &= u * (f_0 * b + f_1 * A * b + \cdots + f_{\mathrm{grad}(f)} * A^{\mathrm{grad}(f)} * b) \\ &= f_0 * \underbrace{u * b}_{s_0} + f_1 * \underbrace{u * A * b}_{s_1} + \cdots + f_{\mathrm{grad}(f)} * \underbrace{u * A^{\mathrm{grad}(f)} * b}_{s_{\mathrm{grad}(f)}}) \end{aligned}$$

Bezeichenen wir nun die Glieder $u * A^i * b$ mit s_i für alle $i = 1, \cdots, \infty$, so können wir das Problem Polynome f mit obiger Eigenschaft zu finden, auf das folgende zurückführen.

2.5. Problem
Sei eine Folge $S = \{(s_i)\}_{i=0}^{2n-1}$ von Körperelementen gegeben, suche ein Polynom f kleinsten positiven Grades das S nach Vorschrift

$$s_j = -\sum_{i=1}^{L} c_i * s_{j-i} \textit{ für } j = n, n+1, \ldots \tag{2.2}$$

erzeugt.

Zum besseren Verständnis des Algorithmus, beschäftigen wir uns mit einigen Eigenschaften von Polynomen.

Polynome von Grad n sind durch 2n Körperelemente eindeutig bestimmt. Diese Aussage wird in nachfolgendem Satz zum Ausdruck gebracht.

2.6. Satz
Sei $S = \{(s_i)\}_{i=0}^{2n-1}$. Sei $f \in \mathbb{F}_p[x]$ ein Connectionpolynom, das S erzeugt, so ist f eindeutig bestimmt.

Beweis: durch Widerspruch.
Sei $f \neq g$ ein Connectionpolynom, das S erzeugt.

1. $\mathrm{grad}(f) < \mathrm{grad}(g)$ führt zum Widerspruch zur Minimalität von g

2. $\mathrm{grad}(f) > \mathrm{grad}(g)$ führt zum Widerspruch zur Minimalität von f

3. $\mathrm{grad}(f) = \mathrm{grad}(g) = L$

$$\begin{aligned} s_j = -\textstyle\sum_{i=1}^{L} g_i * s_{j-i} &= -\textstyle\sum_{i=1}^{L} f_i * s_{j-i} \ \forall j = L, \cdots, 2n-1 \\ &\Longleftrightarrow \textstyle\sum_{i=1}^{L}(g_i * s_{j-i} - f_i * s_{j-i}) = 0 \\ &\Longleftrightarrow \textstyle\sum_{i=1}^{L}(g_i - f_i) * s_{j-i} = 0 \end{aligned}$$

Diese Summe entspricht einem Gleichungssystem der Form

$$\begin{pmatrix} g_1 - f_1 & 0 & \cdots & 0 \\ 0 & g_2 - f_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & g_L - f_L \end{pmatrix} * \begin{pmatrix} s_{j-1} \\ s_{j-2} \\ \vdots \\ s_{j-L} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Da im Allgemeinen $s_{j-i} \neq 0$, sehen wir direkt, die Matrixgleichungen sind nur erfüllt, wenn $g_i - f_i = 0$. Daraus folgt $g_i = f_i \; \forall i = 1, \cdots, L$. Also ist $f = g$. Dies widerspricht der Annahme und die Aussage des Satzes ist bewiesen.

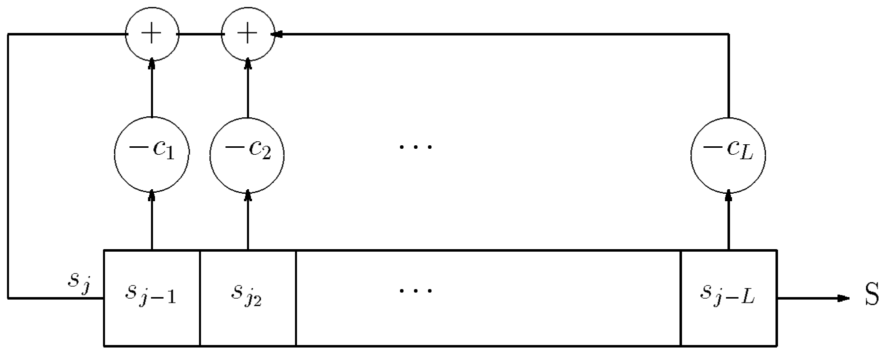
■

Kommen wir nun auf unser ursprüngliches Problem zurück. J.L Massey liefert uns in seinem Artikel [Mas69] eine Lösung. Er beschreibt den Algorithmus mittels Shift-Registern. Da diese Darstellungsweise sehr anschaulich ist, werden wir sie auch verwenden.

2.7. Definition (Lineares Feedback-Shift-Register)
Sei K ein Körper, $c_i \in K$, $i = 1, \ldots, n$ Koeffizienten eines Polynoms C. Sei S eine Folge in der die einzelnen Folgenglieder s_j durch folgende Vorschrift hervorgehen,

$$s_j = -\sum_{i=1}^{L} c_i * s_{j-i} \text{ für } j = n, n+1, \ldots$$

Ein Register, das diese Folge erzeugt, nennen wir lineares Feedback-Shift-Register oder kurz LFSR der Länge L.
Die Initialzustände dieses Registers sind die Koeffizienten c_i des Polynoms C. Wir nennen dieses Polynom Connection- oder Verknüpfungspolynom.
Speziell ist das Connectionpolynom das Polynom kleinsten positiven Grades, das die Folge S erzeugt.



Allgemeines lineares Feedback–Shift–Register mit L Zuständen

Sei $-c1, \cdots, -c_L$ eine feste Initalisierung des LFSR, so erzeugen wir aus den Folgengliedern $s_{j-L}, \cdots, s_{j-1}$ ein weiteres Folgenglied s_j, indem wir die Folgenglieder s_{j-k} mit c_k multiplizieren ($k = 1, \cdots, L$) und die entstandenen Produkte aufaddieren. Dann shiften wir den Inhalt des Shiftregisters der Folge um eine Stelle nach rechts, um die gleiche Ausgangsbasis für die Bestimmung eines weiteren Folgengliedes zu erhalten.

Wir führen eine Sprechweise ein:

2.8. Definition
Wir sagen, ein LFSR erzeugt (generiert) eine endliche Folge $s_0, \ldots, s_{n-1}$, wenn diese Folge mit den ersten n Ausgaben des LFSR bei fester Initialisierung übereinstimmt.

Um ein Polynom g kleinsten positiven Grades zu erzeugen, beginnen wir mit einem Polynom nullten Grades. Der Einfachkeit halber wählen wir $g = 1$, da 1 nur sich selbst als Teiler besitzt.

Der nächste Satz liefert uns eine Aussage über die Gradänderung des Connectionpolynoms.

2.9. Satz
Erzeugt ein LFSR der Länge L eine Folge $S = (s_0, \ldots, s_{N-1})$ aber nicht eine Folge $S^{'} = (s_0, \ldots, s_N)$, dann hat jedes LFSR, das $S^{'}$ erzeugt die Länge $L^{'} \geq N + 1 - L$.

Beweis:
Für $L > N$ ist die Behauptung erfüllt, da die initellen Zustände gleich den Folgengliedern sind.
Betrachten wir nun den Fall $L \leq N$:
Seinen F_1, F_2 zwei LFSR.

$$F_1 \text{ erzeugt } s_j = -\sum_{i=1}^{L} c_i * s_{j-i} \text{ für } j = L, \ldots, N-1 \text{ aber nicht } s_N \tag{2.3}$$

$$F_2 \text{ erzeugt } s_j = -\sum_{k=1}^{L^{'}} c_k^{'} * s_{j-k} = s_i \text{ für } j = L^{'}, \ldots, N \tag{2.4}$$

Annahme: $L^{'} \leq N - L$

$$\begin{aligned} -\textstyle\sum_{i=1}^{L} c_i * s_{N-i} &= +\textstyle\sum_{i=1}^{L} c_i * \sum_{k=1}^{L^{'}} c_k^{'} * s_{N-i-k} \\ &= +\textstyle\sum_{k=1}^{L^{'}} c_k^{'} * \sum_{i=1}^{L} c_i * s_{N-k-i} \\ &= -\textstyle\sum_{k=1}^{L^{'}} c_{k'} * s_{N-k} \\ &= s_N \end{aligned}$$

Diese Aussage widerspricht (2.3). Somit ist die Behauptung gezeigt. ■

Insgesamt liefert uns das folgenden

2.10. Algorithmus [Berlekamp-Massey]

Berechnet zur Folge S der Länge 2n ein Polynom p minimaler Länge, das die Folge S generiert

Eingabe: Folge S, n
Ausgabe: Polynom C

(1)	$x = 1; N = 0; C = 1;$	Initialisierung
(2)	$B = 1; L = 0; b = 1;$	
(3)	**while** $(N \neq n)$ **do**	
(4)	$\quad d = S_N + \sum_{i=1}^{L} C_i * S_{N-i}$	Differenz zur Folge S
(5)	$\quad$ **if** $(d == 0)$ **then**	Kein Fehler
(6)	$\qquad x+ = 1;$	Graddifferenz C – B
(7)	$\quad$ **fi**	
(8)	$\quad$ **if** $(d \neq 0;$ und $2 * L > N)$ **then**	C nicht eindeutig
(9)	$\qquad C(D) = C(D) - d * b^{-1} * D^x * B(D);$	
(10)	$\qquad x+ = 1;$	Graddifferenz C – B
(11)	$\quad$ **fi**	
(12)	$\quad$ **if** $(d \neq 0;$ und $2 * L <= N)$ **then**	C eindeutig
(13)	$\qquad T(D) = C(D);$	Sichere altes C
(14)	$\qquad C(D) = C(D) - d * b^{-1} * D^x * B(D);$	neues C
(15)	$\qquad L = N + 1 - L;$	Länge von C
(16)	$\qquad B(D) = T(D);$	
(17)	$\qquad b = d;$	
(18)	$\qquad x = 1;$	
(19)	$\quad$ **fi**	
(20)	$\quad N+ = 1;$	
(21)	**od**	

In Schritt 1, 2 werden die Connectionpolynome B und C, sowie die Laufvariable N initialisert. Der Abstandzähler x der beiden Polynome bezüglich des Grades wie der noch nicht berechnete Fehler zwischen der Folge S und der Folge erzeugt durch das Connectionpolynom C werden mit 1 vorbesetzt.

Betrachten wir nun die Schleife (3-21):
Wir berechnen zunächst den Fehler d zwischen Folge S und der von C erzeugten nach folgender Vorschrift

$$d = S_N + \sum_{i=1}^{L} C_i * S - N - i.$$

Dies impliziert, d ist die Differenz des tatsächliche Folgengliedes und dem berechneten.

1. Schritt (5-7) $d = 0$,
   Das aktuelle Connectionpolynom C erzeugt auch s_N. Deshalb brauchen wir das Register der Körperelemente nur nach links zu shiften. Wir erhöhen daher x um 1.

2. Schritt (8-11) $d \neq 0$ und $2L > N$,
   so wissen wir nach Satz 2.6, daß C nicht eindeutig bestimmt ist. Wir können daher C durch das Polynom B, das Connectionpolynom vor der letzten Gradänderung, nachbessern ohne die Länge des Registers b.z.w. den Grad des Polynoms zu ändern. Der Abstandszähler x wird um eins erhöht.

3. Schritt (12-19) $d \neq 0$ und $2L \leq N$,
   hier wissen wir nach Satz 2.6, daß eine Längenänderung erfolgen muß. Wir sichern das Connectionpolynom C in T, bessern C mit Hilfe des Polynoms B nach und weisen dem Polynom B T zu. In b speichern wir den Fehler zwischen Folgenglied s_N und dem von B erzeugten. Merke in B steht das Polynom C vor der Änderung. Danach berechnen wir die Länge des neuen Connectionpolynoms C. x wird wieder auf 1 zurückgesetzt.

Es bleibt zu zeigen, daß $C(D) = C(D) - d * b^{-1} * D^x * B(D)$ eine geeignete Wahl für das neue Connectionpolynom ist.

2.11. Satz

Sei eine Folge S gegeben, $C(D) = 1 + c_1^{(N)} * D + \cdots + c_{L_N(S)} * D^{L_N(S)}$ *ist eine Polynom minimaler Länge, das die Folge* $s_0, \cdots, s_{N-1}$ *erzeugt.*

Beweis:

Wir nehmen an, daß Connectionpolynome $c^{(N)}(D)$ für $N = 1, \cdots, n$ gibt, wobei für $N = 1, \cdots, n-1$ $L_{N+1}(S) = max[L_N(S), N + 1 - L_N(S)]$ hält.
Nachzuweisen ist die Gleichheit für n:

$$s_j = \sum_{i=1}^{L_N(S)} c_i^{(N)} * s_{N-i} = \begin{cases} 0 & j = L_N(S), \cdots, n-1 \\ d_n & j = n \end{cases}$$
$$d_n = s_N + \sum_{i=1}^{L_N(S)} c_I^{(N)} * s_{N-i}$$

Ist $d_n = 0$, so erzeugt $C^{(n)}(D)$ die ersten $n+1$ Folgenglieder von S. Somit ist $L_{n+1}(S) = L_n(S)$ und $C^{(n+1)}(D) = C^{(n)}(D)$.
Ist $d_n \neq 0$

$$\text{Sei } L_m(S) < L_n(S) \text{ und } L_{m+1}(S) = L_n(S) \tag{2.5}$$

Beim Übergang von m nach $m+1$ wird eine Längenänderung erforderlich, denn das LFSR mit Connectionpolynom $C^{(m)}(D)$ der Länge $L_m(S)$ kann die Folge $s_0, \cdots, s_{m-1}$ nicht erzeugen. Es gilt:

$$s_j + \sum_{i=1}^{L_m(S)} c_i^{(m)} * s_{j-i} = \begin{cases} 0 & j = L_m(S), \cdots, m-1 \\ d_m \neq 0 & j = m \end{cases} \tag{2.6}$$

Nach Voraussetung gilt:
$L_{m+1}(S) = max[L_m(S), m+1-L_m(S)]$ aus (2.5) folgt dann $L_n(S) = m+1-L_m(S)$

Somit haben wir gezeigt, daß die Länge des veränderten Connectionpolynom korrekt ist. Wir zeigen nun, daß das Polynom auch von der Form $C^{n+1}(D) = C^{(n)}(D) - d_n * d_m^1 * D^{n-m} * C^{(m)}(D)$ ist.

Der Grad von $C(D)$ ist höchstens $L_{n+1}(S) = max[L_n(S), n-m-L_n(S)] = max[L_n(S), n+1-L_n(S)]$
Aus der Darstellung $C^{(n+1)}(D) = C^{(n)}(D) - d_n * d_m^{-1} * D^{n-m} * C^{(m)}(D)$ folgt:

$$\begin{aligned} s_j + \textstyle\sum_{i=1}^{L_n(S)} c_i * s_{j-i} &= s_j + \textstyle\sum_{i=1}^{L_n(S)} c_i^{(n)} * s_{j-i} - d_n * d_m^{-1} \\ &\qquad * [s_{j-n+m} + \textstyle\sum_{i=1}^{L_m(S)} c_i^{(m)} * s_{j-n+m-i}] \\ &= \begin{cases} 0 & j = L, \cdots, n-1 \\ dn - dn * dm^{-1} * d_m = 0 & j = n \end{cases} \end{aligned}$$

■

Für diesen Algorithmus können wir folgende Komplexität angeben.

2.12. Satz
Sei $A \in K^{n,n}$, $u, b \in K^n$, f_u^b das Polynom zu A, sodaß gilt $u * f_u^b(A) * b = 0$, dann gilt:

Der Berlekamp-Massey-Algorithmus benötigt

$O(n^2)$ *Körperoperationen*
$O(n)$ *Speicherplatz*

zur Berechnung von f.

Beweis:
Wir betrachten zunächst die Schleife, sie wird n-mal durchlaufen. Welche Kosten verursachen jetzt die einzelnen Schritte.

1. Die Initialisierung in Schritt (1 - 2) benötigt 6 Körperoperationen, sie wird im Gegensatz zu den weiteren Schritten nur einmal ausgeführt.

2. Die Berechnung der Schleifenbedingung benötigt eine Körperoperation.

3. Zur Berechnung der Folge S in Schritt (4) sind L Multiplikationen sowie Additionen notwendig, d.h. wir benötigen etwa $2 * L \leq 2 * n$ Körperoperationen

4. In Schritt (5-7) werden nur zwei Körperoperationen ausgeführt.

5. Zur Korrektur des Polynoms in Schritt (8 -11) wird ein Shift um x Einheiten eine Skalarmultiplikation sowie eine Polynomaddition ausgeführt. Jede dieser Funktionen benötigt etwa n Körperoperationen, also zusammen $3 * n$.

6. Zusätzlich zu den Opertationen in Schritt 4) wird in Schritt (12 - 19) eine Sicherung des alten Polynoms benötigt, sodaß wir weitere n Operationen für das Kopieren der Polynome benötigen. Somit verlangt dieser Schritt etwa $5 * n$ Körperopertionen.

7. Die Erhöhung der Schleifenvariable in Schritt (20) benötigt eine Körperoperation.

Dies führt zu einer Gesamtlaufzeit von

$$O(6 + n * (2n + 5n)) = O(6 + 7n^2) = O(n^2)$$

Somit ist die Komplexität der Laufzeit gezeigt. Es bleibt noch die Speicherplatzkomplexität zu zeigen.

Speicherplatz benötigen wir für drei Polynome, die Eingabefolge S und die drei Elemente d, x und b, d.h wir benötigen

$$3n + 2n + 3 = 5n + 3 = O(n)$$

Körpereinheiten. ■

Zum Abschluß dieses Abschnittes möchten wir noch ein Beispiel angeben.

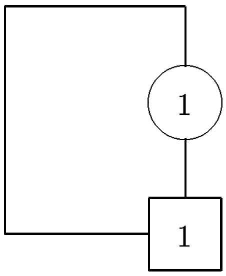
2.13. Beispiel
Wir wollen ein Polynom kleinsten positiven Grades über $\mathbb{F}_7$ berechnen, das die Folge 1 2 3 4 5 1 erzeugt.

B bezeichne entsprechend der Definition in Algorithmus dem Connectionpolynom vor der Längenänderung, sowie C das jeweils aktuelle Connection-Polynom. Der Fehler zur Folge S wird in d abgelegt. Tritt eine Längenänderung ein, so wird d in b gesichert. x gibt an, um wieviel B gegenüber C bei der Korrektur von C geshiftet werden muß. Die Variable N sagt uns, wie oft wir die Schleife schon durchlaufen haben.

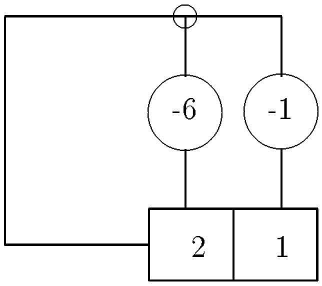
Initialisierung

$C = 1;\ B = 1;\ d = 1;\ b = 1;\ L = 0;\ x = 1;\ N = 0$

Im ersten Schleifendurchlauf erzeugen wie das folgende LFSR.

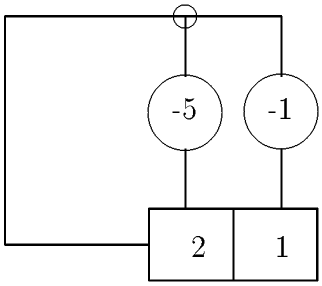


$d = s_0 = 1$
Es erzeugt die Folge 1 1 1 1 1 1 ...
$d \neq 0$ und $2 * L \leq N$, so führen wir den Code von Schritt 12 – 19 aus. Nach Beendigung der Schleife stehen in den Variablen
$C = 1 + 6D;\ B = 1;\ b = 1;\ x = 1;\ L = 1;\ N = 1$
C entsteht dabei durch die Vorschrift $C = C - d * b^{-1} * D * B = 1 - 1 * 1 * D * 1 = 1 * 6D$, die Länge des Shiftregisters b.z.w. der Grad des Connection-Polynoms berechnet sich durch $L = N + 1 - L = 1$.



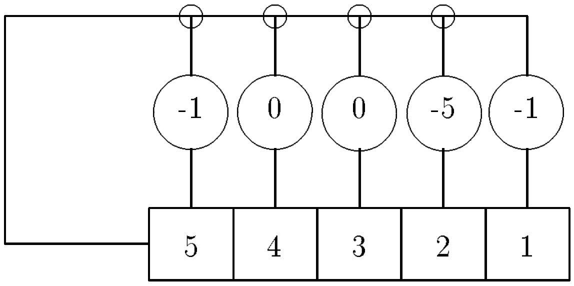
Dieses Bild zeigt den Startzustand des erzeugten Registers. Es erzeugt die Folge 1 2 1 6 5 6 ...

$d = s_1 + c_1 * s_0 = 1$
$d \neq 0$ und $2L > N$ so führen wir den Code von Schritt (8 – 11) aus.
$C = 1 + 6 * D - 1 * 1 * D * 1 = 1 + 5 * D$ nach Vorschrift in Schritt (9)
Nach Beendigung der Schleife ist $B = 1;\ L = 1;\ b = 1;\ x = 2;\ N = 2$.



Dieses Bild zeigt den Startzustand des erzeugten Registers. Es erzeugt die Folge 1 2 3 4 5 6 ...

Wir sehen an der Folge, das C schon auf 5 Stellen korrekt erzeugt wird. Daher wird in den nächsten 2 Durchläufen jeweils der Code in Schritt (5-7) ausgeführt.
Danach gelten folgende Werte:
$C = 1 + 5 * D$; $B = 1$; $L = 1$; $b = 1$; $x = 4$; $N = 4$
Wir berechnen $d = s_4 + c_1 * s_3 = 2$
Da $2L \leq N$, ist führen wir den Code von Schritt $(11 - 19)$ aus.
$C = 1 + 5 * D - 2 * 1 * D^4 * 1 = 1 + 5 * D + 5 * D^4$; $B = 1 + 5 * D$; $L = 4 + 1 - 1 = 4$; $b = 2$; $x = 1$; $N = 5$



Dieses Bild zeigt das entsprechende LFSR das die Folge S= 1 2 3 4 5 1 ... erzeugt.

2.3 Der Wiedemann-Algorithmus für reguläre Matrizen

Bevor wir auf die verschiedenen Varianten dieses Algorithmus eingehen, möchten wir Gemeinsamkeiten beschreiben.

In jeder Variante berechnen wir eine Folge $\{< u, A^i * b >\}$. Sie dient als Eingabe für den Berlekamp-Massey-Algorithmus zur Berechnung des Minimalpolynoms f^A zur Matrix A. u und b sind Vektoren.

Dem Algorithmus liegt folgende Idee zugrunde:
Für das Minimalpolynom f^A zur Matrix A gilt:

$$\begin{aligned} f^A(A) = 0 &\iff f^A(A) * b = 0 \, \forall b \in K^n \\ &\iff u * f^A(A) * b = 0 \, \forall u, b \in K^n \end{aligned}$$

Wir bestimmen daher Polynome f mit der Eigenschaft $u * f(A) * b = 0$, um daraus das kleinste gemeinsamen Vielfache zu bestimmen. Dieses entspricht dann dem Minimalpolynom $f^A(X)$.

2.14. Definition (Operator)
Seien V, W Vektorräume und $g : V \longrightarrow W$ eine lineare Abbildung. Besitzt g bezüglich der Basen B_V, B_W eine Darstellung $g(x) = A * x$, so nennen wir A einen Operator für g.

2.15. Definition (f^b)
Sei $A \in K^{n,n}$, $b \in K^n$, sei $S = Span(b, A * b, A^2 * b, \cdots)$, sei A_S der Operator A eingeschränkt auf S, so bezeichnen wir mit $f^b(z)$ das Minimalpolynom zu A_S. Die Einschränkung besteht darin, daß für Polynome über S die Bedingung $f(A) * b = 0$ erfüllt sein muß statt $f(A) = 0$.

Eine bedeutende Tatsache ist, daß die Elemente von S mit den Elementen eines Ringes $R = K[X]/(f^b(X))$ identifiziert werden können. Dies zeigen wir mit dem nächsten Satz.

2.16. Satz
Sei $A \in K^{n,n}$,$f^b(X) \in K[X]$, $f^b(X)$ das Polynom minimalen positiven Grades mit $f^b(A) = 0$, $S = Span(b, A * b, A^2 * b, \cdots)$ dann gilt: $K[X]/(f^b(X)) \hat{=} S$.

Beweis:
Wir definieren uns eine Abbildung $h : K[X] \longrightarrow S : X^i \mapsto A^i * b$, $i \in \mathbb{N}_0$, $A^0 = I$

Nach dem Homomorphiesatz für Vektorräume ist zu zeigen

1. h ist ein Vektorraumhomomorphismus
2. $kern(h) = f^b(x) * K[X]$
3. $bild(h) = S$

Dann folgt aus der Satz 1.26 $K[X]/f^b(X) * K[X] \cong S$.

1. Seien $\sum_{i=0}^{\infty} a_i * X^i$ und $\sum_{i=0}^{\infty} b_i * X^i$ Polynome aus $K[X]$, $k \in K$, dann gilt:

   $h(\sum_{i=0}^{\infty} a_i * X^i + \sum_{i=0}^{\infty} b_i * X^i) = h(\sum_{i=0}^{\infty}(a_i + b_i) * X^i)$
   $= \sum_{i=0}^{\infty}(a_i + b_i) * A^i * b = \sum_{i=0}^{\infty} a_i * A^i * b + \sum_{i=0}^{\infty} b_i * A^i * b$

   $h(k * \sum_{i=0}^{\infty} a_i * X^i) = h(\sum_{i=0}^{\infty} k * a_i * X^i)$
   $= \sum_{i=0}^{\infty} k * a_i * A^i * b = k * \sum_{i=0}^{\infty} a_i * X^i$

2. $kern(h) = \{z \in K[X] : h(z) = 0\}$

   (a) Sei $r(X)$ ein beliebiges Polynom aus $K[X]$

$$\begin{aligned} h(f^b(X) * r(X)) &= h((f_0 + f_1 * X + \cdots + f_l * X^l) * r(X)) \\ &= (f_0 * b + f_1 * A * b + \cdots + f_l * A * b) * r(A) \\ &= \underbrace{f^b(A) * b}_{0} * r(A) = 0 \end{aligned}$$

   $\Longrightarrow f^b(X) \subseteq kern(k)$

   (b) Sei $g \in kern(h)$ beliebig, $g \notin f^b(X) * K[X]$, $\text{grad}(g(X)) = k$

$$\begin{aligned} h(g(X)) &= h(g_0 + g_1 * X + \cdots + g_k * X^k) \\ &= g_0 * b + g_1 * A * b + \cdots + g_k * A^k * b \\ &= g(A) * b = 0 \end{aligned}$$

   $k < \text{grad}(f^b(X))$ führt zum Widerspruch der Minimalität von $f^b(X)$.

   $k \geq \text{grad}(f^b(X))$: z.z $f^b(X)|g(X)$

$$f^b(X)|g(x) \Longleftrightarrow g(X) = t(X) * f^b(X) + r(X)$$
$$g(A) * b = \underbrace{t(A) * \underbrace{f^b(A)}_{0} * b}_{0} + r(A) * b = 0$$

   Die Gleichung ist nur erfüllbar für $r(A) = 0$.
   Da $\text{grad}(r(X)) < \text{grad}(f^b(X))$ folgt $r(X) = 0$, wegen der Minimalität von $f^b(X)$.

3. z.z. Für jedes $\alpha \in S$ gibt es ein Polynom $g \in K[X]$, sodaß $h(g(X)) = \alpha$

   Sei $\alpha \in S$ beliebig, so gilt $\alpha = \sum_{i=0}^{\infty} \alpha_i * A^i * b$
   Setze $g(X) = \sum_{i=0} \alpha_i * X^i \in K[X]$
   $\Longrightarrow h(g(X)) = h(\sum_{i=0}^{\infty} \alpha_i * X^i) = \sum_{i=0}^{\infty} \alpha_i * A^i * b = \alpha.$

■

Über die Bildung des Skalarprodukt erhalten wir die Folge $\{< u, A^i * b >\}_{i=0}^{2n-1}$. Erzeugen wir eine Folge von Nullen, so bringt uns diese keinen weiteren Faktor für das Minimalpolynom f^A. Dieser Fall tritt ein, wenn $u \in S^{\perp}$ liegt, d.h. u senkrecht auf allen Vektoren $A^i * b$ steht.
Erhalten wir ein Nichtnullfolge, so berechnen wir ein Polynom f_u^b.

2.17. Definition (f_u^b)
Sei $A \in K^{n,n}, b \in K^n, f_u^b$ ist das Polynom kleinsten minimalen Grades, das die Eigenschaft $u * f_u^b(A) * b = 0$ erfüllt.

2.18. Satz

Das Polynom $f_u^b(X)$ kann durch den Algorithmus 2.10 berechnet werden, indem man den Koeffizienten c_i des Connectionpolynoms $C(X)$ den Koeffizienten $f_{\mathrm{grad}(C)-i}$ zuordnet.

Beweis:
Der Berlekamp-Massey-Algorithmus liefert uns zu einer Folge $\{< u, A^i b >\}_{i=0}^{2n-1}$ ein Polynom $C(X)$ nach Vorschrift (2.2). Daher gilt: $-\sum_{i=0}^{\mathrm{grad}(C)} c_i * s_{j-i} = 0$ für $j = \mathrm{grad}(C), \cdots, 2n-1$, $s_k = u * A^k * b$.

Wir wissen $f_u^b(X) = \sum_{i=0} f_i * X^i$ mit der Eigenschaft $u * f_u^b(A) * b = 0$.

$$\begin{aligned} u * f_u^b(A) * b &= u * (\textstyle\sum_{i=0}^{\mathrm{grad}(f)} f_i * A^i) * b \\ &= \textstyle\sum_{i=0}^{\mathrm{grad}(f)} f_i * u * A^i * b \\ &= \textstyle\sum_{i=0}^{\mathrm{grad}(f)} f_i * s_i \end{aligned}$$

Wegen der Minimalität von f_u^b nach Definition 2.17 und der Minimalität von C gilt: $\mathrm{grad}(f) = \mathrm{grad}(C)$. Daher können die Koeffizienten von C eindeutig auf f_u^b abgebildet werden und es gilt $f_i = -c_{\mathrm{grad}(C)-i}$.

■

Betrachten wir verschiedene u, so erhalten wir mehrere f_u^b und können f^b daraus berechnen.

2.19. Satz

Sei $u, b \in K^n$, f_u^b Polynom über K erzeugt durch die Anwendung des Berlekamp-Massey-Algorithmus auf die Folge $/< u, A^i b > /$ mit $\mathrm{grad}(f_u^b) = d$ so gilt:

$$f^b(x) = \mathrm{kgV}(f_u^b(x)) \, \forall u \in K^n$$

Beweis: z.z. $\mathrm{kgV}(f_u^b(A) * b) = 0$

$f_u^b(X)$ ist ein Polynom minimalen positiven Grades, das $u * f_u^b(A) * b$ erfüllt. Der Grad von $f_u^b(X)$ sei d. $f_u^b(X)$ entsteht nach Satz 2.18 aus dem Connectionpolynom C, erzeugt durch den Algorithmus 2.10 bei Eingabe der Folge $S = \{< u, A^i * b >\}_{i=0}^{2n-1}$, dann gilt:

$$\begin{aligned} u * f_u^b(A) * b &= u * (f_0^u * I + f_1^u * A + \cdots + f_d^u * A) * b. \qquad \text{(Satz 2.18)} \\ &= u * (-c_0 * A^d - c_1 * A^{d-1} - \cdots - c_d * A^0) * b \\ &= -c_0 * u * A^d * b - c_1 * u * A^{d-1} * b - \cdots - c_d * I * b \\ &= -c_0 * s_d - c_1 * s_{d-1} - \cdots - c_d * s_0 \\ &= 0. \end{aligned}$$

c_i bezeichne die Koeffizienten des Connectionpolynoms C. $s_i = < u, A^i * b >$ sind die Folgenglieder von S.

Sei $f^b(S) = f_0 + s_d + f_1 * s_{d-1} + \cdots + f_d * s_0$. Nach Definition 2.15 ist $f^b(A) * b = 0$. Wir brauchen nur noch zu zeigen, daß $f_u^b(X) | f^b(X)$.

$\exists\, g(X),\, r(X) \in K[X] : f^b(X) = g(X) * f_u^b(X) + r(X)$
$f^b(A) * b = \underbrace{g(A) * \underbrace{f_u^b(A)}_{0}}_{0} * b + r(A) * b = 0$

Da $\mathrm{grad}(r(X)) < \mathrm{grad}(f_u^b(X))$ und $f_u^b(X)$ minimal folgt $r(X) = 0$. ■

Aus mehreren Polynomen f^b können wir dann das Minimalpolynom f^A berechnen, wie sagt uns der nächste Satz.

2.20. Satz
Sei $u, b \in K^n$, $f^b \in K[X]$ $\mathrm{grad}(f^b) = d$ so gilt:

$$f^A(x) = \mathrm{kgV}(f^b(x))\, \forall b \in K^n$$

Beweis:
z.z. $f^b(x) \mid f^A(x)$

Wir wissen $f^A(A) = 0$, $f^A(A) * b = 0$, $f^b(A) * b = 0$
$f^A(A) = 0$ laut Definition 2.1, daraus folgt $f^A(A) * b = 0$, $f^b(A) * b = 0$ laut Definition 2.15. Da $f^A(A) * b = 0$ und $f^b(A) * b = 0$, müssen die beiden Polynome einen gemeinsamen Faktor besitzen. Wegen der Minimalität von $f^b(x)$ ist dieser Faktor gleich $f^b(x)$. Somit teilt f^b f^A.

■

Diese Ergebnisse sind in der Praxis nicht zu realisieren. Für eine algorithmische Lösung dieses Problems können nur endlich viele Berechnungen durchgeführt werden. Deshalb müssen wir uns auf endliche Zahl von f_u^b und f^b beschränken.

Dies führt nun zu folgenden Problem:

2.21. Problem
Mit welcher Wahrscheinlichkeit ist f^b aus einer endlichen Anzahl von f_u^b zu berechnen?
Mit welcher Wahrscheinlichkeit ist f^A aus einer endlichen Anzahl von f^b zu berechnen?

Die Definitionen und Sätze dieses Abschnitts gehen auf [Wie86] zurück.

2.22. Definition
Sei p *die Ordnung des Körpers, sei* f_u^b, $u \in K^n$ *das Polynom das durch den Berlekamp-Massey-Algorithmus erzeugt wird, sei* f^b *das Minimalpolynom zu* b, d_i *der Grad der irreduziblen Teilpolynome, so liefert*

$$\phi(f^b) = \prod_i (1 - p^{-d_i}) \tag{2.7}$$

ein Maß für die Wahrscheinlichkeit, daß $f^b = f_u^b$.

2.23. Satz
Sei $\mathrm{grad}(f) \leq p + \cdots + p^j$ *für ein* $j \in \mathbb{N}$, *dann ist* $\phi(f^b) \geq \frac{1}{6j}$.

Beweis:
Die Faktorisierung des Kreisteilungspolynoms $z^{p^i} - z$ enthält alle irreduziblem Polynome von Grad i, d.h. es gibt höchstens $\frac{p^i}{i}$ irreduzible Polynome vom Grad i. Sei nun $x_i * \frac{p^i}{i}$ die Anzahl der irreduziblen Polynome vom Grad i, die f teilen, dann gilt unter der Beschränkung $0 \leq x_i \leq 1$

$$\sum_{i=1}^{\infty} x_i * p^i \leq \mathrm{grad}(f) \leq p + \cdots + p^j \tag{2.8}$$

Unter obiger Beschränkung ist p^i wachsend in i und der Koeffizient $\frac{1}{i}$ fallend in i. Wir erhalten dann den maximalen Wert, indem wir die ersten j Variablen auf 1 setzen und die restlichen auf Null.

Sei $p \geq 2$

$$\begin{array}{rcl} \log \phi(f) & = & \sum_i \log(1 - p^{-i}) * x_i \frac{p^i}{i} \\ & \geq & -\sum_i (p^{-i} + p^{-2i}) * x_i \frac{p^i}{i} \\ & \geq & -\sum_i \frac{x_i}{i} - \log 2 \end{array}$$

$\log(\phi(f)) \geq -H_j - \log 2$ und $H_j < 1 + \log(j)$ liefert das Ergebnis.

■

Dieses Ergebnis zeigt, daß nach genügend großer Anzahl verschiedener u zur Berechnung von f_u^b eines gefunden wird, sodaß $f_u^b = f^b$. In der Regel wird f^b früher gefunden, da wir mehrere f_u^b betrachten.

2.24. Definition

Sei p die Ordnung des Körpers, seien $f^b_{u_i}$, $u_i \in K^n$ für $i = 1, \cdots, k$ Polynome die durch den Berlekamp-Massey-Algorithmus erzeugt werden. f^b das Minimalpolynom in Abhängigkeit von b, d_i der Grad der irreduziblen Teilpolynome, so stellt

$$\phi_k(f^{b_1}, \cdots, f^{b_k}) = \prod(1 - p^{-kd_i}) \tag{2.9}$$

ein Maß für die Wahrscheinlichkeit, daß $f^b = \mathrm{kgV}(f^b_{u_1}, \cdots, f^b_{u_k})$.

2.25. Satz

Seien f^b_u für k zufällig gewählte u berechnet, dann ist die Wahrscheinlichkeit, daß

$$f^b = \mathrm{kgV}(f^b_{u_1}, \cdots, f^b_{u_k})$$

größer als $1 - log(\frac{p^{k-1}}{p^{k-1}-1})$.

Beweis:

Wir gehen davon aus, daß $f^b_u(x)$ für k zufällig gewählte u berechnet wurde.
Analog zu Satz Satz 2.23 kann man zeigen, daß die Wahrscheinlichkeit f^b durch die Bildung des kleinsten gemeinsamen Vielfachen über k Polynome f^b_u zu erzeugen, gleich $\phi_k(f^b)$ ist.

Es gilt desweiteren:

$$\begin{aligned}\phi(f) &\geq 1 + \sum p^{-kd} \\ &\geq 1 - (\tfrac{p^1}{1} * p^{-k} - \tfrac{p^2}{2} * p^{-2k} + \cdots) \\ &= 1 - \log(\tfrac{p^{k-1}}{p^{k-1}-1})\end{aligned}$$

■

2.26. Beispiel

Wir geben an, wie wahrscheinlich es ist bei einem Körper der Dimension p beim k-ten Schleifendurchlauf das Minimalpolynom f^A berechnet zu haben.

p=2:		p=7:		p=1373:	
k=2	30 %	k=2	93 %	k=2	99 %
k=3	71 %	k=3	97 %	k=3	99 %
k=4	97 %	k=4	99 %	k=4	99 %

Wie wir sehen ist die Wahrscheinlichkeit unabhängig von der Dimension der Matrix. Sie ändert sich nur mit der Ordnung des Körpers. Wir führen diese Tatsache darauf zurück, daß die Anzahl der irreduziblen Polynome im Verhältnis zur Ordnung des Körpers stark abnimmt.

2.27. Problem
Wie groß ist nun die Wahrscheinlichkeit f^A aus f^b zu erzeugen, wenn b zufällig gewählt wurde?

Der folgende Satz gibt uns darüber Auskunft.

2.28. Satz
Seien $b_1, \cdots, b_k$ unabhängig per Zufall gewählt, die Wahrscheinlichkeit daß

$$\mathrm{kgV}(f^{b_1}, \cdots, f^{b_k}) = f^A$$

ist, ist mindestens

$$\phi(f^A)$$

Beweis:
Sei $A \in K^{n,n}, v \in K^n$ sodaß $f^v = f^A$
Für die Vektoren $b_1, \cdots, b_k$ sei $f_{b_i}^v$ das Polynom minimalen positiven Grades, das die Folge $\{(b_j, A^i * v)\}_{i=0}^{\infty}$ erzeugt. Sei $h = \underset{j}{\mathrm{kgV}}(f_{b_j})$, so ist nach Definition 2.24 $h = f^v = f^A$ mit der Wahrscheinlichkeit $\phi(f^A)$. ■

2.3.1 Der deterministische Ansatz

Wir stellen hier zwei Varianten vor:

1. eine mit Speicherung der Vektoren $v_i = A^i * b$ für $i = 1, \cdots, 2n$
2. eine ohne diese Vorberechnung.

Wir beginnen mit Fall 1)

2.29. Algorithmus [Wiedemann – deterministisch]

Berechnet deterministisch eine Lösung des Gleichungssystems $A * x = b$ nach einer Idee von Douglas H. Wiedemann [Wie86]

EINGABE: A,b
AUSGABE: lg

(1)	**for** $(i = 0, i < 2n; i++;)$ **do**	Vorberechnung der v_i
(2)	$\quad v_i = A^i * b;$	
(3)	**od**	
(4)	$k = 0;$	k Laufvariable
(5)	$g = 1;$	Variable für Minimalpolynom wird mit 1 initialisiert
(6)	**while** $(\text{grad}(g) < n \wedge k < n)$ **do**	
(7)	$\quad u = e_{k+1}$	e_{k+1} k-ter Einheitsvektor
(8)	$\quad$ **for** $(i = 0; i < 2n - \text{grad}(g); i++)$ **do**	seq Folge für Berlekamp-Massey
(9)	$\qquad seq_i = \text{skalarprod}(u, v_i);$	
(10)	$\quad$ **od**	
(11)	$\quad folg = \text{polymul}(g^{-1}, seq);$	berechnet Folge $(u, v_i)_{i=0}^{2n-1}$
(12)	$\quad f = \text{berlekamp}(folg, n - \text{grad}(g));$	berechne Faktor f des Minimalpolynoms g
(13)	$\quad g = f * g;$	Multiplikation der Faktoren
(14)	$\quad k++;$	
(15)	**od**	
(16)	$lg = \text{lösung}(g, v);$	

In Zeile (1-3) berechnen wir die Vektoren $v_i = A^i * b$ für $i = 1, \cdots, 2n-1$. Sie werden nur einmal für das gesamte Programm berechnet.

In Zeile (4,5) initialisieren wir die Laufvariable $k = 0$ und setzen das Polynom $g = 1$

Nun beginnen wir mit der Schleife (6-15)

Aus den Vektoren v_i und dem in Schritt (7) gesetzten Einheitsvektor e_{k+1} berechnen wir die Folge $\{(u_{k+1}, v_i)\}_{i=0}^{2n-1}$. Aus dieser Folge extrahieren wir den Anteil, der uns einen gemeinsamen Faktor f_u^b liefern würde. Dies geschieht in Schritt (11) mittels Polynommultiplikation. Wir erhalten dadurch die Folge $\{(g^{-1} * S)[i]\}_{i=0}^{l-d}$ für $d = \text{grad}(g)$ und $l = l\ddot{a}nge(S)$. Diese Folge dient als Eingabe für den Berlekamp-Massey-Algorithmus in Schritt (12) zur Bestimmung eines Faktors des Minimalpolynoms f^b zur Matrix A. f^b erhalten wir durch Multiplikation der einzelnen Faktoren f_u^b, die während aller Schleifendurchläufe berechnet werden, daher multiplizieren wir in Schritt (13) die Faktoren auf.
Wir erhöhen noch die Schleifenvariable k und wiederholen die Schleife (6-15) bis

$\mathrm{grad}(g) \leq n$ oder alle n Einheitsvektoren betrachtet wurden.
f^A ist nun berechnet, somit können wir jetzt (16) die Lösung berechnen.

2.30. Bemerkung
Sei $A \in K^{n,n}, b \in K^n$, bezeichne w die Anzahl der Einträge ungleich Null in A, so hat der oben beschriebene deterministische Algorithmus nach [Wie86] eine Laufzeit von

$$O(n * (w + n * \log(n) * \log\log(n)))$$

bei einem Speicherplatzbedarf von

$$O(n + w + n^2).$$

Eine Alternative zu diesem Algorithmus bietet der folgende Algorithmus

2.31. Algorithmus [Wiedemann – deterministisch, Alternative] [Wie86]

Berechnet deterministisch eine Lösung des Gleichungssystems
$A * x = b$ nach einer Idee von Douglas H. Wiedemann

EINGABE: A,b
AUSGABE: lg

$k = 0; y_k = 0;$	
while $((b_k \neq 0) \vee (d_k > n))$ **do**	keine Lösung gefunden
$\quad u_{k+1} = e_{k+1}$	e_{k+1} (k+1)-ter Einheitsvektor
$\quad S = \{(u_{k+1}, A^i * b)\}_{i=0}^{2*(n-d_k)-1};$	Berechne Folge S
$\quad f = \mathrm{Berlekamp}(S, n);$	Berechne Faktor f_u^b
$\quad y_{k+1} = y_k + f_{k+1}^-(A) * b_k$	berechne Zwischenlösung
$\quad b_k = b_0 + A * y_{k+1}$	berechne Fehler in b
$\quad d_{k+1} = d_k + \mathrm{grad}(f_{k+1});$	Anpassen des Grades
$\quad k++;$	
od	
$lg = -y_k;$	konvertiere y_k zur Lösung

Dabei bezeichne $f_{k+1}^-(A) = (f_{k+1}(A) - f_{k+1}(0)) * A^{-1}$. Der Wert $f_{k+1}^-(A) * b_k$ kann dabei ohne Invertierung der Matrix A berechnet werden.

$$f_{k+1}^-(A) = \sum_{i=1}^{d} f_i * A^{i-1} * b_k \tag{2.10}$$

Wir stellen schon beim Überblicken des Algorithmus fest, daß er anders aufgebaut ist als der erste. Wir berechnen hier Polynome f_u^b für unterschiedliche u und b. Daher macht es hier auch wenig Sinn die Vektoren v_i über mehrere Schleifen zu speichern.

Wir betrachten jetzt die Schritte einzeln.
In Zeile (1) wird die Laufvariable $k = 0$ initialisiert. In Zeile (2) steht die Abbruchbedingung der Schleife (2-9) $b_k = 0$ oder $d_k > n$. Ist $b_k = 0$ erfüllt terminiert der Algorithmus mit einer Lösung $lg = -y_k$.
Bis wir jedoch diesen Punkt erreichen, muß die Schleife (2-9) mehrmals durchgeführt werden. Wir beginnen mit $b_k = b$ im ersten Durchlauf. Aus b_k, e_{k+1} und der Matrix A berechnen wir die Folge $S = \{< e_{k+1}, A^i * b_k >\}_{i=0}^{2(n-d_k)-1}$, wie beim ersten Algorithmus. Diese dient als Eingabe des Berlekamp-Massey-Algorithmus zu Bestimmung eines Faktors $f_{e_k+1}^{b_k}$. Wir wollen ihn zur Vereinfachung der Schreibweise mit f bezeichen. Aus f, A, und b_k berechnen wir nach Vorschrift (2.10) $f_{k+1}^{-}(A) * b_k$ und danach die Zwischenlösung y_{k+1} in Zeile (5). In Zeile (6) berechnen wir dann den Fehler $b_k = b - A * y_{k+1}$. Wir passen die Variable d_k noch an den aktuellen Grad des Minimalpolynoms f^A an und können zum nächsten Schleifendurchgang gehen. Nach maximal n Schleifendurchläufen terminiert der Algorithmus, da wir A als regulär vorausgesetzt haben und somit b wie auch x von den Vektoren e_i der Einheitsbasis linear abhängig sind.

2.32. Bemerkung
Sei $A \in K^{n,n}, x, b \in K^n$, Sei w die Anzahl der Einträge ungleich Null der Matrix A, so hat der obige Algorithmus eine Laufzeit von

$$O(n * (w * n + n * \log(n) * \log(\log(n))))$$

bei einem Speicherplatzbedarf von

$$O(n + w).$$

Die Laufzeit ist abgeleitet von den Angaben in [Wie86]. Der zusätzliche Faktor n ist bedingt durch die Berechnung der Vektoren $v_i = A^i * b$ für $i = 0, \cdots, 2n - 1$ in jedem Schleifendurchlauf.

Vergleichen wir die beiden Algorithmen.
Wie wir schon gesehen haben ist der erste Algorithmus in der Laufzeit, um eine Potenz in n besser. Dies liegt daran, daß wir im zweiten Algorithmus die Vektoren $A^i * b$, $i = 1, \cdots, 2n$ jeweils neu berechnen müssen. Dahingegen hat er einen Vorteil von einer Potenz in n beim Speicherbedarf.

2.33. Beispiel
Sei n die Dimension der quadratischen Matrix A, so benötigen wir $2n$ Vektoren der Länge n. Diese müssen gespeichert werden. Wir haben speziell für $n = 10.000$ und $n = 100.000$ den externen Speicherplatzbedarf berechnet.

- n=10.000: 782 Mbyte
- n=100.000: 78,12 Gbyte

Für Anwendungen in der Computer Algebra mit Matrizen in dieser Größenordnung ist er daher weniger geeignet.
Ein weiterer Vorteil des zweiten Algorithmus liegt darin, das wir während der Berechnung des Minimalpolynoms f^A schon die Lösung mit iterieren. Die Schleife wird auch nur im schlechtesten Fall k-mal ausgeführt, da der Algorithmus direkt terminiert, wenn das Minimalpolynom schon erzeugt ist. Eine kleine Modefikation des Algorithmus bringt eine weitere Verbesserung der Laufzeit. Wir wählen die Vektoren e_k nicht mehr in fortlaufender Nummerierung sondern setzen k auf den ersten Index in f_k^-, der ungleich Null ist. Dies erspart uns Schleifendurchläufe die keine Faktoren bringen, da der gewählte Eigenvektor senkrecht auf $f^-(A) * b$ steht.(siehe Beispiel 2.35)

2.34. Bemerkung
Wir vermuten, daß der Algorithmus mit dieser Änderung in der Komplexitätsklasse $O(\log(n) * (w * n + n * \log(n) * \log\log(n))$ liegt. Einen Beweis liegt bis jetzt unseres Wissen nicht vor.

Die Vermutung begründet sich auf Meßergebnisse. Siehe dazu Seite 67. Theoretisch lassen sich die Ergebnisse dadurch erklären, daß die Schleife meist weniger als n-mal durchlaufen wird und die Anzahl der zu berechneten Vektoren $A^i * b$ mit jedem Faktor abnimmt.

2.35. Beispiel
Berechnung einer Lösung des Gleichungssystems $A * x = b$ mittels Algorithmus 2.31 über den Körper $\mathbb{F}_p$.

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix} \quad b = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix}$$

1. $k = 0, b_0 = b, d = 0, y_0 = 0$

   $$v_1 = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} \quad v_2 = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} \quad v_3 = \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} \quad v_4 = \begin{pmatrix} 6 \\ 3 \\ 3 \end{pmatrix} \quad v_5 = \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} \quad v_6 = \begin{pmatrix} 4 \\ 3 \\ 1 \end{pmatrix}$$

   Zusammen mit dem Vektor $e_1 = (1\ 0\ 0)$ erhalten wir eine Folge

   $S = 2\ 1\ 0\ 6\ 5\ 4$

   Bei Eingabe von S in den Berlekamp-Massey-Algorithmus erzeugen wir das

Polynom $f_u^b = 1 + 5x + x^2$
daraus berechnen wir

$$f_{k+1}^{-}(A) * b_k = 5 * \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \\ 6 \end{pmatrix}$$

Wir erhalten daraus eine Zwischenlösung

$$y_1 = y_0 + f_{k+1}^{-}(A) * b_0 = \begin{pmatrix} 4 \\ 4 \\ 6 \end{pmatrix}$$

Wir erhalten einen Fehler von

$$b_1 = b + A * y_1 = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix} * \begin{pmatrix} 4 \\ 4 \\ 6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix}$$

und der Grad des Connectionpolynoms ist.

$$d_k = d_k + \text{grad}(f) = 2$$

2. $k = 1, b_1 = (005), d_1 = 2$
   Wir berechnen die Vektoren $v_i = A^i * b_1$

   $$v_1 = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix} \; v_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

   Zusammen mit dem Vektor $e_2 = (0\ 1\ 0)$ erhalten wir eine Nullfolge. Sie liefert uns keinen weiteren Faktor.

3. $k = 2, b_2 = (0\ 0\ 5), d_2 = 2$
   Wir berechnen die Vektoren $v_i = A^i * b_1$

   $$v_1 = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix} \; v_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

   Diese liefern mit $e_3 = (0\ 0\ 1)$ die Folge $S = 5\ 1$ und somit das Polynom $f_u^b = 1 + 2x$

   Wir berechnen $f_u^{b_2}(A) * b_2 = 2 * \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}$

   Daraus berechnen wir dann die Zwischenlösung y_3

$$y_3 = y_2 + f_u^{b_2}(A) * b_2 = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}$$

Wir überprüfen die Abbruchbedingung

$$b_3 = b - A * y_3 = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix} * \begin{pmatrix} 4 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Somit haben wir eine Lösung des Gleichungssystems gefunden und

$$lg = -y_3 = \begin{pmatrix} 3 \\ 3 \\ 5 \end{pmatrix}.$$

2.3.2 Wiedemann – probabilistisch

Die probabilistischen Versionen des Wiedemann-Algorithmus sind ähnlich, denen der deterministischen. Sie verwenden jedoch statt e_k beliebige Vektoren u_k. Dies ist erlaubt, da jedes $u_k \in K^n$ durch die n Einheitsvektoren e_k des K^n dargestellt werden kann.

Die Schleifendurchläufe lassen sich nach Wiedemann im Mittel auf $O(\log(n))$ reduzieren, so erhalten wir eine Laufzeit von

$$O(\log(n) * (w + n * \log(n) * \log\log(n))$$

bei einer Speicherplatzbedarf von

$$O(n + n^2 + w).$$

Für die 2-te Variante ist die Laufzeit

$$O(\log(n) * (w * n + n * \log(n) * \log\log(n)))$$

bei einem Speicherplatzbedarf von

$$O(n + w).$$

Wägen wir beide probabilistischen Versionen gegeneinander ab, so zeigt sich das gleiche Bild, wie bei den deterministischen, für Anwendungen auf Matrizen der Größenordnung ≥ 10000 ist die erste Version wenig geeignet.

Vergleichen wir jeweils die zweite Version der deterministischen und probabilistischen Algorithmen zeigt sich, daß der probabilistische im Mittel schneller ist als der deterministische. Deshalb werden wir uns in der weiteren Beschreibung auf den letzteren beziehen. Die übrigen Varianten können bei entsprechenden Erweiterungen gleichermaßen verwendet werden.

2.4 Erweiterung für beliebige quadratische Matrizen

In Gegensatz zum vorangegangen Abschnitt betrachten wir jetzt beliebige $n \times n$-Matrizen über dem Körper K.

Sei $A \in K^{n,n}$ singulär, so hat das Minimalpolynom der Matrix A $\lambda = 0$ als Nullstelle, d.h. der Algorithmus für reguläre Matrizen schlägt fehl, sobald ein Faktor mit konstantem Koeffizienten 0 auftritt.

Hier setzten wir mit unserer Erweiterung an. Finden wir einen Faktor mit konstantem Koeffizienten 0, so muß der Algorithmus diese anzeigen. Durch eine Modifikation des Algorithmus für reguläre Matrizen werden wir dieser Sache gerecht. Der modifizierte Algorithmus lautet:

2.36. Algorithmus [Wiedemann - Erweiterung]

Berechnet probabilistich eine Lösung des Gleichungssystems $A*x = b$ oder weißt die Singularität der Matrix A nach.

EINGABE: A ,b
AUSGABE: lg,pol,F_b

(1)	$k = 0; y_k = 0; pol = 1;$	
(2)	**while** $(b_k \neq 0)$ **do**	keine Lösung gefunden
(3)	$u_{k+1} = \text{random},(n)$	u_{k+1} wird zufällig gesetzt
(4)	$S = \{(u_{k+1}, A^i * b)\}_{i=0}^{2*(n-d)-1};$	Berechne Folge S
(5)	$f = \text{Berlekamp}(S, n);$	Berechne Faktor f_u^b

if $(f_0 = 0)$ **then**	Fall der konstante Term von $f_0 = 0$
$\quad F_b = f_{k+1}^{-}(A) * b_k;$	Berechne Fehler in b
$\quad y_{k+1} = y_k + F_b$	berechne Zwischenlösung
$\quad$ break;	
fi	
$y_{k+1} = y_k + f_{k+1}^{-}(A) * b_k;$	berechne Zwischenlösung
$b_k = b_0 + A * y_{k+1};$	berechne Fehler in b
$d_{k+1} = d_k + deg(f_{k+1});$	Anpassen des Grades
$k{+}{+};$	
od	
$lg = -y_k;$	konvertiere y_k zur Lösung
$pol = 0;$	

Wir führen zwei weitere Variablen ein. *pol* ist ein Flag, das angibt, ob wir eine Lösung gefunden haben. Ist $pol \neq 0$ ist die Matrix singulär und F_b gibt uns den Fehler in b an. Diese Daten werden benötigt, um gegebenfalls abhängige Zeilen oder Spalten zu berechnen.

Liefert obiger Algorithmus $pol = 0$, so war die Matrix A regulär und in lg steht die Lösung des Systems. Ist $pol \neq 0$ muß mittels F_b eine linear abhängige Spalte berechnet werden. Ist $F_b = 0$, so ist A zwar singulär, wir haben jedoch eine Lösung für $A * x = b$ gefunden. Nur für den Fall $F_b \neq 0$ ist der Algorithmus weiterzuführen.

2.37. Problem
Wie berechnen wir nun eine linear abhängige Spalte?

Wir bestimmen zunächst die Position des ersten Vorkommens eines Eintrage $\neq 0$ in f_{k+1}^{-}, dann merken wir uns die gestrichenen Spalten in einem Vektor und streichen dann diesen aus der Matrix A. Wir erhalten dann eine n-$\times$-(n-1)-Matrix B.

Es bleibt nun noch zu zeigen

2.38. Problem
Wie berechnen wir eine linear abhängige Zeile aus der Matrix B?

Wir streichen die n-te Zeile aus der Matrix B, sichern diese Zeile im Vektor d und transponieren die resultierende $(n-1) \times (n-1)$-Matrix. Diese Matrix nennen wir A^T zur Vereinfachung der Schreibweise. Wir lösen das Gleichungssystem $A^T * x = d$ mit unserem erweiterten Algorithmus um eine Abhängigkeit festzustellen. Erhalten wir eine Lösung dieses Systems so ist d linear abhängig von dem Spaltenvektoren von A^T und somit d eine linear abhängige Zeile von A. Sonst läßt sich durch dem Algorithmus zur Suche einer linear abhängigen Spalte eine andere Spalte von A^T berechnen, die uns eine linear abhängige Zeile für A liefert.

Wir können den Algorithmus auf Kosten einer schlechteren Laufzeit durch sukzessives Zeilen- und Spaltenstreichen terminieren lassen, falls keine Lösung für $A * x = b$ existiert. Eine bessere Lösung ist abzufragen, ob das Erzeugnis Span(lg) der berechneten Lösung lg für $A^T * x = d$ und das Erzeugnis Span(b) des Vektors b einen gemeinesamen Teilraum haben oder nicht. Ist der Schnitt von Span(lg) und Span(A) leer so existiert keine Lösung des Systems.

Liefert das Lösungsverfahren für $A^T * x = d$ eine Abhängigkeit, darf der Schnitt der Erzeugnisse Span(F_b) und Span(A) nicht leer sein, um eine Lösung des System $A * x = b$ nicht ausschließen zu können.

Nachfolgender Satz beschreibt diesen Sachverhalt formal:

2.39. Satz
Sei $A \in K^{n,n}, a_1, \ldots, a_n$ die Spaltenvektoren der Matrix A, dann gilt:

1. *a_n ist linear abhängig von $a_1, \ldots, a_{n-1}$ und das Skalarprodukt $< x, b >= 0$ mit x ist nicht triviale Lösung des homogenen Gleichungssystems $A * x = 0$ so gibt es keine Lösung für $A * x = b$.*

2. *a_n linear unabhängig von $a_1, \ldots, a_{n-1}$ und das Skalarprodukt $< F_b, b >= 0$ mit F_b der Fehler in b angibt, so hat $A * x = b$ keine Lösung.*

Beweis:

1. a_n ist linear abhängig von $a_1, \cdots, a_{n-1}$
   $\Longrightarrow$ es gibt eine nicht triviale Lösung des homogenen Gleichungssystems $A*x = 0$.

   Das Gleichungssystem $A * x = b$ habe eine Lösung
   $\Longrightarrow b \in \text{Span}(a_1, \cdots, a_{n-1})$
   Sei lg ein nicht triviale Lösung von $A * x = 0$
   $\Longrightarrow lg \in \text{Span}(a_1, \cdots, a_{n-1})$
   $\Longrightarrow < lg, b > \neq 0$.

2. a_n ist linear unabhängig von $a_1, \cdots, a_{n-1}$
   Da A jedoch singulär ist, gibt es einen Vektor a_i der linear abhängig von $\{a_1, \cdots, a_{i-1}, a_{i+1}, \cdots a_n\}$ ist.

   Habe $A * x = b$ eine Lösung, so liegt b im Span$(a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$
   $F_b = b - A*y$ der Fehler im Ergebnis. y ist die durch den Algorithmus bestimmte Lösung. Da wir die Lösbarkeit des Gleichungssystems $A*x = b$ vorausgesetzt haben, gibt es auch ein Lösung für $A * x - A * y = \mathbb{F}_b$
   $\Longrightarrow F_b$ ist linear abhängig von $\{a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n\}$
   $\Longrightarrow < F_b, b > \neq 0$

$\blacksquare$

Auf diesem Hintergrund entstand nach einer Idee von Douglas H.Wiedemann der

2.40. Algorithmus [Wiedemann quadratisch]

Algorithmus zur Lösung Gleichungssysteme mit beliebigen quadratischen Matrizen

EINGABE: A,b
AUSGABE: lg

```
while (1) do
   pol = Wiedeprob3(A, b, lg, F_b);
   if (pol = NULL) then                    lg ist Lösung
      exit;
   fi
   i = vek_vgl(F_b, 0);                    F_b ist Fehler in b
   if (i = 0) then                         Lösung gefunden
      exit;
   fi
   mat_spaltenstreich(A, i);               i linear abhängige Spalte
   vek = suchevek(A, n);                   sichere Zeile n
   A^T = transmat(A);                      Transponiert A ohne n-te Spalte
   pol = Wiedeprob3(A^T, vek, lg, F_vek);
   if (pol = 0) then
      if (< b, lg >= 0) then
         exit;                             keine Lösung
      else
         i = n;                            a_n linear abhängig
      fi
   else
      if (< b, F_vek >= 0) then
         exit;                             keine Lösung
      else
         i = vek_vgl(F_vek, 0);            berechnet i aus dem Fehler f_A b und
                                           dem Polynom pol
      fi
      mat_zeilenstreich(A, i);
   od
```

In Zeile (2) untersuchen wir, ob das Gleichungssystem $A * x = b$ eine reguläre Koeffizientenmatrix A besitzt oder ob sie singulär ist.
Ist $pol = NULL$, so ist A regulär und die Lösung ist schon berechnet worden. Sie steht in lg. In diesem Fall können wir den Algorithmus terminieren (Zeile 4).
Ist $pol \neq NULL$, so ist A singulär. Wir berechnen deshalb in Zeile 6 aus dem Fehler F_b durch einen Vergleich mit dem Nullvektor, die erste Position ungleich Null in F_b. Diese Position sei i.
Ist $F_b = 0$ haben wir trotz singulärer Koeffizientenmatrix A eine Lösung für $A*x = b$ gefunden und terminieren der Algorithmus (Zeile 8). Ist $F_b \neq 0$, so streichen wir die i-te Spalte aus der Matrix A (Zeile 10).
Um eine linear abhängige Zeile zu finden, sichern wir uns den n-ten Zeilenvektor von A und tranponieren die verbleibenden $(n-1) \times (n-1)$-Matrix. Die resultierende Matrix nennen wir A^T. Wir wenden Algorithmus 2.36 auf A^T und vek an, um eine Abhängigkeit zu finden.
Ist $pol = NULL$, so überprüfen wir mit dem Skalarprodukt $< lg, b >$, ob eine Lösung existiert. ist $< lg, b >= 0$, so wissen wir nach Satz 2.39, daß das Gleichungssystem $A * x = b$ keine Lösung haben kann, und terminieren den Algorithmus (Zeile 16).
Ist $pol \neq NULL$, so überprüfen wir auch hier, ob eine Lösung existiert. Ist $< F_b, b > = 0$, so wissen wir nach Satz 2.39 gibt es keine Lösung, und wir terminieren den Algorithmus (Zeile 22).
Wenn wir den Algorithmus weiterführen, streichen wir eine linear abhängige Zeile aus A (Zeile 26) und beginnen die Schleife erneut.

Die Schleife wird solange ausgeführt, bis der Algorithmus eine Lösung findet, oder in Schritt (16) oder (22) ohne Lösung abbricht. Die Schleife wir maximal $(n-1)$-mal durchgeführt, da wir für eine 1×1-Matrix mit Lösungsvektor $b \in K^1$ entscheiden können, ob $A * x = b$ lösbar ist.

Eine Aussage über Laufzeitverhalten und Speicherplatzbedarf dieses Algorithmus liefert uns nachfolgender Satz.

2.41. Satz

Sei $A \in K^{n,n}, r = rang(A)$
Der Algorithmus 2.40 für beliebige quadratische Matrizen hat eine Laufzeit von

$$O((n-r) * (2n * \log(n) * (w + \log(n) * \log(\log(n))) + 3w + n))$$

und benötigt

$$O(2w + 4n)$$

Speicherplatz.

Beweis:

1. Laufzeit:
   Sei $A \in K^{n,n}, r = rang(A)$, w die Anzahl der Eintrage, so werden maximal n-r Zeilen- und Spaltenstreichoperationen ausgeführt, um eine Aussage über die Lösbarkeit vom $A * x = b$ zu erhalten. Für uns heißt das, wir durchlaufen die Schleife maximal $(n - r)$-mal.
   In jeder Schleife werden folgenden Funktionen aufgerufen:

Wiedeprob3	$O(n * \log(n) * (w + \log(n) * \log(\log(n))))$
mat_spaltenstreich	$O(w)$
suchevek	$O(n)$
transmat	$O(w)$
skalarprod	$O(n)$
mat_zeilenstreich	$O(n)$

   Dies führt zu pro Schleife von $O(2n*\log(n)*(w+\log(n)*\log(\log(n)))+3w+n)$. Multiplizieren wir Kosten pro Schleife mit der Anzahl der Schleifendurchläufe erhalten wir als Ergebnis unsere Behauptung.

2. Speicherplatzbedarf:
   Die Routine benötigt Speicher für

zwei Matrizen	$2w$
drei Vektoren	$3n$
ein Polynom	n

   Dies führt zu einem Gesamtbedarf von $O(2w + 4n)$.

■

Es bleibt nur noch zu zeigen, wie wir eine Lösung für beliebige Matrizen finden können. Dieser Frage werden wir im nächsten Abschnitt nachgehen.

2.5 Erweiterung für beliebige Gleichungssysteme

Hier gibt es zwei Varianten die angewendet werden können.

1. Die Rechteckmatrix wird zu einer quadratischen Matrix erweitert, dann wenden dann den Wiedemann-Algorithmus für quadratische Matrizen darauf an

2. Wir schneiden uns eine quadratische Matrix aus der Rechteckmatrix aus wenden den Algorithmus für quadratische Matrizen darauf an. Die nicht betrachteten Zeilen oder Spalten werden durch als linear abhängig erkannte ersetzt, und der Algorithmus für quadratische Matrizen wiederholt angewendet.

Wir beginnen mit der Beschreibung der ersten Variante.

2.5.1 Erweiterung auf eine quadratische Matrix

2.42. Problem

Wie muß die Erweiterung der Rechteckmatrix aussehen, damit eine Lösung des erweiterten Systems auch Lösung des ursprünglichen Systems ist?
Wie können wir aus einer Lösung des erweiterten Systems eine Lösung für das ursprüngliche berechnen?

Eine Lösung ist die fehlenden Zeilen oder Spalten mit Nullen zu füllen. In diesem Fall sind die zusätzlichen Gleichungen immer erfüllbar und zusätzliche Nullspalten ändern nichts an den Gleichungssystemen.

Diese Art der Erweiterung hat jedoch einen großen Nachteil: Wir fügen eine Vielzahl von linear abhängigen Vektoren ein. Dies wirkt sich negativ auf das Laufzeitverhalten aus.

Einen besseren Ansatz ist folgender: Wir erweitern die Matrix A durch linear unabhängige Vektoren, die wir zufällig erzeugen. Ist die Matrix A von Maximalgrad, d.h. der Rang(A)=min(n,m), so erhalten wir eine reguläre Matrix B, für die eine eindeutig bestimmte Lösung existiert. Ist A singulär, so bleibt die Anzahl der linear abhängigen Vektoren konstant. Falls $A * x = b$ eine Lösung hat, hat auch $B * x = b$ eine Lösung.

Wir unterscheiden zwei Fälle:

1. Überbestimmte Gleichungssysteme mit $n > m$

2.43. Satz

Sei $A \in K^{n,m}$, $n < m$, Sei $e : K^{n,m} \longrightarrow K^{m,m}$ Einbettungshomomorphismus, der A durch Hinzunahme von zu den Zeilen von A linear unabhängigen Vektoren erweitert, sei $Q \in K^{n,m}$ der Operator zu e, dann gilt:

$A * x = b$ hat genau dann eine Lösung, wenn für $e(A) = Q * A$ und $e(b) = Q * b$ das Gleichungssystem $Q * A * x = Q * b$ eine Lösung hat.
Die Lösung des erweiterten Systems ist auch die Lösung von $A * x = b$.

Beweis:
Wir wollen zeigen, daß $e(A * x = b) = Q * A * x = b$

Sei lg eine beliebige Lösung des Gleichungssystems $A * x = b$
$\Longrightarrow e(A * lg) = e(b)$
$\Longleftrightarrow e(A) * e(lg) = e(b)$ (e Homomorphismus)
$\Longleftrightarrow Q * A * e(lg) = Q * b$
$\Longrightarrow e(lg)$ ist Lösung für $Q * A * x = Q * b$.

$e(lg) = lg$ folgt aus der Äquivalenz $A * x = b \Longleftrightarrow Q * A * x = Q * b$, da Q Operator des Einbettungshomomorphismus e in $K^{m,m}$ ein zum span$(a_1, \cdots, a_n)$ isomorphes Bild anlegt. Die a_i, $i = 1, \cdots, n$ bezeichnen die Zeilenvektoren von A.

■

2. Unterbestimmte Gleichungssysteme mit $n < m$

2.44. Satz
Sei $A \in K^{n,m}$; $n > m$, sei $e : K^m \longrightarrow K^n$ Einbettungshomomorphismus, der A durch Hinzunahme von zu den Vektoren aus A linear unabhängigen Vektoren erweitert, sei $Q \in K^{m,n}$ der Operator zu e, dann gilt:

$A * x = b$ hat genau dann eine Lösung, wenn für $e(A) = A * Q$ $A * Q * y = b$ eine Lösung hat.

Die Lösung für $A * x = b$ erhalten wir aus y, in dem wir die zusätzlichen Variablen auf Null setzen.

Beweis:
Wir wollen zeigen, daß $A * x = b \Longleftrightarrow A * Q * y = b$ mit $x = Q * y$ ist.

Sei lg eine beliebige Lösung von $A * x = b$
$\Longrightarrow A * lg = b$
$\Longleftrightarrow e(A * lg) = b$
$\Longleftrightarrow e(A) * e(lg) = b$ (e Homomorphismus)
$\Longleftrightarrow A * Q * e(lg) = b$
d.h. $e(lg)$ ist Lösung für $A * Q * y = b$

Zu zeigen bleibt, daß wir aus $e(lg)$ eine Lösung lg erhalten, indem wir die zusätzlichen Variablen auf 0 setzen.

Hierzu betrachten wir den kanonischen Vektorraumhomomorphismus $h : K^n \longrightarrow M$, der Vektoren aus M identisch auf sich abbildet. $M = span(a_1, \cdots, a_m)$ mit a_i, $i = 1, \cdots, m$ die Spaltenvektoren von A.

Seien $(v_1, \cdots, v_r)$ eine Basis von A, $r = \text{rang}(A)$ und $(v_1, \cdots, v_n)$ eine Basis des $K^{n,n}$. Somit läß sich jeder Vektor $z \in K^{n,n}$ darstellen als

$$z = \sum_{i=1}^{r} \lambda_i * v_i + \sum_{i=r+1}^{n} \lambda_i * v_i$$

Daher gilt:

$$
\begin{aligned}
h(z) &= h(\sum_{i=1}^{r} \lambda_i * v_i + \sum_{i=r+1}^{n} \lambda_i * v_i) \\
&= h(\sum_{i=1}^{r} \lambda_i * v_i) + h(\sum_{i=r+1}^{n} \lambda_i * v_i) \\
&= \underbrace{\sum_{i=1}^{r} \lambda_i * v_i}_{\in M} + h(\sum_{i=r+1}^{n} \lambda_i * v_i)
\end{aligned}
$$

Daraus folgt nun $\sum_{i=r+1}^{n} \lambda_i * v_i \in M$. Die v_i für $i = 1, \cdots, r$ linear unabhängig von v_i für $i = r+1, \cdots, n$ sind, ergibt sich

$$h(\sum_{i=r+1}^{n} \lambda_i * v_i) = 0.$$

■

2.45. Problem
Wie bestimmen wir die zu den Vektoren von A linear unabhängigen Vektoren zum Auffüllen zur Matrix B?
Können wir die Matrix B so erzeugen, daß sie dünnbesetzt ist?

Diese Fragen gehen wir hier auch getrennt für überbestimmte und unterbestimmte Gleichungssystem nach.

Sei $A * x = b$ ein unterbestimmtes Gleichungssystem, so fehlen uns Zeilenvektoren, um die Matrix A auf eine $m \times m$-Matrix zu erweitern. Mit der Hinzunahme von zusätzlichen Zeilenvektoren sind Einträge in den Vektor b einzufügen. Wir wählen diese Eintrage gleich Null. Die hinzugefügten Gleichungen bilden ein homogenes Gleichungssystem. Es ist zumindest mit der trivialen Lösung $x = 0$ lösbar.

Um die Dünnbesetztheit der Matrix B zu garantieren, führen wir Gewichte w_i ein. Sie sind ein Maß für die Anzahl der Einträge pro Zeile. Sei p die Dimension der Körpers K, so wählen wir für die i-te Zeile ein Gewicht von $w_i \leq 1 - p^{-i}$.

Dies führt zu folgenden

2.46. Algorithmus [Einbettung]

Algorithmus zur Erweiterung einer Rechteckmatrix auf quadratische Gestalt
nach einer Idee von Douglas H. Wiedemann

EINGABE: A
AUSGABE: B

(1) $i = \min(n, m)$
(2) **while** $(i < \max(n, m))$ **do**
(3) $\quad w_i = \text{rand}()\%(1 - p^{-i})$ — wähle $w_i \leq 1 - p^{-i}$
(4) $\quad \text{random}(vek, w_i, \max(n, m))$ — besetze den Vektor vek der Länge m zufällig mit w_i Einträgen ungleich Null
(5) **od**

Betrachten wir jetzt den zweiten Fall. Das Gleichungssystem $A * x = b$ ist überbestimmt, d.h wir müssen weitere Spalten hinzufügen um ein quadratisches Gleichungssystem zu erzeugen. Die Wahl der Spaltenvektoren erfolgt wie im unterbestimmten Fall, obiger Algorithmus kann also auch für überbestimmte Gleichungssysteme angewendet werden.

Durch diese Variante lassen sich auch mehrere Lösungen des gleichen Gleichungssystem finden.

Da wir die Vektoren zufällig wählen, erhalten wir nicht mit 100%iger Sicherheit eine Matrix B, die die gleiche Anzahl linear abhängiger Vektoren hat wie A.

Douglas H. Wiedmann gibt uns zur Bestimmung der erweiterten Matrix B eine Laufzeit von

$$0(\max(n, m) * (w + \max(n, m) * \log(\max(n, m))) \tag{2.11}$$

an bei einer Wahrscheinlichkeit von 1/2 das B nur zu A linear unabhängigen Vektoren erweitert wurde.

Sie $t = \max(n, m)$ so führt dies zu Gesamtkosten von

$$0(t*(w+t*\log(t)+(t-r)*2*t*\log(t)*w+\log(t)*\log(\log(t)))+3w+t) \tag{2.12}$$

bei einem Speicherplatzbedarf von

$$0(3 * w + 4 * t). \tag{2.13}$$

2.5.2 Reduktion auf eine quadratische Matrix

Die zweiter Variante Gleichungssysteme mit Rechteckmatrizen zu lösen ist, eine Teilmatrix auszuschneiden, eine Lösung für das Teilsystem zu suchen und diese dann in eine Lösung des ursprünglichen Systems zu überführen.

Auch hier betrachten wir überstimmte und unterbestimmte Gleichungssysteme getrennt voneinander.

Unterbestimmte Gleichungssysteme

Es gibt mehr Variablen zu bestimmen als Gleichungen vorhanden sind. Wie oben schon erwähnt, schneiden wir die ersten n Spalten der Matrix A aus und lösen das System $A_n * x = b$. Hat dieses System eine Lösung so ist x erweitert um n-m Nulleinträge Lösung des Gesamtsystems. Sonst haben wir in eine linear abhängige Spalte gefunden, die wir durch eine Spalte von A ersetzten, die wir bis jetzt nicht betrachtet haben. Dann beginnen wir das Spiel von neuem. Wir terminieren den Algorithmus, wenn wir eine Teillösung gefunden haben, indem wir die Einträge in x bezüglich der Spaltenvertauschen umordnen, oder alle Spalten von A betrachtet wurden.

Den theoretischen Hintergrund liefert uns der

2.47. Satz
Sei $A \in K^{n,n}, n < m$
$A * x = b$ hat eine Lösung, genau dann wenn eine quadratische Matrix $A_{m,m}$ existiert, die eine Lösung für $A_{m,m} * x_m = b$ liefert. $A_{m,m}$ entsteht aus A durch Wahl von m paarweise verschiedenen Spaltenvektoren.

Beweis:

- Sei Q eine $n \times m$-Matrix dünnbesetzt von rang(n), Q wirke rangerhaltend auf A, sei $x = Q * y$
  Sei x_b Lösung zu $A * x = b$, dann folgt $A * x = b \Longleftrightarrow A * Q * y = b$ Setze $A_{m,m} = A * Q$ so hat $A_{m,m} * y = b$ eine Lösung.
- Sei y_b Lösung zu $A_{m,m} * y = b$
  Setzen wir die Eintrage in x_b entsprechend den verwendeten Spalten vom A und setzen die übrigen Eintrag auf Null, so ist x_b Lösung des Gleichungssystems $A * x = b$.

■

Daraus resultiert folgender

2.48. Algorithmus [nkm]

Wiedemann Algorithmus für unterbestimmte Gleichungssysteme nach einer Idee von Douglas H. Wiedemann

EINGABE: A,b
AUSGABE: lsg

(1)	aktuell=m;	Anzahl der betrachten Spalten
(2)	$B = \text{quadmat}(A)$;	B enthält die ersten n Spalten
(3)	**do**	
(4)	$p = \text{pwiedemann}(B, b, lg)$;	löse $B * lg = b$
(5)	**if** (p==NULL) **then**	Teillösung gefunden
(6)	**for** $(i = 0; i < n, i++)$ **do**	Umordung der Einträge entsprechend der Vertauschungen
(7)	$lg[i] = erg[spvek[i] - 1]$	
(8)	exit;	Lösung gefunden
(9)	**od**	
(10)	**else**	
(11)	**if** $(aktuell < n)$ **then**	noch nicht alle Spalten betrachtet
(12)	**do**	
(13)	$j = \text{mat\_spaltentausch}(B, i, A, ++aktuell)$	
(14)	**while** $(j = 0)$	ersetzt keine Nullzeilen
(15)	**if** $(j \neq 0)$ **then**	
(16)	$spvek[i-1] = aktuell$	Eintragung der ersetzenden Spalte
(17)	**else**	
(18)	exit;	keine Lösung
(19)	**fi**	
(20)	**fi**	
(21)	**fi**	
(22)	**while** $(aktuell \geq n)$	

In Schritt (2) führen wie durch den Aufruf von quadmat() das Ausschneiden der ersten n Spalten aus. Danach beginnen wir mit der Schleife (3 – 22).
Wir wenden die Routine pwiedemann() auf B und b an und erhalten entweder eine Lösung des Systems oder die Position einer linear abhängigen Spalte.

Ist $p = NULL$ (Zeile 5), so haben wir für $B * x = b$ eine Lösung gefunden. In Zeile (6 – 9) ordnen wir die Einträge in lg entsprechend den Vertauschungen um. Das Ergebnis steht dann in erg und wir terminieren den Algorithmus.

Ist $p \neq NULL$, so liefert uns der Algorithmus pwiedemann() die Position einer linear abhängigen Spalte. Diese ersetzen wir durch mat_spaltentausch() in Zeile (13) durch eine noch nicht betrachtete Spalte von A. Sind schon alle Spalten von A betrachtet, terminiert der Algorithmus ohne Lösung.

Wir betrachten nun die Kosten die dieser Algorithmus verursacht.

2.49. Satz

Sei A eine $n \times m$-Matrix, r=rang(A), w die Anzahl der Einträge dieser Matrix so benötigt der Algorithmus 2.48

$$0((m-n)*((n-r)*(2n*\log(n)*(w+\log(n)*\log(\log(n))+3w+n)+w))) \tag{2.14}$$

bei einem Speicherplatzbedarf von

$$2w + 6n \tag{2.15}$$

Beweis:
Die Routine quadmat() benötigt einen Paŝs über die Matrix A, um die Spalten auszuschneiden also $0(w)$ Körperopertationen. Das Gleiche gilt für mat_spaltentausch(). Für die Routine pwiedemann() haben wir in 1.39 schon gezeigt, das sie eine Laufzeit von $0((n-r)*(2n*\log(n)*(w+\log(n)*\log(\log(n)))$ hat.

Die Prozedur quadmat() wird nur einmal vor Schleifenbeginn ausgeführt, während pwiedemann() und mat_spaltentausch in der Schleife ausgerufen werden. Diese werden im schlechtesten Fall $(m-n)$-mal wiederholt.

Dies führt zu einer Gesamtlaufzeit von

$$0((m-n)*((n-r)*(2n*\log(n)*(w+\log(n)*\log(\log(n))+3w+n)+w))).$$

Zu zeigen bleibt die Speicherplatzkomplexität. Wir benötigen Speicherplatz für zwei Matrizen, A, B, für fünf Vektoren und ein Polynom p. Dies für zu eine Bedarf von

$$2*w+6*n$$

■

Wir betrachten nun den noch verbleibenden Fall des überbestimmten Gleichungssystems.

Überbestimmte Gleichungssysteme

Der Algorithmus basiert aus der Tatsache, daß das Gleichungssystem $A*x=b$ nur dann eine Lösung haben kann, wenn jedes Teilsystem eine Lösung hat. Eine genauere Aussage liefert der nächste

2.50. Satz
Sei $A \in K^{n,m}$, $n > m$,
Sei A_n eine Matrix, gebildet aus n paarweise verschiedenen Zeilenvektoren von A, sei $b_n \in K^n$. Die Einträge von b_n sind bezüglich der Zeilen für A_n gesetzt, dann gilt: $A_n * x = b_n$ ist nicht lösbar für alle A_n erzeugbar aus A, genau dann wenn $A*x=b$ keine Lösung hat.

Beweis:
Sei $h : K^n \longrightarrow K^m$, die lineare Abbildung, die alle Vektoren aus A auf die Vektoren von A_n abbildet. Sei Q der Operator von h.

z.z.: $A * x = b$ hat eine Lösung, wenn es eine lineare Abbildung h gibt, sodaß $h(A * X) = b$ eine Lösung hat.

Sei lg eine Lösung für $A * x = b$

$$\begin{aligned} \Longrightarrow \quad h(A * lg) = b \quad &\Longleftrightarrow h(A) * h(lg) = b \\ &\Longleftrightarrow A * Q * h(lg) = b \end{aligned}$$

$\Longrightarrow h(lg)$ ist Lösung für $A * Q * y = b$. ∎

2.51. Algorithmus [mkn]

Wiedemann Algorithmus für überbestimmte Gleichungssysteme nach der Idee von Douglas H. Wiedemann

EINGABE: A,b
AUSGABE: lsg

j=0;	
B=quadmat(A);	B ersten m Zeilen von A
h=cat(b);	h ersten m Einträge von b
do	
$j++$;	
$p =$ pwiedemann(B, h, lg);	
if $(p \neq NULL)$ **then**	Lösung gefunden
exit;	keine Lösung
else	
matmulvek$(A, lg, hilf)$;	teste Lösung
$h_2 =$ vek_vgl$(b, hilf, m)$;	
if $(h_2 = 0)$ **then**	
exit;	Lösung gefunden
else	
$d =$ suchevek(A, h_2);	
$p =$ pwiedemann$(atrans, d, lg)$;	
if (p==NULL) **then**	keine Lösung
exit;	
else	
mat_zeilentausch(B, i, d);	linear abhängige Zeile wird durch eine ersetzt die noch nicht betrachte wurde
$h[i-1] = b[h_2-1]$;	der entsprechende Eintrag in h wird überschrieben
fi	

```
(23)       fi
(24)     fi
(25)     while (j ≤ n) do
```

Wir schneiden auch hier zunächst eine quadratische Matrix B aus (Zeile 2). Sie besteht aus den ersten m Zeilen aus A. Entsprechend der gewählten Zeilenvektoren von A setzen wir die Einträge von h (Zeile 3).

Nun beginnt die Schleife (4 – 25):
Wir wenden pwiedemann() auf B und h an. Liefert die Routine $p \neq Null$ zurück, wissen wir, das Gleichungssystem $B * x = h$ hat keine Lösung. Daraus folgt nach Satz Satz 2.50, daß das ursprüngliche Gleichungssystem $A * x = b$ auch keine Lösung besitzt. Deshalb terminieren wir den Algorithmus an dieser Stelle. (Zeile 7 – 8)

Ist $p = NULL$, so müssen wir testen, ob die Lösung des Teilsystems schon eine Lösung des Gesamtsystems ist. Dies geschieht, indem wir $A * lg$ berechnen (Zeile 10 – 11) und das Ergebnis mit h vergleichen (Zeile 12). Ist $h_2 = 0$ haben wir eine Lösung gefunden und wir können den Algorithmus terminieren.

Ist $h_2 \neq 0$, so extrahieren die h_2-te Zeile d der Matrix A. Finden wir mittel pwiedemann() eine Lösung für $A^T * x = d$ (Zeile 16), so ist d linear abhängig von den verbleibenden Vektoren der Matrix A. Daraus folgt direkt, das das Gleichungssystem $A * x = b$ keine Lösung hat. Ist d jedoch linear unabhängig von den übrigen Vektoren aus A, so tauschen wir die h_2-te Zeile mit dem Vektor d und beginnen die Schleife von neuem.

2.52. Satz *Der Algorithmus 2.51 hat eine Laufzeit von*

$$O((n - m) * ((n - r) * (2n * \log(n) * w + \log(n) * \log(\log(n))) + 2w + n) = w)$$

bei einem Speicherplatzbedarf von

$$O(w + 4n).$$

Dabei sei A eine $n \times m$-Matrix $r = \operatorname{rang}(A)$, w die Anzahl der Einträge dieser Matrix.

Beweis: Wir verwenden in diesem Algorithmus die Routinen

- quadmat()
  wir müssen jeden Eintrag der Matrix betrachten, daher benötigen wir w Körperoperationen.

- pwiedemann()
  Hier haben wir schon gezeigt, daß er Zeit $O((n-r)*(2n*\log(n)*w+\log(n)*\log(\log(n)))$ benötigt. Wir nehmen dabei an, daß der Rang der erzeugten quadratischen Matrix nicht wesentlich vom Rang der Ausgangsmatrix abweicht, die Anzahl der Einträge ist kleiner als w.

- mat_zeilentausch()
  Diese Routine benötigt maximal $O(n)$ Körperoperationen, da wir eine Zeile der Matrix durch einen Vektor in dichter Repräsentation überschreiben.

- matmulvek()
  hat Laufzeit O(w).

- transmat()
  Jeder Eintrag der Matrix muß betrachtet werden, daher beläuft sich die Laufzeit auf $O(w)$.

Die Routine quatmat() tritt innerhalb der Schleife nicht auf, wird also nur einmal ausgeführt.
In schlechtesten Fall wird die Schleife $(n-m)$-mal durchlaufen, so daß wir eine Gesamtlaufzeit von

$$O((n-m)*[(n-r)*(2n*\log(n)*w+\log(n)*\log(\log(n)))+2w+n]=w)$$

nachweisen können.

Speicherplatz benötigen wir für drei Matrizen A, B, A^T, für fünf Vektoren und ein Polynom, so kommen wir auf einen Bedarf von

$$O(3w+6n).$$

■

Kapitel 3

Zur Implementierung

Die Algorithmen in Kapitel 2 sind so implementiert worden, daß sie unabhängig von den gewählten Datenstrukturen für Matrix und Vektor sind. Sie können jederzeit durch eine Arithmetik mit gleichen Schnittstellendefinition ersetzt werden.

Wir wollen in diesem Kapitel folgenden Konventionen beachten: n bezeichne die Anzahl der Zeilen der Matrix, m die Anzahl der Spalten. Mit w bezeichnen wir die Anzahl der Nichtnulleinträge der Matrix. Desweiteren beschränken wir uns in diesem Kapitel auf Körper der Ordnung $p \leq 2^{32}$. Unter einer Körperoperation wollen wir Lese- und Schreiboperationen eines Körperelements bezeichnen. Wir beziehen die arithmetischen Operationen auf dem Körper auch in diesen Begriff mit ein.

Wir beginnen mit der Beschreibung der

3.1 Externen Datenstrukturen

Die Eingabedaten für Vektoren und Matrizen liegen in Dateien vor. Die Datei zur Vektoreingabe hat dabei folgendes Format:

Als erster Eintrag steht die Länge n des Vektors, dann folgen die Vektoreinträge als Folge von Elementen aus K. Wir wählten die dichte Repräsentation, da die Vektoren in unseren Anwendung überwiegend dicht besetzt sind. Eine Darstellung in dünner Repräsentation, (Position, Wert), für allen Nichtnulleinträge verdoppelt etwa den Speicherbedarf. Selbst wenn die Vektoren nur halb besetzt sind bringt die dünne Repräsentation keine Verbesserung.

Das Problem der Matrixdarstellung gestaltet sich komplexer. Durch das sequentielle Eingabemedium, Festplatte, sind wir in unserer Auswahl stark eingeschränkt.

Eine Möglichkeit besteht darin, die Matrix zeilenweise in einer Datei abzulegen.

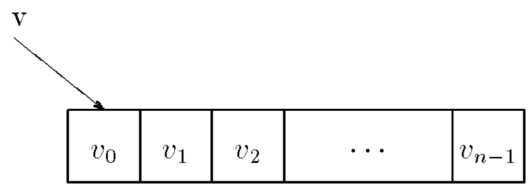
Wählen wir eine dichte Repräsentation für die Matrix A, so speichern wir die Nulleinträge mit ab, und erhalten einen konstanten Speicherplatzbedarf von $n * m$, ohne darauf zu achten, ob A dicht oder dünnbesetzt ist. Beziehen wir die Dichte w der Matrix in unsere Überlegungen mit ein, so existiert folgenden Darstellung:

Die beiden ersten Einträge enthalten die Dimension der Matrix, Zeilenanzahl gefolgt von der Spaltenanzahl der Matrix. Danach folgen die Matrixeinträge ungleich Null mit Angabe der Zeilen- und Spaltenposition. Wir haben dann einen Speicherplatzbedarf von $3 * w$. Ist die zu betrachtende Matrix dicht besetzt, entspricht dies eine Verdreifachung des Speicherbedarfes. Der Wiedemannalgorithmus, den wir hier betrachten, ist jedoch gedacht für stark dünnbesetzte Matrizen. Daher verwenden wir die zweite Darstellung in unserer Implementierung.

Wir beschreiben nun

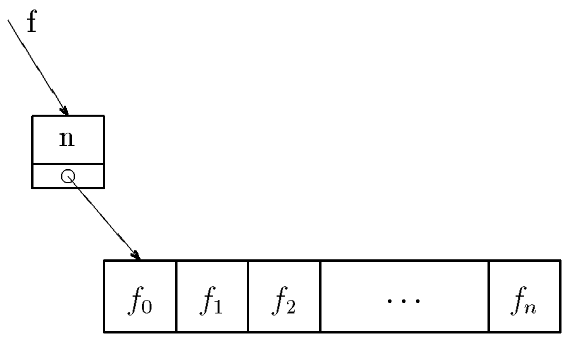
3.2 Die internen Darstellung der Datenstrukturen

Einen Vektor der Länge n stellen wir als Liste von Körperelementen dar. Da die Vektoren, die wir verwenden überwiegend dicht besetzt sind, lohnt es sich nicht über eine dünne Repräsentation zu diskutieren. Neben den größeren Platzbedarf setzen wir auch die direkte Zugriffmöglichkeit, bedingt durch die Array-Darstellung, aufs Spiel. Für uns sieht daher ein Vektor folgendermaßen aus:



v ist ein Zeiger auf ein Array von Körpereinträgen v_i $i = 0, \cdots, n-1$.

Die Darstellung der Polynome gleicht der des Vektors. Für Polynome führen wir zusätzlich zum Array eine Struktur ein, die den Grad des Polynoms speichert. Ein Polynom f vom $\mathrm{grad}(f) = n$ hat für uns folgende Darstellung



f ist ein Zeiger auf eine Struktur Polynom. Sie enthält als Eintrag den Grad des Polynoms und einen Zeiger auf ein Array der Polynomkoeffizienten $f_0, \cdots, f_n$. f_i ist dabei der Koeffizient des Monoms $f_i * X^i$ des Polynoms f.

Wesentlich ausführlicher gehen wir auf die Matrixdarstellung ein. Wir wollen zwei Varianten vorstellen und motivieren unsere Wahl der Darstellung.
Zur Diskussion stehen:

1. Darstellung als lineare Liste
2. Darstellung als Array von Zeilenvektoren

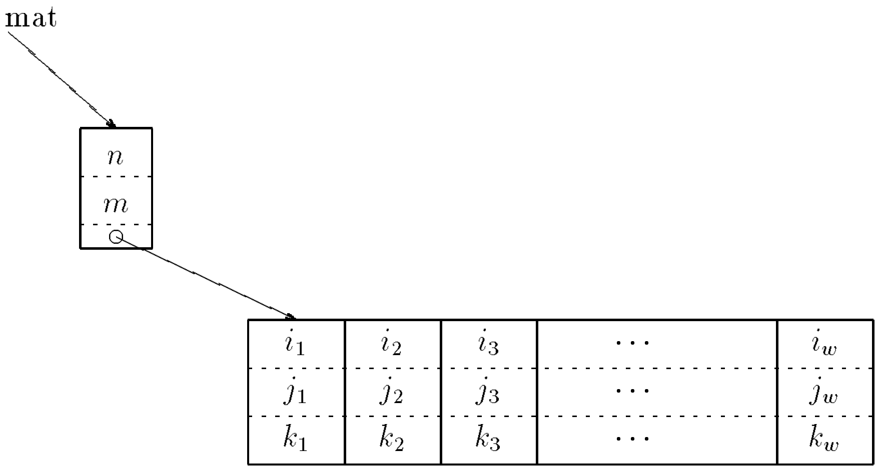
Wir wollen folgende Kriterien anwenden:

1. Komplexität im Aufbau der Datenstruktur
2. Speicherplatzbedarf
3. Kosten für matrixspezifische Routinen wie Zeilen- und Spaltenoperationen und Transponiertenbildung.

3.2.1 Matrixdarstellung als lineare Liste

Wir definieren eine Struktur Matrix als eine Zeiger auf eine lineare Liste von Matrixeinträgen. Im Matrixkopf sind sowohl Zeilen- wie Spaltenanzahl gespeichert.

Ein Matrixeintrag sieht folgendermaßen aus:
Er enthält einen Zeileneintrag i, einen Spalteneintrag j und einen Eintrag für den Wert und der Stelle (i, j). Hieraus resultiert folgende Darstellung für eine $n \times m$-Matrix mit w Einträgen:



Der Zeiger mat zeigt auf eine Struktur Matrix. Sie enthält die Zeilenzahl n und die Spaltenzahl m sowie einen Zeiger auf ein Array von Matrixeinträgen als Eintrag. Ein Matrixeintrag ist eine Struktur (i, j, k). i bezeichnet die Zeilenposition, j die Spaltenposition und k den Wert des Matrixeintrags. w ist die Anzahl der Matrixeinträge der Matrix mat.

Diese Darstellung ist einfach zu realisieren. Wir brauchen für die Einträge nur eine Array von w Matrixeinträgen zu allokieren und können durch einmaliges Lesen der Eingabedatei, die innere Struktur aufbauen. Wir benötigen dazu etwa $3 * w$ Lesezugriffe auf Körperelemente.

Für die Darstellung der Matrix benötigen wir $3 * w + 2$ Speichereinheiten.

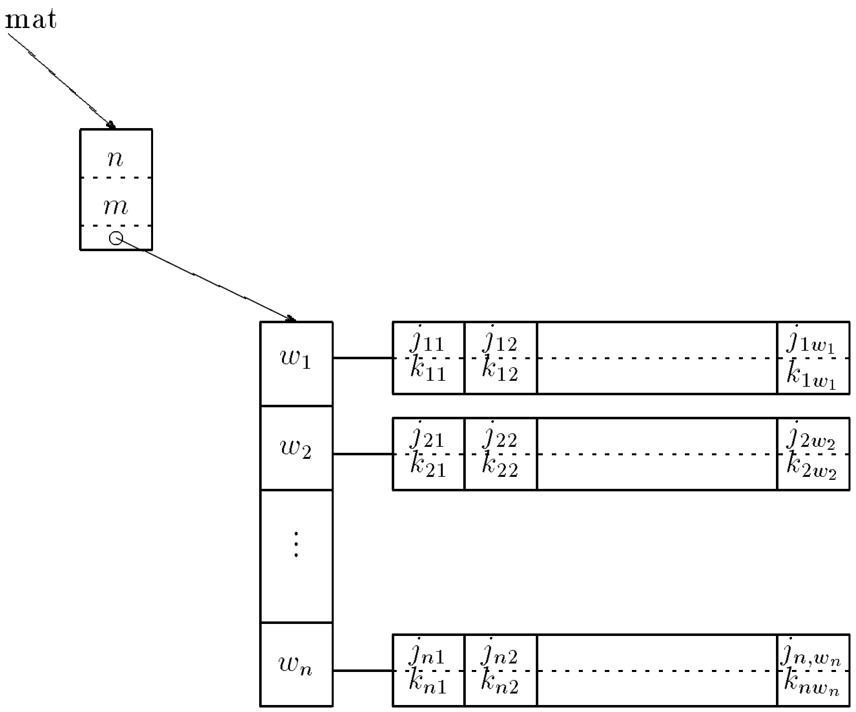
Zum Streichen von Spalten oder Zeilen aus der Matrix müssen wir einmal die gesamte Matrix lesen, d.h. für uns, wir benötigen etwa $3 * w$ Körperoperationen, um diese zu realisieren. Ähnliches gilt für das Vertauschen von Zeilen mit Zeilen oder Spalten mit Spalten.

Die Transponierte einer Matrix läßt sich auch in Linearzeit bezüglich der Nichtnulleinträge w erzeugen, in dem wir eine Liste von Zeigern $count$ einführen.
Bei jedem Einfügevorgang eine Matrixeintrages $(i,\ j,\ k)$ setzen wir $count[l]$ auf den zuletzt eingefügten Eintrag der Zeile l. Fügen wir einen weiteren Matrixeintrag in Zeile l ein, so können wir den Eintrag direkt ohne vorherige Suche nach dem Eintrag unter $count[l]$ einfügen. Daher hat diese Routine eine Laufzeit von $3 * n$.

3.2.2 Matrix als Liste von Zeilenvektoren

Wir definieren eine Struktur Matrix als einen Zeiger auf eine Liste von Vektoren. Diese Vektoren repräsentieren die Zeilen der Matrix. Durch diese Darstellung kann ein Eintrag allein durch die Angabe der Spalte innerhalb des Zeilenvektors lokalisiert werden. Der Matrixkopf entspricht dem der Darstellung als lineare Liste.

Für uns resultiert daraus folgende Darstellung für eine $n \times m$-Matrix mit w Einträgen.



Der Zeiger *mat* zeigt auf eine Struktur Matrix. Sie enthält als Einträge die Zeilenanzahl n, die Spaltenanzahl m und einen Zeiger auf ein Array von Zeilenzeigern. Diese zeigen dann auf ein Array von Matrixeinträgen. Ein Matrixeintrag besteht aus zwei Einträgen, der Spaltenposition j und dem Wert k. w_i, $i = 1, \cdots, n$ gibt jeweils die Länge der Zeile an.

Die Summe aller w_i in dieser Darstellung ergibt die Gesamtanzahl w der Nichtnulleinträge der Matrix.

Diese Darstellung sieht vom Bild her aufwendiger aus. Wir werden jedoch zeigen, das diese Darstellung besser für unsere Anwendung ist als die Darstellung mittels linearer Liste.

Wir legen den Matrixkopf in zwei Operationen an, allokieren eine Liste von n Zeigern auf Zeilenvektoren und fügen die w Matrixeinträge ein. Dies kann in $2+n+3*w$ Körperoperationen bewerkstelligt werden.

Diese Darstellung benutzt etwa $2*w+n+2$ Speichereinheiten.

Betrachten wir nun ihr Verhalten bei den matrixspezifischen Routinen:
Das Streichen von Zeilen kostet uns $O(1)$ Körperoperationen. Wir löschen die Matrixzeile, indem wir den Speicherplatz freigeben und den Zeilenzeiger aus der Liste der Zeiger löschen. Somit erhalten wir wieder unsere ursprüngliche Matrixform.
Das Streichen einer Spalte liegt in der gleichen Komplexitätsklasse, wie bei der Darstellung als lineare Liste. Wir benötigen auch hier $O(w)$ Körperoperationen. Die Konstante vor w ist jedoch besser, da wir im Mittel nicht in jeder Matrixzeile alle Einträge lesen müssen, sondern oft frühzeitig einen Eintrag der zu löschenden Spalte finden.

Bei der Funktion mat_zeilentausch überschreiben wir eine Matrixzeile mit einem Vektor in dichter Repräsentation. Die Kosten dieser Funktion bestehen darin, den Vektor aus dichter Repräsentation in die dünne Darstellung der Matrixzeile zu überführen. Hierzu benötigen wir $2 * n$ Körperoperationen.
Tauschen wir zwei Zeilen innerhalb einer Matrix, so ist die Transformation durch das Ändern zweier Zeigeradressen zu bewerkstelligen.
Die Generierung der transponierten Matrix ist auch hier in Linearzeit bezüglich w möglich, wenn wir eine Liste von Zeigern bereitstellen, die nach jedem Einfügen auf den letzten Eintrag der Zeile zeigt. Wir fügen jeweils nur am Ende einer Zeile ein, da die Matrix zeilenweise sortiert vorliegt. Hier eine Gegenüberstellung.

	lineare Liste	Zeilenvektoren
Aufbau der Struktur	$2 + 3 * w$	$2 + n + 3 * w$
Platzbedarf	$2 + 3 * w$	$2 + n + 2 * w$
Zeilenstreichen	$O(w)$	$O(1)$
Spaltenstreichen	$O(w)$	$O(w)$
Zeilentauschen (intern)	$O(w)$	$O(1)$
Zeilentauschen (extern)	$O(w)$	$O(n)$
Spaltentauschen	$O(w)$	$O(w)$
Transponierte	$O(w)$	$O(w)$

Wir können im Allgemeinen davon ausgehen, daß $n < w$ ist, somit zeigt sich daß die Matrixdarstellung mittels Zeilen besser ist. Die Verbesserung macht sich besonders bemerkbar bei Gleichungssystemen mit singulären Koeffizientenmatrizen und bei überbestimmten Gleichungssystemen bemerkbar. In diesen Fällen werden die oben genannten Routinen Zeilentauschen und Zeilenstreichen verwendet. Aufgrund dieser Analyse haben wir die Darstellung als Listen von Zeilenvektoren für unsere Implementierung gewählt.

3.3 Routinen der Matrix-Vektor-Arithmetik

Im Abschnitt über die Datenstrukturen aber auch im vorangehenden Kapitel haben wir schon einige Routinen angesprochen. Wir wollen diese jetzt vorstellen.

Da wir bei der Implementierung eine statische Speicherverwaltung zugrunde gelegt haben, können die Laufzeiten zwischen den theoretischen Betrachtungen aus vorherigen Abschnitt etwas differieren, da wir die Kosten der Speicherverwaltung mit einbeziehen. Wir könnten jedoch feststellen, daß die Einführung der statischen Speicherverwaltung die Gesamtlaufzeit um etwa 10 % verbessert hat.

Wir beginnen unsere Aufstellung mit Routinen die allein die Matrizen behandeln, beschreiben dann die, die Matrizen und Vektoren benutzen und zuletzt die, die zur Polynomarithmetik gehören.

3.3.1 Matrixoperationen

Zu den reinen Matrixoperationen gehören die Routinen

- matvondat($< datei >$)
  baut beim sequentiellen Einlesen aus einer Datei $< datei >$ die innere Struktur der Matrix auf. Ein Zeiger auf die erzeugte Matrix wird zurückgeliefert. Dazu benötigt die Routine etwa $O(w)$ Körperoperationen.

- mat_spaltenstreich(A, i)
  In jeder Matrixzeile der Matrix A wird ein Matrixeintrag mit Position i gesucht. Die Spalteneinträge ab dieser Position werden um eins erniedrigt. Dies führt zu einer Laufzeit von $O(w)$.

- mat_zeilenstreich(A, i)
  Wir löschen die i-te Zeile aus der Matrix A, indem wir die Zeiger auf die folgenden Zeilen, um eine Position nach oben wandern lassen. D.h. die frühere j-te Zeile ist jetzt die (j-1)-te Zeile ab Zeile i. Wir benötigen hierzu $n - i$ Körperoperationen also im schlechtesten Fall $n - 1$.

- mat_spaltentausch(A, i, B, j)
  Wir überschreiben die i-te Spalte von A mit der j-ten Spalte von B, indem wir die Einträge überschreiben. Zum Suchen der entsprechenden Einträge benötigen wir etwa w Körperoperationen. Diese Routine verwenden wir bei unterbestimmten Gleichungssystemen, um die noch nicht untersuchte j-te Spalte der Matrix B mit der in A linear abhängigen Zeile i zu vertauschen.

- mat_zuweisung(A)
  legt eine identische Kopie der Matrix A an. Diese Routine kostet uns $O(w)$.

- quadmat(A)
  Sei n das Minimum von Zeilenindex und Spaltenindex, so schneidet diese Routine eine quadratische Matrix der Dimension n aus. Diese Routine hat somit eine Laufzeit von $O(w)$.

- transmat(A)
  erzeugt die Transponierte zur Matrix A. Sie kann in etwa w Körperoperationen erzeugt werden.

Die Basis dieser Algorithmen sind die Algorithmen zum Einfügen, Streichen oder Suchen in linearen Listen. Sie haben eine Laufzeit von $O(n)$, wenn wir unter n die Länge der Liste verstehen. Wir möchten hier nicht näher auf diese Algorithmen eingehen, da sie in einer Standartvorlesung über Datenstrukturen und Algorithmen behandelt werden. Wir werden sie als bekannt voraussetzen.

3.3.2 Routinen der Vektoren

In diesem Abschnitt stellen wir die Implementierung der Funktionen rein auf Vektoren vor.

- einheitsvek(vek, n, i)
  Diese Routine erzeugt einen Vektor der Länge n mit einem einzigen Eintrag ungleich Null an der Stelle i. Dieser Eintrag hat den Wert 1. Ist $i = 0$, so erzeugt diese Routine den Nullvektor, daher verwenden wir sie oft zum Initialisieren von Vektoren. Diese Funktion verursacht Kosten von $O(n)$.

- komulvek($k, v, prod, n$)
  berechnet die skalare Multiplikation des Körperselementes $k \in K$ mit dem Vektor v der Länge n. Jeder Eintrag v_i wird mit k mittels Körpermultiplikation berechnet. Das Ergebnis wird in prod zurückgeliefert. Dieser Routine benötigt dazu etwa n Körperoperationen.

- vek_vgl(x, y, n)
  In dieser Routine untersuchen wir die Einträge der Vektoren x, y. Sind alle $x_i = y_i$ für $i = 1, \cdots, n$, so gibt sie 0 zurück. Gibt es ein i, sodaß $x_i \neq y_i$, so liefert die Routine die Position des kleinsten Eintrags $x_i \neq y_i$ zurück. Sie benötigt dazu $O(n)$ Zeit.

- random(v, n)
  erzeugt einen Vektor v der Länge n. Die Vektoreinträge sind Zufallswerte aus dem Körper K. Zum Setzen dieser Einträge benötigt random() n Körperoperationen.

- vek_zuweisung(x, y, n)
  Wir legen eine Kopie des Vektors x unter y an. Die Vektoren haben die Länge n. Zum Anlegen der Kopie benötigen wir n Körperoperationen.

- vekadd(x, y, z, n)
  Wir ermöglichen mit dieser Routine die koeffizientenweise Addition der Vektoreinträge der Vektor x mit y. Das Ergebnis steht in z. Um die Addition zweier Vektoren der Länge n ausführen zu können, benötigen wir n Körperoperationen.

- skalarprod(x, y, z, n)
  berechnet zu den Vektoren x, y der Länge n das Skalarprodukt nach folgender Vorschrift:

  $$z = \sum_{i=1}^{n} x_i * y_i$$

  Die Routine benötigt dazu Zeit $O(n)$.

- vekvondat($datei, n$)
liest aus der Datei *datei* einen Vektor der Länge n ein. Dazu benötigt vekvondat() n Körperoperationen.

- vekausgabe(x, n)
gibt den Vektor x der Länge n auf dem Bildschirm aus. Dafür benötigt die Routine Zeit $O(n)$.

- vekaufdat(x, n)
ist das Äquivalent zu vekausgabe(). Sie schreibt den Vektor x jedoch in eine Datei. Die Laufzeit beträgt daher auch $O(n)$.

3.3.3 Operationen zwischen Matrizen und Vektoren

Wir haben bis jetzt die Operationen auf Matrizen und Vektoren betrachtet. Nun wollen wir die Routinen vorstellen, die mit beiden agieren.

- mat_zeilentausch(A, i, d)
Wir überschreiben bei Ausführung dieser Routine die i-te Zeile der Matrix A mit den Einträgen des Vektors d. Wir benötigen dazu maximal n Körperoperationen.

- matmulvek(A, vek, erg)
erlaubt uns eine Matrix A mit einem Vektor *vek* zu multiplizieren. Dies geschieht nach der Vorschrift $erg_j = \sum_{i=1}^{n} a_{ij} * v_i$ für alle $j = 1, \cdots, m$. Sie benötigt w Körperoperationen.

- suchevek(A, i)
extrahiert den i-ten Zeilenvekor aus der Matrix A. Sie benötigt dazu Zeit $O(n)$.

- suchespalte(A, i)
liefert den i-ten Spaltenvektor zurück. Wir suchen dabei in jede Zeile einen Eintrag mit Spaltenindex i. Dies führt zu einer Laufzeit von $O(w)$.

3.3.4 Routinen der Polynomarithmetik

Jetzt fehlen nur noch die Routinen der Polynomarithmetik, dieser werden wir jetzt betrachten.

- poly_zuweisung(f, g)
Diese Funktion liefert uns eine Kopie des Polynoms g unter dem Namen f an. Ist g eine Polynom von Grad n, so sind $n + 1$ Körperoperationen dazu auszuführen.

- poly_add(f, g, z)
ist ähnlich der Vektoraddition. Die Koeffizienten der Polynome f und g werden komponentenweise addiert. Der Grad des Ergebnispolynom z ist dann $\max(\mathrm{grad}(p)), \mathrm{grad}(g)) = n$. Hieraus resultiert die Laufzeit von $O(n)$.

- poly_mult(f, g, z)
Die Multiplikation zweier Polynome berechnet sich folgendermaßen:

$$f * g = \sum_{i=0}^{n} \sum_{j=0}^{m} f_i * g_j * x^{i+j}$$

Wir erhalten ein Polynom $z = f * g$ mit $grad(n+m)$. Die Routine poly_mult() berechnet z in $O(n^2)$.

- poly_tausch(g, f)
liefert uns eine Polynom g dessen Koeffizienten die multiplikativen Inversen des Polynoms f sind. Die multiplikativen Inversen werden mit dem Euklidischen Algorithmus [Knu81] berechnet. Durch die Beschränkung auf den endlichen Körper K, beträgt die Laufzeit des Euklidischen Algorithmus $O(1)$, sodaß die Laufzeit von poly_tausch() durch $O(n)$ beschränkt ist.

Kapitel 4

Testergebnisse

Wir haben uns bis jetzt mit der Theorie beschäftigt. Um diese zu untermauern führten wir ein Vielzahl von Testläufen durch, um zu sehen wie nah sich Theorie und Praxis kommen. Laufzeitangaben werden wir, wenn nicht anders angegeben, in MIPS (Mega Instructions Per Second) angeben.

4.1 Der deterministische Wiedemann

Wir vergleichen hier drei Versionen des Wiedemannalgorithmus. Sie sind abgeleitet von Algorithmus 2.29. Version 1 entspricht diesem Algorithmus, Version 2 testet in jedem Schleifendurchlauf, ob wir das Minimalpolynom schon berechnet haben. Version 3 modefiziert Version 2 derart, daß die Einheitsvektoren nicht mehr in numerischer Reihenfolge, sondern in Abhängigkeit des Fehlers, der in der Berechnung auftritt, benutzt werden.

Wir testeten verschiedene Gleichungssysteme mit den drei Versionen. Hieraus resultiert die nachfolgende Tabelle:

Version 1	Version 2	Version 3
27182	3058	780
26601	3180	799
25839	3022	773
26591	3108	782
25180	7236	874
25170	7223	772
25629	7471	780
25184	7235	770
24761	743	498

Version 1	Version 2	Version 3
26889	1480	704
26658	1457	692
26033	1439	677
26946	1463	703

Betrachten wir den Mittelwert obiger Daten, so zeigt sich, daß Version 2 nur noch 14 % der Laufzeit des ersten Algorithmus benötigt, Version 3 benötigt nur noch 2,8 % im Vergleich zum Ausgangsalgorithmus.

Ausgehend von Algorithmus Version 3 vergleichen wir unsere Ergebnisse mit dem probabilistischen Ansatz des Wiedemannalgorithmus.

4.2 Vergleich: Wiedemann deterministisch – Wiedemann probabilistisch

Wir vergleichen die Laufzeiten der Algorithmen 2.29 und 2.3.2. Wir führten dazu Tests auf Matrizen unterschiedlicher Dimension und variirender Anzahl von Einträgen aus.

Dimension	Einträge	Alg. 2.31	n	Alg. 2.36	n	wiededeterm	wiedeprob
1000	1527	2031	7	2470	3	6640	6941
	2858	3150	4	3130	2	13790	12677
	3513	3699	4	2735	1	16231	14165
	5091	3885	2	3516	1	16231	14165
2000	1345	3575	9	4023	1	9368	7790
	2429	5247	10	4274	1	8492	10574
	2972	6073	7	6942	2	11836	10627
	3592	6076	7	6951	2	25398	14256
	4912	9232	6	7199	1	17021	18906
	5588	16040	6	9449	1	31749	18671
	7021	14777	4	11054	1	25400	21604
	7799	13691	3	11859	1	25751	23356
	8640	11006	3	11159	1	20719	21200

Die Kosten für beide Versionen sind etwa gleich. In verschiedenen Fällen ist der deterministische schneller, in anderen der probabilistische. Aufgrund dieser Testergebnisse nehmen wir an, das der deterministische Wiedemannalgorithmus 2.31 im Mittel $O(\log(n))$ Schleifendurchläufe benötigt, wie Wiedemann es für den probabilistischen angegeben hat.

Um eine Aussage zu treffen, ob der probabilistische oder der deterministische Ansatz der schneller ist, müßten wir weitere Tests mit Matrizen höherer Dimension durchführen. Dies war auf dem uns zur verfügungstehenden Plattenplatz (crypt5:

/TMP 91399 kByte) nicht möglich wegen des quadratisch mit der Dimension wachsenden externen Speicherbedarfs des Algorithmus 2.29.

4.1. Beispiel Externer Speicherplatzbedarf

Die Angabe des Speicherplatzbedarfs erfolgt in kByte

Dimension	Speicherplatz
1000x1000	8000
2000x2000	32000
3000x3000	72000
4000x4000	128000

Wir sehen an der Anzahl der Schleifendurchläufe für den deterministischen wie dem probabilistischen Algorithmus, daß ein Schleifendurchlauf in deterministischen Algorithmus weniger kostet als im probabilistischen. Dies führen wir darauf zurück, daß zur Bestimmung der Eingabefolge S des Berlekamp-Massey-Algorithmus die Bestimmung des Skalarproduktes nicht explizit ausgeführt werden muß, sondern der k-te Eintrag des Vektors $A^i * b$ ist schon das Folgenglied s_i. Wir sehen jedoch auch, daß wir meist mehr als doppelt soviele Schleifendurchläufe benötigen.

Wenden wir unsere Erfahrungen aus diesem Test auf die Algorithmen ohne Vorberechnung an, so ist der probabilistische Ansatz gebenüber dem deterministischen im Vorteil. Wir haben einen Kostenmehraufwand pro Schleife von $n * w$ gebenüber einer Verbesserung von $n * n$ durch die Berechnung der Skalarprodukte.

Aufgrund dieser Erfahrungen beschäftigten wir uns jetzt überwiegend mit dem probabilistischen Ansatz des Wiedemannalgorithmus ohne Vorberechnungen.

4.3 Wiedemannalgorithmus – Aktuelle Version

Wir betrachten zunächst den quadratischen Fall. Wir können die theoretisch berechnet Laufzeit von

$$O(\log(n) * (w * n + n * \log(n) * \log(\log(n))))$$

für Algorithmus 2.36 und die Abhängigkeit der Laufzeit von 2.40 vom Rang der Koeffizientenmatrix in der Praxis bestätigen.

Die folgende Tabelle bestätigt unsere Aussage:

n	w_1	$time_{w_1}$	w_2	$time_{w_2}$
100	5177	122	11651	133
200	5324	437	11890	457
300	5325	855	11893	1058
400	5435	1892	12286	2288
500	5396	3038	12132	3701
600	5455	5853	12128	6600
700	5457	7008	12208	8208
800	5408	8618	12176	10915

Ändern wir die Anzahl der Matrixeinträge w bei fester Dimension, so zeigt sich die Richtigkeit unserer Aussage. Folgende Gleichung ist erfüllt:

$$time_{w_2} \leq time_{w_1} * 1,2$$

Verdoppeln wir die Dimension n bei konstanter Anzahl von Matrixeinträgen w, so quadriert sich die Laufzeit. Dies sehen wir sehr deutlich, wenn wir die Dimensionen $n = 100$, 200, 400 und 800 betrachten.

Für die Routine 2.40 zeigt sich, wie erwartet, daß die Laufzeit stark vom Rang der Koeffizientenmatrix abhängt.

Nachfolgende Tabelle zeigt uns das Verhalten des Algorithmus 2.40

n	w	MIPS	$n-r$	ELC
1000	5435	21998	1	17 min
	5471	849936	34	11 h 14 min
	5386	48288	6	39 min
	5458	124379	8	1 h 39 min
2000	12233	125752	1	1 h 40 min
	12152	13588023	93	7 Tage 12 h
	12283	205415	2	2 h 43 min
3000	19057	46843	2	37 min
4000	26856	412073	1	5 h 27 min

Für den Fall der Rechteckmatrizen als Koeffizientenmatrix eines Gleichungssystems lassen sich aus den Daten nur schwer Rückschlüsse ziehen, da sie stark von den Laufzeiten der Unterroutinen abhängen. Eine Rechteckmatrix von Maximalgrad kann hochsinguläre Hauptunterminoren haben. Dies schlägt sich negativ auf die Laufzeit von 2.40 aus. Desweiteren ist sie abhängig von der Form der Eingabematrix.

Die Erfahrung hat uns gezeigt:
Der Wiedemannalgorithmus ist besonders geeignet für Gleichungssysteme, deren Koeffizientenmatrix möglichst viele reguläre Hauptunterminoren hat. Dies hat sich besonders im Fall der Rechteckmatrizen bemerkbar gemacht.

4.4 Ausblick

Der von uns implementierte Algorithmus basiert auf einer Körperarithmetik der maximalen Wortlänge von 32 Bits. Dadurch ist unsere Implementierung beschränkt auf endliche Körper der Dimension $p < 2^{32}$.
Um diese Beschränkung zu aufzuheben, benötigen wir eine neue Körperaritmetik.

Das größte von uns bis jetzt verwendete Gleichungssystem hat eine quadratische Koeffizientenmatrix der Dimension $n = 50000$ mit $w = 441454$ Einträgen. Dazu brauchen wir 13656 kByte Hauptspeicher. Hieraus wirft sich die Frage auf, wie wir den Hauptspeicherbedarf senken können.
Wie muß die Datenstruktur der Matrix aussehen, um sie teilweise aus dem Hauptspeicher auslagern zu können?
Wie können die Routinen möglichst effizient der neuen Datenstruktur angepaßt werden?

Ein weiterer interessanter Aspekt ist die Parallelisierung des Wiedemann-Algorithmus. Die Vorgehensweise zur Bestimmung des Minimalpolynoms legt diese Idee nahe. Wir können f^A bestimmen, indem wir Polynome f_u^b und daraus das kleinste gemeinsame Vielfache bestimmen. *Wirkt sich diese Parallelisierung positiv aus?*

Bei der Beschäftigung mit dem Wiedemannalgorithmus machten wir die Erfahrung, daß die Laufzeit sehr stark von der Form der Koeffizientenmatrix abhängt.
Ist die Matrix regulär, so brauchen wir den Algorithmus 2.36 nur einmal auszuführen.
Ist die Matrix singulär, so ist die Laufzeit abhängig von der Lage der linear abhängigen Zeilen oder Spalten.

Sei A eine $n \times n$-Matrix vom $\text{rang}(A) = r < n$:
Sind linear abhängige Zeilen schon unter den ersten Zeilen zu finden, sind mehr Zeilen- und Spaltenstreichoperationen auszuführen. Dies ist dadurch bedingt, daß schon früh Vektoren $vi = A^i * b$ erzeugt werden, die von dem vorangehenden linear abhängig sind und somit eine Polynom mit konstanten Koeffizienten 0 erzeugen. Die Wahrscheinlichkeit ist gering, daß das bestimmte Polynom schon das Minimalpolynom ist.
Stehen in den ersten r Zeilen linear unabhängige Zeilenvektoren, so wird das Minimalpolynom berechnet, schon das Ausführen eines Aufrufs von Algorithmus 2.36 liefert uns eine Lösung, wenn eine existiert.
Diese Behauptung scheint auch für Rechteckmatrizen zu halten.

Gibt es einen Algorithmus, der obige Transformation in annehmbarer Zeit ausführen kann?

Anhang A

Bezeichungen

Die folgenden Bezeichnungen treten in dieser Arbeit immer wieder auf:

- K – bezeichnet den Grundkörper über dem wir rechnen
- p – ist eine Primzahl. Sie gibt die Anzahl der Körperelemente in $K = \mathbb{F}_p$ an.
- $\bmod (p)$ – bezeichen dem kanonischen Homomorphismus von $\mathbb{Z}$ nach $\mathbb{Z}/p\mathbb{Z} \cong \mathbb{F}_p$, der jedem Element aus $\mathbb{Z}$ seine Restklasse zuordnet.
- A, x, b – A bezeichnet die Koeffizientenmatrix des Gleichungssystems $A*x = b$. b ist der Lösungsvektor und x bezeichne die Unbestimmte.
- n, m – gibt die Anzahl der Zeilen bzw. der Spalten der Matrix A an.
- w, w_i – ist die Anzahl der Einträge ungleich Null der Matrix A.
- a_i, a_{ij} – a_i sind die Vektoren der Matrix A, die Koeffizienten von A bezeichnen wir mit a_{ij}.
- $\mathrm{rang}(A)$ – bestimmt uns den Rang zur Matrix A.
- $\det(A)$ – berechnet die Determinante der Matrix A.
- $<,>$ – steht synonym für das Skalarprodukt zweier Vektoren.
- $\mathrm{span}(E)$ – Sei E eine Menge von Vektoren. $\mathrm{span}(E)$ bezeichnet dann den Vektorraum, der durch die Vektoren aus E aufgespannt wird.
- $\dim(V)$ – gibt die Dimension der Vektorräumen V an.
- f, g – bezeichnen Polynome

- grad(f) – gibt die Position des höchsten Koeffizienten ungleich Null des Polynoms f an.

- $f_u^b(X)$ – wird durch den Berlekamp-Massey-Algorithmus 2.10 erzeugt. Es erfüllt die Bedingung $u * f(A) * b = 0$.

- $f^b(X)$ – ist das Polynom, das die Bedingung $f(A) * b = 0$ erfüllt.

- $f^A(X)$ – bezeichnet das Minimalpolynom zur Matrix A.

- kgV(f, g) – berechnet uns das kleinste gemeinsame Vielfache der Polynome f, g.

- $\min(k_1, k_2)$ – liefert uns das Minimum der Körperelemente k_1, k_2.

- $\max(k_1, k_2)$ – liefert uns das Maximum der Körperelemente k_1, k_2.

Literaturverzeichnis

[Hun74] Thomas W. Hungerford. *Algebra*, volume 73 of *Graduate Texts in Mathematics*. Springer, 1974.

[Knu81] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 2 edition, 1981.

[Lam78] Erich Lamprecht. *Einführung in die Algebra*. UTB Mathematik. Birkhäuser, 1978.

[Lan84] Serge Lange. *Algebra*. Addison Wesley, 2 edition, 1984.

[Mas69] James L. Massey. Shift–Register Synthesis and BCH Decoding. *IEEE Tranactions on Information Theory*, 15(1):6, January 1969.

[Meh84] Kurt Mehlhorn. *Sorting and Searching,*, volume 1 of *Data Structures and Algorithms*. Springer Verlag, 1984.

[Rei91] Heinrich Reinhardt, Fritz Soeder. *dtv-Altas zur Mathematik*, volume 1. Deutscher Taschenbuch Verlag GmbH & Co. KG, München, 9 edition, November 1991.

[Rei92] Heinrich Reinhardt, Fritz Soeder. *dtv-Atlas zur Mathematik*, volume 2. Deutscher Taschenbuch Verlag GmbH & Co. KG, München, 8 edition, Januar 1992.

[Sch77] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977.

[Tar79] Andrew Chi-Chih Tarjan, Robert Endre Yao. Storing a Sparce Table. *Communucations of the ACM*, 1979.

[Wie86] Douglas H. Wiedemann. Solving Sparse Linear Equations Over Finite Fields. *IEEE Transactions on Information Theory*, 32, 1986.