Universität Mannheim

Fakultät für Informatik

Lehrstuhl für Theoretische Informatik

Diplomarbeit

Faktorisieren mit dem General Number Field Sieve

Verfasser:
Christian Stöffler
11. Januar 2007

Betreuer:
PD Dr. Stefan Lucks

Zusammenfassung

Als Grundlage moderner Public-Key-Kryptosysteme wird meistens die Annahme zu Grunde gelegt, dass ein oder mehrere Probleme nicht in vertretbarer Zeit berechenbar sind. Beispiele solcher Probleme sind der diskrete Logarithmus, Quadratrische Reste oder das Faktorisieren großer Primzahlprodukte, wobei letzteres das Thema dieser Arbeit ist.

Um die Sicherheit eines Kryptosystems zu beurteilen, muss dieses sinnvollerweise am besten bekannten Algorithmus für das zu Grunde liegende Problem gemessen werden. Im vorliegenden Fall handelt es sich dabei um das General Number Field Sieve (GNFS), eine Weiterentwicklung der Ideen des Quadratischen Siebs (QS). Der Algorithmus verwendet außerdem Algorithmen zum Lösen gigantischer linearer Gleichungssysteme wie z.B. den Block-Lanczos-Algorithmus. Daher wird auch dieses Thema von mir betrachtet.

Im Folgenden werde ich zunächst die notwendigen mathematischen und informatischen Grundlagen darstellen. Anschließend gehe ich auf den Lanczos-Algorithmus ein, da dieser sowohl im QS als auch im GNFS benutzt wird. Danach beschreibe ich das QS, da man dort viele der wesentlichen Eigenschaften im GNFS wiederfindet. Schließlich werde ich die Arbeitsweise des GNFS betrachten und analysieren und zuletzt die Ergebnisse meiner Implementierung und Tests zeigen. Dadurch werden Probleme deutlich, die man beim reinen Betrachten der Theorie nicht unbedingt direkt sieht.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Eine Zahl n zu faktorisieren bedeutet, dass man Primzahlen $p_1, ..., p_k$ und Exponenten $e_1, ..., e_k$ sucht, sodass $n = p_1^{e_1} \cdot \dots \cdot p_k^{e_k}$ gilt. Genau genommen genügt es bereits, eine der Zahlen finden zu können, z.B. p_i, da man die anderen dann einfach erhält, indem man n durch p_i teilt und von vorne beginnt. In dieser Arbeit geht es ausschließlich darum, eine Zahl $n = p \cdot q$ zu faktorisieren, wobei p, q Primzahlen sind, die in etwa die gleiche Größenordnung haben.

Um im Internet sichere Kommunikation zu betreiben, werden Public-Key-Kryptosysteme eingesetzt. Das bekannteste unter diesen ist das RSA-Kryptosystem, dessen Sicherheitsgrundlage ist, dass das Berechnen e-ter Wurzeln modulo n nicht in vertretbarer Zeit möglich ist, falls n das Produkt zweier großer Primzahlen ist. Da das Berechnen dieser RSA-Wurzeln modulo einer Primzahl p einfach ist,[1] beruht die Sicherheit von RSA insbesondere auf dem Problem der Faktorisierung von n.

Falls man die Faktorisierung von n berechnen könnte, so könnte man also auch eine mit RSA gesicherte Kommunikation direkt im Klartext mitlesen. Besonders bedenklich bezüglich der Sicherheit ist, dass per RSA normalerweise Schlüssel ausgetauscht werden, die anschließend für symmetrische Verschlüsselung (z.B. AES) benutzt werden. Es reicht also bereits eine einzige Wurzel zu ziehen, um anschließend eine ganze Kommunikationsserie mitlesen zu können. Deswegen erfordert dieser erste Schritt eine besonders hohe Sicherheit gegen Faktorisierung.

Ein weiterer Anreiz ist die Suche nach Faktorisierungen immer größerer Zahlen. Im Rahmen der RSA-Challenge wurden z.B. Preisgelder von bis zu $200.000 auf Faktorisierungen ausgesetzt. Außerdem gibt es Listen der „most wanted“ Faktorisierungen, z.B. die Cunningham-Tables und das GIMPS-Projekt.

[1] siehe 2.2.4

Es existieren neben Siebalgorithmen zahlreiche andere Ideen zum Faktorisieren von Zahlen. Der wichtigste der anderen Algorithmen ist die Elliptic Curves Method (ECM), die besonders gut dazu geeignet ist Zahlen zu faktorisieren, die kleine Primfaktoren enthalten. Im Fall von zwei etwa gleich großen multiplizierten Primzahlen hingegen ist dieser Algorithmus besonders ineffizient, was ihn z.B. für die RSA-Challenge ungeeignet macht. Allerdings kann man die ECM an manchen Stellen im GNFS verwenden, wenn bekannt ist, dass Zahlen faktorisiert werden müssen, die kleine Fakoren enthalten. Dies führt zu einer Verbesserung der Laufzeit gegenüber trivialer Division.

Ziel dieser Arbeit ist dem Leser zu vermitteln, welche Ideen und mathematische Grundlagen hinter dem GNFS stecken. Es soll detailliert und vor allem vollständig erarbeitet werden, welche Schritte erforderlich sind, um eine Zahl mit dem GNFS zu faktorisieren. Das sehe ich als besonders wichtig an, weil ich häufig lesen musste, dass bestimmte Teilprobleme des Algorithmus „einfach“ gelöst werden können, aber nicht wie genau.[2] Dieser Aspekt wird vor allem durch den praktischen Teil der Arbeit gewährleistet, da am Ende ein lauffähiges Programm Zahlen nur mit den in dieser Arbeit besprochenen Verfahren faktorisiert.

1.2 Aufbau der Arbeit

Beim Verfassen dieser Arbeit musste ich feststellen, dass es sich in den einzelnen Kapiteln um sehr aufwändige und schwierige Themen handelt, zu denen es unter Anderem zahlreiche Dissertationen, gibt die jeweils wieder unterschiedliche Aspekte der Teilthemen betrachten. Daher kann ich unmöglich auf alle Verbesserungen und Beweise eingehen, die es gibt, sondern ich lege besonderen Wert auf das Vermitteln der Ideen der jeweiligen Algorithmen.

Außerdem ist mir aufgefallen, dass in manchen Quellen eher mathematisch im Satz-Beweis-Schema vorgegangen wird und in anderen Arbeiten eher aufbauend mit eingestreuten Begründungen. Letztere Quellen konnte ich wesentlich leichter verstehen, bei den anderen hingegen habe ich lange gebraucht um überhaupt zu merken, dass es sich im Prinzip um das gleiche handelt, nur viel komplizierter. Deswegen verzichte ich so weit wie möglich auf das Zerlegen der Abläufe in einzelne Theoreme und Beweise, um besser nachvollziehen zu können, welche Schritte aus welchem Grund nacheinander ausgeführt werden.

Desweiteren gebe ich die Laufzeit der Algorithmen meistens nur mit der wichtigsten Idee zur Analyse an, da über dieses Thema abermals umfangreiche Arbeiten existieren.

[2] z.B.wird in [23] nur auf Quellen verwiesen, die sich mit dem Lösen von Gleichungssystemen beschäftigen.

Da mir persönlich Pseudocode zum Verständnis nicht mehr hilft als Quellcode, gebe ich die Kernideen der vorgestellten Algorithmen gleich in C/C++ Syntax an. Außerdem werden diese zusätzlich noch im Text beschrieben, wodurch besonders die Ideen des Algorithmus herausgehoben werden sollen.

Besonderen Wert lege ich auf den praktischen Teil dieser Arbeit, in dem ich zumindest eine lauffähige „proof-of-concept"-Version implementiert habe die belegt, dass die Algorithmen auch wirklich funktionieren. Dabei steht die Performance natürlich etwas im Hintergrund, es sind dafür einige Fehler in der Arbeit aufgetaucht, z.B. beim Berechnen der Algebraischen Wurzel in Kapitel 5.4.2, die beim reinen Bearbeiten der Theorie wahrscheinlich nicht aufgefallen wären.

Die Arbeit gliedert sich in die folgenden Kapitel:

- **Grundlagen** - Hier werden die zum Verständnis der Algorithmen notwendigen mathematischen Begriffe und Definitionen zusammengetragen. Außerdem beschreibe ich noch einige wichtige Dinge aus dem Bereich der Informatik, wo unter anderem nochmals verdeutlicht wird, wieso die Faktorisierung eines RSA-Modulus ein gravierendes Problem ist.

- **Lanczos-Algorithmus** - In den Siebalgorithmen müssen riesige, dünn besetzte Gleichungssysteme über $\{0, 1\}$ gelöst werden. Da ab einer gewissen Größe der Gauss-Algorithmus nicht mehr effizient genug ist, wurden andere Ideen entwickelt, die vor allem zum Block-Lanczos- und dem Block-Wiedemann-Algorithmus führten. Den ersten dieser beiden Algorithmen werde ich in Kapitel 3 vorstellen.

- **Quadratisches Sieb** - Die Ideen des Quadratischen Siebs sind einfach nachzuvollziehen und führten schließlich zur Entwicklung des Special Number Field Sieve für spezielle Zahlen, das anschließend zum General Number Field Sieve für beliebige Zahlen erweitert wurde. Da die Siebidee elementar ist und anhand dieses Algorithmus sehr einfach verstanden werden kann, stelle ich diesen vor dem eigentlichen Kern der Arbeit vor.

- **General Number Field Sieve** - Dies ist der wichtigste und gleichzeitig komplexeste Teil der Arbeit. Es sollen die wesentlichen Ideen vermittelt sowie die wichtigsten Ansätze zur Verbesserung vorgestellt werden.

- **Implementierung** - Bei der Umsetzung der Algorithmen gibt es einige Dinge zu beachten. Daher widme ich dieser ein gesondertes Kapitel und beschreibe meine Programme und vor allem die aufgetretenen Probleme.

Die wichtigsten Quellen der einzelnen Kapitel werden jeweils am Anfang des Kapitels angegeben. Nur falls einzelne Abschnitte in einer anderen Quelle zu finden sind, gebe ich diese gesondert an. Einen guten Überblick über die Themen dieser Arbeit erhält man z.B. in [1] oder [2].

Da ich fast keine Vorgaben bezüglich der Form der Arbeit hatte, habe ich z.B. das Literaturverzeichnis auf das wesentliche beschränkt. Sehr interessant finde ich, dass ich fast alle Quellen im Internet finden konnte, sodass im Prinzip alle Informationen frei für jeden verfügbar sind.

Kapitel 2

Grundlagen

Die in dieser Arbeit untersuchten Algorithmen erfordern zahlreiche mathematische Grundlagen aus den Gebieten der algebraischen Zahlentheorie und der Linearen Algebra. Außerdem werden in der Kryptographie geläufige Dinge kurz vorgestellt, wie z.B. der Erweiterte Euklidische Algorithmus.

Die wichtigsten Quellen für dieses Kapitel sind [6], [3], [4] und [5].

2.1 Mathematische Grundlagen

Im folgenden Abschnitt werden einige Sätze und Definitionen vorgestellt, die im Laufe der Arbeit gebraucht werden. Sie werden als „Fakt“ vorgestellt, für die Beweise sei auf theoretischere Arbeiten und Lehrbücher verwiesen, z.B. [3].

2.1.1 Bezeichnungen

Die Menge der *Natürlichen Zahlen* ist $\mathbb{N}$, $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$.
Der Ring der *Ganzen Zahlen* ist $\mathbb{Z}$.
Der *Restklassenring* modulo n der ganzen Zahlen ist $\mathbb{Z}_n$.
Der Körper der *Rationalen Zahlen* ist $\mathbb{Q}$.
Der Körper der *Komplexen Zahlen* ist $\mathbb{C}$.
Die *multiplikative Gruppe* eines Restklassenrings $\mathbb{K}$ heißt $\mathbb{K}^*$.
p teilt n wird geschrieben als $p \mid n$, *teilt nicht* als $p \nmid n$.
Die $k \times k$*-Einheitsmatrix* heißt I_k.
Der zu x *Transponierte Vektor* wird mit x^T bezeichnet.
Die zu M *Transponierte Matrix* heißt M^T.
Der *natürliche Logarithmus* von x wird mit $\log(x)$ bezeichnet.
Für $x \in \mathbb{R}$ heißt $\lceil x \rceil$ aufrunden, $\lfloor x \rfloor$ abrunden und $[x]$ zur nächsten ganzen Zahl hin runden.

2.1.2 Zahlkörper

Da das GNFS, wörtlich übersetzt „Zahlkörpersieb“, Berechnungen über einem Zahlkörper durchführt, muss als erstes geklärt werden, was ein Zahlkörper überhaupt ist und welche Eigenschaften er hat.

Polynomring, Polynom, Wurzel

Sei $\mathbb{K}$ ein Körper und x eine Veränderliche über $\mathbb{K}$. Der *Polynomring* über $\mathbb{K}$ ist definiert als[1]

$$\mathbb{K}[x] := \{f_0 + f_1 \cdot x + ... + f_d \cdot x^d \mid d \in \mathbb{N}_0, f_i \in \mathbb{K}\}$$

Ein Element $f(x) = f_0 + f_1 \cdot x + ... + f_d \cdot x^d \in \mathbb{K}[x]$ mit $f_d \neq 0$ heißt *Polynom* vom *Grad* d. Ein Element $\rho \in \mathbb{K}$ mit $f(\rho) = 0$ heißt *Wurzel* von f.

Satz von Eisenstein

Sei $f(x) = f_0 + f_1 \cdot x + ... + f_d \cdot x^d \in \mathbb{K}[x]$ ein Polynom. f heißt reduzibel, falls zwei andere Polynome $g(x)$ und $h(x)$ existieren, sodass $f(x) = g(x) \cdot h(x)$, andernfalls heißt f irreduzibel.

Ein einfaches und effizientes Irreduzibilitätskriterium ist das folgende: f ist irreduzibel, falls ein Primelement $p \in K$ existiert, sodass gilt:

- $p^2 \nmid f_0$
- $p \nmid f_d$
- $\forall i = 0, ..., d-1$ gilt: $p \mid f_i$

Man sollte dabei allerdings beachten, dass dies ein hinreichendes, aber nicht notwendiges Kriterium ist. Es werden durchaus viele Polynome nicht als irreduzibel erkannt, obwohl sie dies sind.

Zahlkörper

Sei $\mathbb{K} \subseteq \mathbb{C}$ eine mindestens zweielementige Teilmenge der komplexen Zahlen. $\mathbb{K}$ heißt *Zahlkörper*, wenn für $\alpha, \beta, \gamma \in \mathbb{K}, \gamma \neq 0$ gilt:

1. $(\alpha - \beta) \in \mathbb{K}$
2. $(\alpha + \beta) \in \mathbb{K}$
3. $(\alpha \cdot \beta) \in \mathbb{K}$
4. $(\frac{\alpha}{\gamma}) \in \mathbb{K}$

[1] Abgesetzte Formeln werden in dieser Arbeit ohne Satzzeichen abgeschlossen.

Algebraische Zahl

Die Wurzel $\rho \in \mathbb{C}$ eines Polynoms

$$f(x) = f_0 + f_1 \cdot x + ... + f_d \cdot x^d \in \mathbb{Q}[x]$$

heißt *Algebraische Zahl*. Falls f irreduzibel und vom Grad d ist, bezeichnet man d auch als Grad von $\rho = \rho^{(1)}$. Die weiteren komplexen Wurzeln $\rho^{(2)}, ..., \rho^{(d)}$ heißen die *Konjugierten* von ρ.

Der Zahlkörper $\mathbb{Q}(\rho)$

Sei ρ eine Algebraische Zahl vom Grad d. Dann bildet die folgende Menge einen Zahlkörper:

$$\mathbb{Q}(\rho) = \{\alpha | \alpha = a_0 + a_1 \cdot \rho + ... + a_{d-1} \cdot \rho^{d-1}, a_i \in \mathbb{Q}, i = 0, ..., d-1\}$$

Ganze Algebraische Zahlen

Ist $f(x) = f_0 + f_1 \cdot x + ... + f_d \cdot x^d \in \mathbb{Z}[x]$ irreduzibel über $\mathbb{Q}$ und ρ eine Wurzel von $f(x)$, dann heißt ρ *Ganze Algebraische Zahl* vom Grad d. Die Menge $\mathcal{O}_\rho$ der Ganzen Algebraischen Zahlen in $\mathbb{Q}(\rho)$ bildet einen kommutativen nullteilerfreien Ring mit 1, genauso die Menge

$$\mathbb{Z}[\rho] := \{a | a = a_0 + a_1 \cdot \rho + ... + a_{d-1} \cdot \rho^{d-1}, a_i \in \mathbb{Z}, i = 0, ..., d-1\} \subseteq \mathcal{O}_\rho \subseteq \mathbb{Q}(\rho)$$

Konjugierte

Sei $\alpha = a_0 + a_1 \cdot \rho + ... + a_{d-1}\rho^{d-1} \in \mathbb{Q}(\rho)$. Dann sind

$$\alpha^{(i)} = a_0 + a_1 \cdot \rho^{(i)} + ... + a_{d-1}\rho^{(i)^{d-1}}, i = 2, ..., d$$

die Konjugierten der Ganzen Algebraischen Zahl α.

Norm

Für eine Algebraische Zahl α vom Grad d heißt die Zahl

$$N(\alpha) = \prod_{i=1}^{d} \alpha^{(i)}$$

die *Norm* von α.

Ganzheitsbasis

Sei ρ eine Algebraische Zahl vom Grad d. Eine Menge von d Zahlen

$$\{w_1, ..., w_d | w_i \in \mathcal{O}_\rho\}$$

heißt *Ganzheitsbasis* von $\mathbb{Q}(\rho)$, wenn sich jede Zahl $\alpha \in \mathcal{O}_\rho$ eindeutig in der Form

$$\alpha = \sum_{i=1}^{d} z_i \cdot w_i, z_i \in \mathbb{Z}$$

darstellen lässt.

Körperdiskriminante

Sei $\{w_1, ...w_d\}$ eine Basis von $\mathcal{O}_\rho$ und sei $w_{i,j} = w_j^{(i)}$ die i-te Konjugierte von w_j. Dann heißt das Quadrat der Determinante der Matrix $(w_{i,j})$

$$d(\mathbb{Q}(\rho)) := det(w_{i,j})^2$$

Diskriminante des Körpers $\mathbb{Q}(\rho)$.

Polynomdiskriminante

Für die Diskriminante von f gilt:

$$d(f) = (-1)^{\frac{d(d+1)}{2}} \cdot N(f'(\rho))$$

2.1.3 Idealtheorie

Im GNFS ist ein Teilalgorithmus enthalten, der Ideale in Primideale zerlegt. Daher sind im Folgenden die dazu erforderlichen Begriffe aus der Idealtheorie zusammengetragen.

Ideal

Eine Teilmenge $\mathcal{I} \subset \mathcal{O}_\rho$ heißt *Ideal* in $\mathcal{O}_\rho$, wenn für alle $\alpha, \beta \in \mathcal{I}$ und alle $\lambda, \mu \in \mathcal{O}_\rho$ gilt:

$$\lambda \cdot \alpha + \mu \cdot \beta \in \mathcal{I}$$

Seien $\alpha_1, ..., \alpha_k \in \mathcal{O}_\rho$. Dann nennt man

$$\mathcal{I} = \{\alpha | \alpha = \lambda_1 \cdot \alpha_1 + ... + \lambda_k \cdot \alpha_k, \lambda_i \in \mathcal{O}_\rho\}$$

das von $\alpha_1, ..., \alpha_k$ erzeugte Ideal und man schreibt

$$\mathcal{I} = \langle \alpha_1, ..., \alpha_k \rangle$$

Hauptideal

Ein Ideal $\mathcal{I}$ heißt *Hauptideal*, wenn es ein $\alpha \in \mathcal{I}$ gibt mit

$$\mathcal{I} = \langle \alpha \rangle$$

Basis eines Ideals

Sei $\mathcal{I}$ ein Ideal in $\mathcal{O}_\rho$, $\mathcal{I} \neq \langle 0 \rangle$. Dann existiert eine Menge

$$B = \{\alpha_1, ..., \alpha_d, \alpha_i \in \mathcal{O}_\rho\}$$

sodass jede Zahl

$$\alpha = z_1 \cdot \alpha_1 + ... + z_d \cdot \alpha_d \in \mathcal{I}$$

genau einmal dargestellt wird, wenn die Zahlen $z_i, i = 1, ..., d$ alle Zahlen aus $\mathbb{Z}$ durchlaufen. B heißt *Basis* von $\mathcal{I}$.

Norm eines Ideals

Sei $\mathcal{I}$ ein Ideal in $\mathcal{O}_\rho$. Sei weiter $\{w_1, ..., w_d\}$ eine Ganzheitsbasis von $\mathbb{Q}(\rho)$ und $\{\alpha_1, ..., \alpha_d\}$ eine Basis von $\mathcal{I}$ mit

$$\alpha_i = \sum_{j=1}^{d} a_{i,j} \cdot w_j, a_{i,j} \in \mathbb{Z}$$

Dann heißt der Absolutbetrag der Determinante der Matrix $(a_{i,j})$, $i, j = 1, ..., d$ *Norm* von $\mathcal{I}$:

$$N(\mathcal{I}) := |det(a_{i,j})|$$

Norm eines Hauptideals

Ist $\mathcal{I} = \langle \alpha \rangle$ ein Hauptideal in $\mathbb{Q}(\rho)$, dann gilt

$$N(\mathcal{I}) = |N(\alpha)|$$

Ist α eine Primzahl, so gilt

$$N(\mathcal{I}) = \alpha^d$$

Primideal

Seien $\mathcal{A} = \langle \alpha_1, ..., \alpha_r \rangle$ und $\mathcal{B} = \langle \beta_1, ..., \beta_s \rangle$ Ideale in $\mathbb{Q}(\rho)$. Unter dem *Produkt* der Ideale $\mathcal{A}$ und $\mathcal{B}$ versteht man das Ideal

$$\mathcal{A} \cdot \mathcal{B} := \langle \alpha_1 \cdot \beta_1, ..., \alpha_1 \cdot \beta_s, \alpha_2 \cdot \beta_1, ..., \alpha_r \cdot \beta_s \rangle$$

$\mathcal{A}$ heißt *teilbar* durch $\mathcal{B}$, wenn es ein Ideal $\mathcal{C}$ gibt mit

$$\mathcal{A} = \mathcal{B} \cdot \mathcal{C}$$

Ein Ideal $\mathcal{P}$ heißt *Primideal*, wenn gilt

1. $\mathcal{P} \neq \langle 1 \rangle$
2. $\mathcal{A} | \mathcal{P} \Rightarrow \mathcal{A} = \langle 1 \rangle$ oder $\mathcal{A} = \mathcal{P}$

Ideal und Norm

Sei $\mathcal{I}$ ein Ideal in $\mathcal{O}_\rho$.

1. Wenn $N(\mathcal{I})$ eine Primzahl ist, dann ist $\mathcal{I}$ ein Primideal.
2. $N(\mathcal{I}) \in \mathcal{I}$
3. $N(\mathcal{I}) = |\mathcal{O}_\rho / \mathcal{I}|$
4. Wenn $\mathcal{I}$ ein Primideal, dann gilt $p \in \mathcal{I}$ für genau eine Primzahl p und $N(\mathcal{I}) = p^m$ für ein $m \leq d$.

Fundamentalsatz der Idealtheorie

Jedes von $\langle 0 \rangle$ und $\langle 1 \rangle$ verschiedene Ideal in $\mathcal{O}_\rho$ lässt sich bis auf die Reihenfolge der Faktoren auf eindeutige Weise als Produkt von Primidealen darstellen.

Index von ρ in $\mathcal{O}_\rho$

Die Zahl $[\mathcal{O}_\rho : \mathbb{Z}[\rho]]$ mit

$$[\mathcal{O}_\rho : \mathbb{Z}[\rho]]^2 = \frac{d(f)}{d(\mathbb{Q}(\rho))}$$

heißt *Index* von ρ in $\mathcal{O}_\rho$.

Zerlegung von $p\mathcal{O}_\rho$

Sei p prim mit $p \nmid [\mathcal{O}_\rho : \mathbb{Z}[\rho]]$. Zerfällt $f(x)$ über $\mathbb{Z}_p$ in

$$f(x) = \prod_{i=1}^{r} f_i(x)^{e_i} \bmod p$$

wobei $f_i(x) \in \mathbb{Z}_p[x]$ irreduzibel über $\mathbb{Z}_p$ ist, dann zerfällt das Hauptideal $p\mathcal{O}_\rho$ in

$$p\mathcal{O}_\rho = \prod_{i=1}^{r} \mathcal{P}_i^{e_i}$$

wobei $\mathcal{P}_i$ das von $\langle p, f_i(\rho)\rangle$ erzeugte Primideal ist.

2.1.4 Lineare Algebra

Für den Lanczos-Algorithmus werden einige Dinge aus dem Gebiet der Linearen Algebra benötigt. Insbesondere Matrix- und Abbildungseigenschaften wie z.B. der Begriff „selbstadjungiert“ sind für das Verständnis wichtig.

Unterraum

Sei E ein Vektorraum und $F \subset E$. F heißt *Unterraum* von E, falls

- $\forall x, y \in F$ gilt: $x + y \in F$
- $\forall x \in F, \lambda \in \mathbb{R}$ gilt: $\lambda \cdot x \in F$

Aufgespannter Unterraum

Sei E ein Vektorraum und $x_1, ..., x_k \in E$. Dann ist

$$F := \{\lambda_1 \cdot x_1 + \cdots + \lambda_k \cdot x_k | \lambda_1, ..., \lambda_k \in \mathbb{R}\}$$

ein Unterraum von E. F heißt der von $\{x_1, ..., x_k\}$ erzeugte oder aufgespannte Unterraum von E und man schreibt auch

$$F := span\{x_1, ..., x_k\}$$

Matrix einer Abbildung

Sei E ein n-dimensionaler Vektorraum mit der Basis $B = \{b_1, ..., b_n\}$ und $l : E \longrightarrow E$ eine lineare Abbildung. Dann gibt es eine eindeutige Darstellung von $l(b_i)$ als Linearkombination:

$$l(b_i) = \sum_{i=1}^{n} \alpha_i \cdot b_i, \quad i = 1, ..., n$$

Die hierdurch definierte $n \times n$-Matrix

$$M(l) := \begin{pmatrix} \alpha_{1,1} & \cdots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{n,1} & \cdots & \alpha_{n,n} \end{pmatrix}$$

heißt Matrix von l bezüglich der Abbildung l.

Adjungierte Abbildung

Sei E ein n-dimensionaler Vektorraum und $l : E \longrightarrow E$ eine lineare Abbildung. Dann existiert genau eine lineare Abbildung $\tilde{l} : E \longrightarrow E$ mit

$$\tilde{l}(x) \cdot y = x \cdot l(y), \quad \forall x, y \in E$$

Diese Abbildung heißt die zu l *adjungierte Abbildung*.

Matrix der adjungierten Abbildung

Sei $M(l)$ die Matrix von $l : E \longrightarrow E$ und $M(\tilde{l})$ die Matrix von $\tilde{l} : E \longrightarrow E$. Dann gilt

$$M(\tilde{l}) = M(l)^T$$

Selbstadjungierte Abbildungen

Die Abbildung l heißt *selbstadjungiert*, falls

$$l(x) \cdot y = x \cdot l(y), \quad \forall x, y \in E$$

gilt, also falls $l = \tilde{l}$. Als direkte Folge daraus ergibt sich

$$M(l) = M(l)^T$$

also ist die Matrix $M(l)$ symmetrisch.

Lineares Gleichungssystem

Ein System von n linearen Gleichungen

$$\begin{array}{ccccc} a_{1,1} \cdot x_1 & + \cdots + & a_{n,1} \cdot x_n & = y_1 \\ \vdots & & \vdots & \vdots \\ a_{1,n} \cdot x_1 & + \cdots + & a_{n,n} \cdot x_n & = y_n \end{array}$$

oder kurz

$$\sum_{i=1}^{n} a_{i,j} \cdot x_i = y_j, \quad j = 1, ..., n$$

heißt Lineares Gleichungssystem (LGS) und ist äquivalent zu

$$A \cdot x = y$$

mit der Koeffizientenmatrix $A = (a_{i,j})$ sowie den Vektoren $x = (x_1, ..., x_n)^T$ und $y = (y_1, ..., y_n)^T$.

Rang einer Matrix

Der *Rang* einer Matrix M ist definiert als die Anzahl der linear unabhängigen Zeilenvektoren von M. Der Wert $rang(M)$ lässt sich berechnen, indem man die Matrix in obere Dreiecksform bringt und die Zeilen zählt, die nicht nur Nullen enthalten.

2.1.5 Sonstige Mathematik

In diesem Abschnitt folgen noch einige grundlegende Dinge aus dem Gebiet der Zahlentheorie, die vor allem in der Kryptographie oft benötigt werden.

Die Eulersche φ-Funktion

Die Funktion $\varphi(n)$ ist definiert als die Anzahl der Zahlen $x \leq n$, die zu n teilerfremd sind, also

$$\varphi(n) := |\{x \leq n \mid ggT(n, x) = 1\}|$$

Sei p eine Primzahl. Dann ist $\varphi(p) = p - 1$, da p nur durch 1 und sich selbst teilbar ist. Falls $n = p \cdot q$ mit p, q prim, dann ist

$$\begin{aligned} \varphi(n) &= \varphi(p \cdot q) \\ &= \varphi(p) \cdot \varphi(p) \\ &= (p-1)(q-1) \end{aligned}$$

Der kleine Satz von Fermat

Für die modulare Arithmetik ist dieser Satz von entscheidender Bedeutung. Die Aussage ist sehr einfach: Sei p eine Primzahl und $x \in \mathbb{Z}_p$. Dann gilt:

$$x^{p-1} = 1 \bmod p$$

Der Satz von Euler

Die Verallgemeinerung des Satzes von Fermat für beliebiges n ist

$$x^{\varphi(n)} = 1 \bmod n$$

wobei $\varphi(n)$ die Eulersche φ-Funktion ist und $ggt(n, x) = 1$. Eine wichtige Folgerung aus diesem Satz ist

$$x^a \bmod n = x^{a \bmod \varphi(n)} \bmod n$$

B-Glätte

Sei $B \in \mathbb{N}$ und $\mathcal{F}_k = \{p_1, p_2, ..., p_k | p_i > p_{i-1}, p_i \text{ prim}, p_i \leq B\}$ die aufsteigend sortierte Menge der Primzahlen $\leq B$. Eine Zahl n heißt B-*glatt*, falls n nur Primfaktoren $\leq B$ hat. Sei $S \subseteq F_k$ eine Teilmenge von F_k. Dann heißt n glatt bezüglich S, falls n nur Primteiler aus S enthält. F_k heißt auch Faktorbasis und ist in dieser Arbeit sortiert definiert, damit später einfachere Schreibweisen benutzt werden können.

Legendre-Symbol

Das *Legendre-Symbol* $\left(\frac{a}{p}\right)$ gibt an, ob eine Zahl $a \in \mathbb{Z}$ ein quadratischer Rest modulo einer ungeraden Primzahl p ist:

$$\left(\frac{a}{p}\right) := \begin{cases} 1, & \exists\, b \in \mathbb{Z} \text{ mit } b^2 = a \bmod p \\ -1, & \nexists\, b \in \mathbb{Z} \text{ mit } b^2 = a \bmod p \\ 0, & \text{falls } p|a \end{cases}$$

Es kann unter den Bedingungen dieser Arbeit berechnet werden durch

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \bmod p$$

2.2 Informatische Grundlagen

In diesem Abschnitt werden Algorithmen vorgestellt, die grundlegend für die Implementation der komplexeren Programme sind. Außerdem wird die Motivation der Arbeit anhand des RSA-Kryptosystems nochmals verdeutlicht.

2.2.1 Algorithmus von Euklid

Der *Algorithmus von Euklid* ist eigentlich dazu gedacht, den größten gemeinsamen Teiler (ggT) zweier Zahlen zu berechnen. Man kann ihn allerdings auch so erweitern, dass man mit ihm das Inverse einer Zahl modulo einer anderen berechnen kann:

```
Eingabe: p, q
Ausgabe: a = ggT(p,q) und Zahlen u, v mit: a = up + vq
a=max(p, q), b=min(p, q)
u=1, v=0, u'=0, v'=1
while(b!=0){
   c=a mod b, d=a div b
   u"=u-du', v"=v-dv'
   u=u', v=v', u'=u", v'=v", a=b, b=c
}
return(a, u, v)
```

Abbildung 2.1: Erweiterter Euklidischer Algorithmus

Ein Korrektheitsbeweis für diesen Algorithmus wird z.B. in [5] geführt. Im Fall, dass p, q Primzahlen sind, gilt:

$$\begin{aligned} ggT(p,q) &= 1 \\ \Rightarrow 1 &= up + vq \\ \Rightarrow up &= 1 \bmod q \\ vq &= 1 \bmod p \end{aligned}$$

Also ist u das multiplikative Inverse von p modulo q und v das Inverse von q modulo p.

2.2.2 Der Square-and-multiply-Algorithmus

Für die Exponentiation modulo einer Zahl n gibt es einen sehr effizienten Algorithmus, der insbesondere in der Public-Key-Kryptographie sehr oft verwendet

wird. Anstatt das Ergebnis einer Berechnung $a^b \bmod n$ zu berechnen, indem man erst a^b berechnet und anschließend den Rest bei Ganzzahldivision durch n ausgibt, geht man etwas geschickter vor. Dazu benötigt man erst einmal die Binärdarstellung der m-bit-Zahl b:

$$b = \sum_{i=0}^{m-1} b_i \cdot 2^i$$

Die Idee ist dann, im Gegensatz zum naiven Algorithmus, die Zwischenergebnisse möglichst klein zu halten, um so Rechenaufwand zu sparen:

$$\begin{aligned} a^b \bmod n &= a^{\sum b_i \cdot 2^i} \bmod n \\ &= \prod_{i=0}^{m-1} a^{b_i \cdot 2^i} \bmod n \\ &= \prod_{i,b_i} a^{2^i} \bmod n \\ &= \prod_{i,b_i} (a^{2^i} \bmod n) \bmod n \end{aligned}$$

Durch das wiederholte modulo-Rechnen in den Zwischenschritten entstehen keine Zwischenergebnisse, die größer als n sind. Ein weiterer Trick ist das Verwenden der folgenden Äquivalenz:

$$(a^{2^{i-1}})^2 \bmod n = a^{2^i} \bmod n$$

Man kann also den Exponenten von a schrittweise halbieren. Insgesamt erhält man den folgenen Algorithmus:

```
Eingabe: a, b, n
Ausgabe: a^b mod n
```

```
int s=1;
int t=a mod n;
for(int i=0; i<m; i++){
   if(b_i==1)s=s*t mod m;
   t=t*t mod n; //in Iteration i ist t = a^(2^i) mod n
}
return(s);
```

Abbildung 2.2: Square-and-multiply-Algorithmus

Statt wie im naiven Ansatz b Multiplikationen durchzuführen, muss man nur noch höchstens $2 \cdot \log_2(b)$ Multiplikationen, was sich bei großen Werten sehr schnell bemerkbar macht.

2.2.3 Das RSA-Kryptosystem

Das *RSA-Kryptosystem* ist nach deren Entwicklern Rivest, Shamir und Adleman benannt und wahrscheinlich das verbreitetste Public-Key-Kryptosystem. Es ist ein wichtiger Punkt in der Motivation dieser Arbeit und soll deswegen hier kurz vorgestellt werden.

Man wählt zwei Primzahlen p, q und berechnet $n = pq$. Dann berechnet man

$$\varphi(n) = (p-1)(q-1)$$

und wählt $e \in \mathbb{Z}_{\varphi(n)}$, um das Inverse $d = e^{-1} \bmod \varphi(n)$ zu bestimmen[2]. Die öffentlichen Schlüssel sind (n, e) und der geheime Schlüssel ist d.

Eine Nachricht X wird verschlüsselt, indem man mit Hilfe des öffentlichen Schlüssels e den Chiffretext $Y = X^e \bmod n$ berechnet. Um Y wieder zu entschlüsseln, berechnet man $Y^d = X^{de} = X^{1 \bmod \varphi(n)} = X \bmod n$ und erhält so wieder den Klartext.[3]

Ein Angreifer, der ohne Kenntnis der geheimen Schlüssel die Kryptogramme entschlüsseln will müsste die e-te Wurzel aus Y modulo n berechnen. Falls er die Zahl n faktorisieren kann, ist er dazu in der Lage, was im folgenden Abschnitt beschrieben wird.

2.2.4 e-te Wurzeln modulo p

Eine Zahl x heißt e-te Wurzel von y (modulo p) falls $y = x^e \bmod p$ gilt. Das Berechnen dieser Wurzeln ist im Gegensatz zum Berechnen von e-ten Wurzeln modulo n leicht.

Da p prim ist, gilt offensichtlich $ggt(p, x) = 1$. Nach 2.1.5 gilt dann

$$x^{\varphi(p)} = 1 \bmod p$$

Um die e-te Wurzel zu ziehen, rechnet man wie folgt:

$$\begin{aligned} & y = x^e \\ \Leftrightarrow\ & y^d = (x^e)^d \\ \Leftrightarrow\ & y^d = x^{ed} \\ \Rightarrow\ & (y^d = x \Leftrightarrow ed = 1 \bmod \varphi(p)) \end{aligned}$$

Man muss also lediglich das Inverse

$$d = e^{-1} \bmod \varphi(p)$$

[2] siehe 2.2.1
[3] siehe 2.1.5

berechnen, was dank $\varphi(p) = p - 1$ und 2.2.1 effizient geht. Hier sieht man auch deutlich, dass das Ziehen e-ter Wurzeln modulo n leicht ist, falls man n faktorisieren kann, da man damit $\varphi(n) = (p-1) \cdot (q-1)$ berechnen kann, was direkt zum geheimen Schlüssel $d = e^{-1} \bmod \varphi(n)$ führt. Damit wäre also die Verschlüsselung gebrochen und der Angreifer hätte sein Ziel erreicht.

Kapitel 3

Der Lanczos-Algorithmus

Im Verlauf der Arbeit wird es erforderlich sein, riesige Lineare Gleichungssysteme über $\mathbb{F}_2$ zu lösen. Dazu gibt es bereits mehrere Algorithmen, z.B. *Structured Gauss Elimination*, die *Block-Lanczos-Methode* und der *Block-Wiedemann-Algorithmus*.

Da der Gauss-Algorithmus mit einer Laufzeit von $O(n^3)$ für die Größe der hier entstehenden LGS nicht mehr effizient genug ist, wird in diesem Abschnitt der Lanczos Algorithmus vorgestellt, der den Gauss-Algorithmus für dieses Problem bei dünnbesetzten Matrizen bezüglich der Performance um ein Vielfaches übertrifft. Die Laufzeit ist in

$$O(d \cdot n^2) + O(n^2)$$

wobei d die durchschnittliche Anzahl der Einsen pro Spalte ist. Für den Wiedemann-Algorithmus siehe z.B. [27].

Die wichtigsten Quellen für dieses Kapitel sind insbersondere [25], [24] und [26]. Die erste Quelle enthält den hier beschriebenen ursprünglichen Algorithmus von Montgomery, in der dritten Quelle ist eine noch bessere und komplexere Methode mit Lookahead angegeben.

3.1 Standard-Lanczos

Im Standardverfahren werden Gleichungssysteme über $\mathbb{R}$ mittels einer Iteration approximativ gelöst und nicht über $\mathbb{F}_2$. Allerdings ist die Lösung ab einem bestimmten Iterationsschritt exakt und das Verfahren kann auch auf andere algebraische Systeme übertragen werden.

3.1.1 Die Grundidee

Die Grundidee des Verfahrens ist die Verwendung von Krylov-Unterräumen, die wie folgt definiert sind:

Sei A eine symmetrische positiv definite $n \times n$-Matrix über $\mathbb{R}$ und $b \in \mathbb{R}^n$. Der von b erzeugte Krylov-Unterraum ist

$$W := span\left(\{b, A \cdot b, A^2 \cdot b, A^3 \cdot b, ..., A^{m-1} \cdot b\}\right)$$

wobei $span\left(\{M\}\right)$ den von M erzeugten Vektorraum bezeichnet und m der minimale Index ist, für den W nicht mehr durch Hinzufügen weiterer Vektoren erweitert wird.

Falls man eine Basis $W' = \{w_0, ..., w_{m-1}\}$ des von b erzeugten Krylov-Vektorraumes W finden kann mit den Eigenschaften

- $w_i^T \cdot A \cdot w_j = 0$ für alle $i \neq j$, d.h. die Basis ist orthogonal
- $w_i^T \cdot A \cdot w_i \neq 0$ für alle $0 \leq i < m$

dann ist die gesuchte Lösung $x \in \mathbb{R}^n$ zu $A \cdot x = b$ gegeben durch

$$x = \sum_{j=0}^{m-1} \frac{w_j^T \cdot b}{w_j^T \cdot A \cdot w_j} \cdot w_j$$

3.1.2 Die Hintergründe

Um zu zeigen, dass das oben angegebene x auch tatsächlich eine Lösung zu $A \cdot x = b$ ist, geht man wie folgt vor:

$$\begin{aligned}
w_i^T \cdot A \cdot x &= w_i^T \cdot A \cdot \left(\frac{w_0^T \cdot b}{w_0^T \cdot A \cdot w_0} \cdot w_0 + \cdots + \frac{w_{m-1}^T \cdot b}{w_{m-1}^T \cdot A \cdot w_{m-1}} \cdot w_{m-1} \right) \\
&= \frac{w_0^T \cdot b}{w_0^T \cdot A \cdot w_0} \cdot w_i^T \cdot A \cdot w_0 + \cdots + \frac{w_i^T \cdot b}{w_i^T \cdot A \cdot w_i} \cdot w_i^T \cdot A \cdot w_i \\
&+ \cdots + \frac{w_{m-1}^T \cdot b}{w_{m-1}^T \cdot A \cdot w_{m-1}} \cdot w_i^T \cdot A \cdot w_{m-1} \\
&= 0 + \cdots + 0 + \frac{w_i^T \cdot b}{w_i^T \cdot A \cdot w_i} \cdot w_i^T \cdot A \cdot w_i + 0 + \cdots + 0 \\
&= w_i^T \cdot b
\end{aligned}$$

Dies gilt für alle $w_i \in W'$ und, da W' eine Basis von W ist, auch für alle $w \in W$. Folglich gilt für alle $w \in W$ die Gleichung

$$\begin{aligned} & w^T \cdot A \cdot x = w^T \cdot b \\ \Leftrightarrow & w^T \cdot A \cdot x - w^T \cdot b = 0 \\ \Leftrightarrow & w^T \cdot (A \cdot x - b) = 0 \end{aligned}$$

Da A symmetrisch ist, ist die zu A zugehörige Abbildung selbstadjungiert. Daher gilt folgende Gleichheit:

$$w_i^T \cdot A \cdot (A \cdot x - b) = (A \cdot w_i)^T \cdot (A \cdot x - b)$$

Außerdem ist $A \cdot w_i$ ein Element aus W und daher gilt, wie gerade gezeigt,

$$w_i^T \cdot A \cdot (A \cdot x - b) = 0$$

Genauso ist $A \cdot x - b$ ein Element aus W, weshalb es eine kanonische Darstellung bezüglich der Basis W' gibt:

$$A \cdot x - b = c_0 \cdot w_0 + \cdots + c_{m-1} \cdot w_{m-1}$$

Also gilt für alle $w_i \in W'$

$$\begin{aligned} 0 &= w_i^T \cdot A \cdot (A \cdot x - b) \\ &= w_i^T \cdot A \cdot (c_0 \cdot w_0 + \cdots + c_{m-1} \cdot w_{m-1}) \\ &= c_0 \cdot w_i^T \cdot A \cdot w_0 + \cdots + c_i \cdot w_i^T \cdot A \cdot w_i + \cdots + c_{m-1} \cdot w_i^T \cdot A \cdot w_{m-1} \\ &= c_i \cdot w_i^T \cdot A \cdot w_i \end{aligned}$$

Da nach den Basiseigenschaften $w_i^T \cdot A \cdot w_i \neq 0$ gilt, muss $c_i = 0$ sein, und da i beliebig war, muss $c_i = 0$ für alle $0 \leq i < m$ sein, also

$$\begin{aligned} & A \cdot x - b = 0 \\ \Leftrightarrow & A \cdot x = b \end{aligned}$$

Die gesuchte Lösung ist also gefunden, indem die Basisvektoren W' sehr geschickt mit speziellen Eigenschaften gewählt wurden.

Diese Basisvektoren $w_0, ..., w_{m-1}$ sind also der Kern des Algorithmus und deren Berechnung soll als nächstes in Angriff genommen werden. Dazu wird eine leichte Abwandlung der Gram-Schmidt-Methode benutzt, und zwar wie folgt:

$$\begin{aligned} w_0 &= b \\ w_i &= A \cdot w_{i-1} - \sum_{j=0}^{i-1} c_{ij} \cdot w_j \end{aligned}$$

wobei

$$c_{ij} = \frac{w_j^T \cdot A^2 \cdot w_{i-1}}{w_j^T \cdot A \cdot w_j}$$

bis ein m gefunden ist, für das $w_m = \overrightarrow{0}$ gilt.

Zu zeigen ist, dass $W' = \{w_0, ..., w_{m-1}\}$ eine Basis von W mit den beiden geforderten Eigenschaften ist, was per Induktion gezeigt werden kann. Sei also die Menge $W_i' = \{w_0, ..., w_i\}$ für $0 \leq i \leq l-1$ konstruiert und erfülle die geforderten Bedingungen.[1]

Die Elemente der Menge W_l' müssen linear unabhängig sein, da sonst gelten würde:

$$\begin{aligned} 0 &= a_0 \cdot w_0 + \cdots + a_{l-1} \cdot w_{l-1} \\ \Rightarrow 0 &= A \cdot (a_0 \cdot w_0 + \cdots + a_{l-1} \cdot w_{l-1}) \end{aligned}$$

Daraus kann man für jedes j mit $0 \leq j \leq l-1$ herleiten:

$$\begin{aligned} 0 &= w_j^T \cdot 0 \\ &= w_j^T \cdot A \cdot (a_0 \cdot w_0 + \cdots + a_{l-1} \cdot w_{l-1}) \\ &= w_j^T \cdot (a_0 \cdot A \cdot w_0 + \cdots + a_{l-1} \cdot A \cdot w_{l-1}) \\ &= a_0 \cdot w_j^T \cdot A \cdot w_0 + \cdots + a_j \cdot w_j^T \cdot A \cdot w_j + \cdots + a_{l-1} \cdot w_j^T \cdot A \cdot w_{l-1} \\ &= a_j \cdot w_j^T \cdot A \cdot w_j \end{aligned}$$

Da laut Annahme die Eigenschaft $w_j^T \cdot A \cdot w_j \neq 0$ gilt, muss $a_j = 0$ sein, und das für alle $0 \leq j \leq l-1$. Also existiert nur die triviale Linearkombination von $w_0, ..., w_{l-1}$ zum Nullvektor, weshalb die Elemente von W_l' linear unabhängig sein müssen. Kommen wir nun zur Berechnung von w_l:

$$w_l = A \cdot w_{l-1} - \frac{w_0^T \cdot A^2 \cdot w_{l-1}}{w_0 \cdot A \cdot w_0} \cdot w_0 - \cdots - \frac{w_{l-1}^T \cdot A^2 \cdot w_{l-1}}{w_{l-1} \cdot A \cdot w_{l-1}} \cdot w_{l-1}$$

[1] Der Induktionsanfang für $l = 1$ ist trivial.

Setzt man dies für alle $w_j \in W'_{l-1}$ ein, dann folgt

$$\begin{aligned}
&w_j^T \cdot A \cdot w_l \\
&= w_j^T \cdot A \cdot \left(A \cdot w_{l-1} - \frac{w_0^T \cdot A^2 \cdot w_{l-1}}{w_0 \cdot A \cdot w_0} \cdot w_0 - \cdots - \frac{w_{l-1}^T \cdot A^2 \cdot w_{l-1}}{w_{l-1} \cdot A \cdot w_{l-1}} \cdot w_{l-1} \right) \\
&= w_j^T \cdot A^2 \cdot w_{l-1} - \frac{w_0^T \cdot A^2 \cdot w_{l-1}}{w_0 \cdot A \cdot w_0} \cdot w_j^T \cdot A \cdot w_0 - \cdots \\
&- \frac{w_j^T \cdot A^2 \cdot w_{l-1}}{w_j \cdot A \cdot w_j} \cdot w_j^T \cdot A \cdot w_j - \cdots - \frac{w_{l-1}^T \cdot A^2 \cdot w_{l-1}}{w_{l-1} \cdot A \cdot w_{l-1}} \cdot w_j^T \cdot A \cdot w_{l-1} \\
&= w_j^T \cdot A^2 \cdot w_{l-1} - 0 - \cdots - 0 - w_j^T \cdot A^2 \cdot w_{l-1} - 0 - \cdots - 0 \\
&= 0
\end{aligned}$$

Also ist die Bedingung $w_i^T \cdot A \cdot w_j = 0$ für $i \neq j$ erfüllt.

3.1.3 Verbesserungen

Die Iteration kann noch vereinfacht werden. Sei $j < i - 2$. Mit

$$\begin{aligned}
w_{j+1} &= A \cdot w_j - \frac{w_0^T \cdot A^2 \cdot w_j}{w_0 \cdot A \cdot w_0} \cdot w_0 - \cdots - \frac{w_j^T \cdot A^2 \cdot w_j}{w_j \cdot A \cdot w_j} \cdot w_j \\
\Leftrightarrow A \cdot w_j &= w_{j+1} + \frac{w_0^T \cdot A^2 \cdot w_j}{w_0 \cdot A \cdot w_0} \cdot w_0 + \cdots + \frac{w_j^T \cdot A^2 \cdot w_j}{w_j \cdot A \cdot w_j} \cdot w_j
\end{aligned}$$

und der Symmetrie von A (d.h. die zugehörige Abbildung ist selbstadjungiert) folgt:

$$\begin{aligned}
&w_j^T \cdot A^2 \cdot w_{i-1} \\
&= (A \cdot w_j)^T \cdot A \cdot w_{i-1} \\
&= \left(w_{j+1} + \frac{w_0^T \cdot A^2 \cdot w_j}{w_0 \cdot A \cdot w_0} \cdot w_0 + \cdots + \frac{w_j^T \cdot A^2 \cdot w_j}{w_j \cdot A \cdot w_j} \cdot w_j \right)^T \cdot A \cdot w_{i-1} \\
&= w_{j+1}^T \cdot A \cdot w_{i-1} + \frac{w_0^T \cdot A^2 \cdot w_j}{w_0 \cdot A \cdot w_0} \cdot w_0^T \cdot A \cdot w_{i-1} + \cdots \\
&+ \frac{w_j^T \cdot A^2 \cdot w_j}{w_j \cdot A \cdot w_j} \cdot w_j^T \cdot A \cdot w_{i-1} \\
&= 0
\end{aligned}$$

da $j < i - 2$ und damit $j + 1 < i - 1$ und für $i \neq j$ gilt $w_j^T \cdot A \cdot w_i = 0$. Es fallen also für $j < i - 2$ alle Zähler der Summe in der Iteration weg. Man erhält

deswegen:

$$
\begin{aligned}
w_0 &= b \\
w_1 &= A \cdot w_0 - c_{1,0} \cdot w_0 \\
w_i &= A \cdot w_{i-1} - c_{i,i-1} \cdot w_{i-1} - c_{i,i-2} \cdot w_{i-2}
\end{aligned}
$$

3.2 Block-Lanczos

Im Gegensatz zum Standardalgorithmus müssen in den in dieser Arbeit vorgestellten Siebalgorithmen einige Hindernisse überwunden werden, um Lanczos auch auf Matrizen über $\mathbb{F}_2$ statt $\mathbb{R}$ anwenden zu können. Dabei entsteht sogar noch eine enorme Laufzeitverbesserung, da durch das Verwenden von Bit-Blöcken und logischen Operatoren sehr viele Berechnungen gleichzeitig ausgeführt werden können.

Legt man eine 64-bit-Architektur zu Grunde, kann man die komponentenweise Addition von Vektoren über $\mathbb{F}_2$ immer mit 64 Vektoren gleichzeitig durch eine einzige XOR-Operation erledigen. Dadurch können z.B. zwei $n \times n$-Matrizen in $n \cdot \frac{n}{64}$ Rechenschritten addiert werden, was theoretisch einen Geschwindigkeitsvorteil um den Faktor 64 bedeutet.

3.2.1 Auftretende Probleme

Als erstes ist die Bedingung, dass die Matrix symmetrisch sein muss, in den Siebalgorithmen normalerweise nicht erfüllt. Außerdem ist stets eine Kombination zum Nullvektor gesucht, also $b = 0$.

Betrachtet man die Iteration etwas genauer, so fällt auf, dass diese sofort abbricht, da bereits $w_0 = 0$ ist. Es wird also für den Nullvektor nur die triviale Lösung $x = 0$ berechnet.

Ein weiteres Problem ist, dass die Bedingung $w_i^T \cdot A \cdot w_j = 0$ für $i \neq j$ im Fall von Vektoren über $\mathbb{F}_2$ nicht erfüllt sein muss.

3.2.2 Lösungen

Das Symmetrieproblem lässt sich für eine $n_1 \times n_2$-Matrix B sehr elegant umgehen, indem man das Gleichungssystem $A \cdot x = 0$ mit der symmetrischen $n_2 \times n_2$-Matrix $A = B^T \cdot B$ löst. Eine Lösung x für $A \cdot x = 0$ ist gleichfalls eine Lösung für $B \cdot x = 0$, da

$$
\begin{aligned}
0 = A \cdot x = \left(B^T \cdot B\right) \cdot x = B^T \cdot (B \cdot x) \\
\Rightarrow B \cdot x = 0
\end{aligned}
$$

Die anderen beiden Probleme werden umgangen, indem man mit Unterräumen von Vektoren arbeitet, anstatt mit einzelnen Vektoren. Dazu wählt man eine weitere Matrix Y mit zufälligen Spaltenvektoren, sodass $A \cdot Y$ nicht die Nullmatrix ist. Mit $A \cdot Y$ wird dann analog gearbeitet wie mit b.

Dazu wird eine Folge von Unterräumen $\mathcal{W}_i$ analog zu den w_i mittels einer Iteration berechnet. Die Unterräume $\mathcal{W}_i$ haben spezielle Eigenschaften, sodass die Schwierigkeiten mit $w_i^T \cdot B \cdot w_j = 0$ für $i \neq j$ umgangen werden.

Anschließend wird mit einer ähnlichen Formel wie zur Berechnung von x oben eine Matrix X berechnet, für die $A \cdot X = A \cdot Y$ gilt. Damit gilt $A \cdot (X - Y) = 0$ und es können Linearkombinationen der Spalten von $X - Y$ berechnet werden, die Lösungen zu $A \cdot x = 0$ bilden.

Die Konstuktion der Folge von Unterräumen wird im folgenden Abschnitt genauer beschrieben.

3.2.3 Folgen orthogonaler Unterräume

In diesem Abschnitt gelten folgende Notationen. A ist eine symmetrische $n \times n$-Matrix über $\mathbb{F}_2$. Zwei Vektoren $v, w \in \mathbb{F}_2^n$ heißen A-orthogonal, falls $v^T \cdot A \cdot w = 0$. Seien $\mathcal{V}, \mathcal{W}$ Unterräume von $\mathbb{F}_2^n$. Zur Vereinfachung dienen folgende Schreibweisen:

- $\mathcal{V} + \mathcal{W} = \{v + w | v \in \mathcal{V}, w \in \mathcal{W}\}$
- $A \cdot \mathcal{V} = \{A \cdot v | v \in \mathcal{V}\}$
- $\mathcal{V}^T \cdot \mathcal{W} = \{v^T \cdot w | v \in \mathcal{V}, w \in \mathcal{W}\}$
- $\mathcal{V}$ und $\mathcal{W}$ heißen A-orthogonal, falls für alle $v \in \mathcal{V}, w \in \mathcal{W}$ gilt, dass v und w A-orthogonal sind. Das ist gleichbedeutend mit $\mathcal{V}^T \cdot A \cdot \mathcal{W} = \{0\}$.
- Sei V eine $n_1 \times_2$-Matrix. $span\,(V)$ ist der von den Spaltenvektoren von V aufgespannte Unterraum von $\mathbb{F}_2^{n_1}$.
- Die Anzahl der bits eines Hauptspeicherworts sei N.
- $\mathcal{W} \subseteq \mathbb{F}_2^n$ heißt A-invertierbar, falls $\mathcal{W}$ eine Basis hat, deren Spaltenvektoren eine Matrix W bilden, für die gilt, dass $W^T \cdot A \cdot W$ invertierbar ist.

Die Idee ist, die gleichen Methoden zu verwenden wie im Standardalgorithmus. Dazu müssen alle dort verwendeten Ideen verallgemeinert werden. Deswegen stelle ich zuerst einmal die „neue Grundidee“ vor.

Man berechnet eine Folge von Unterräumen $(\mathcal{W}_i)_{i_0}^{m-1}$, die paarweise A-orthogonal sind. Die Verallgemeinerung der oben beschriebenen Bedingungen an diese Folge ist:

- $\mathcal{W}_i$ ist A-invertierbar.
- $\mathcal{W}_j^T \cdot A \cdot \mathcal{W}_i = \{0\}$, falls $i \neq j$.
- $A \cdot \mathcal{W} \subseteq \mathcal{W}$, wobei $\mathcal{W} = \mathcal{W}_0 + \cdots + \mathcal{W}_{m-1}$.

Angenommen, eine solche Folge sei berechnet, dann kann man zu jedem Unterraum $\mathcal{W}_j$ eine Basis angeben, deren Elemente die Spaltenvektoren einer Matrix W_j bilden. Mit diesen W_j lässt sich die Lösung des inhomogenen Gleichungssystems $A \cdot x = b$ berechnen, indem man x wie folgt konstruiert:

$$x = \sum_{j=0}^{m-1} W_j \cdot \left(W_j^T \cdot A \cdot W_j\right)^{-1} \cdot W_j^T \cdot b$$

Die Folge von Unterräumen ist der neue Kern des Algorithmus und wird durch eine Verallgemeinerung der Lanczos-Iteration berechnet. Dazu wählt man eine zufällige Startmatrix V_0 und geht wie folgt vor:

$$\begin{aligned} W_i &= V_i \cdot S_i \\ V_{i+1} &= A \cdot W_i \cdot S_i^T + V_i - \sum_{j=0}^{i} W_j \cdot C_{i+1,j} \\ \mathcal{W}_i &= span\,(W_i) \end{aligned}$$

Dabei ist

$$C_{i+1,j} = \left(W_j^T \cdot A \cdot W_j\right)^{-1} \cdot W_j^T \cdot A \cdot \left(A \cdot W_i \cdot S_i^T + V_i\right)$$

und S_i eine $N \times N_i$-Projektion, die so gewählt ist, dass $W_i^T \cdot A \cdot W_i$ invertierbar und gleichzeitig N_i maximal ist. Es werden also bei der Berechnung von W_i so viele Spalten wie möglich von V_i ausgewählt, sodass W_i noch A-invertierbar ist.

Die Iteration bricht ab, falls $V_i^T \cdot A \cdot V_i = 0$ für ein $i = m$ gilt. Auf den Beweis für die Korrektheit der neuen Iteration möchte ich an dieser Stelle verzichten, da er analog zum Beweis im Standardverfahren per Induktion aufgebaut ist. Er ist nachzulesen in [25], wo die Iteration außerdem, wie im Standardfall, weiter vereinfacht wird. Letztendlich bleibt die folgende vereinfachte Iteration:

$$V_{i+1} = A \cdot V_i \cdot S_i \cdot S_i^T + V_i \cdot D_{i+1} + V_{i-1} \cdot E_{i+1} + V_{i-2} \cdot F_{i+1}$$

für $i \geq 0$ mit den Hilfsmatrizen

$$\begin{aligned}
W_i^{inv} &= S_i \cdot \left(W_i^T \cdot A \cdot W_i\right)^{-1} \cdot S_i^T \\
&= S_i \cdot \left(S_i^T \cdot V_i^T \cdot A \cdot V_i \cdot S_i\right)^{-1} \cdot S_i^T \\
D_{i+1} &= I_N - W_i^{inv} \cdot \left(V_i^T \cdot A^2 \cdot V_i \cdot S_i \cdot S_i^T + V_i^T \cdot A \cdot V_i\right) \\
E_{i+1} &= -W_{i-1}^{inv} \cdot V_i^T \cdot A \cdot V_i \cdot S_i \cdot S_i^T \\
F_{i+1} &= -W_{i-1}^{inv} \cdot \left(I_N - Vi - 1^T \cdot A \cdot V_{i-1} \cdot W_{i-1}^{inv}\right) \\
&\quad \cdot \left(V_{i-1}^T \cdot A^2 \cdot V_{i-1} \cdot S_{i-1} \cdot S_{i-1}^T + V_{i-1}^T \cdot A \cdot V_{i-1}\right) \cdot S_i \cdot S_i^T
\end{aligned}$$

und den Zusatzbedingungen, dass W_j^{inv} und V_j gleich 0 und S_j gleich I_N gesetzt werden für $j < 0$.

Um mit dieser Iteration ein homogenes Gleichungssystem zu lösen, also $A \cdot x = 0$ statt $A \cdot x = b$, wählt man eine zufällige $n \times N$-Matrix Y über $\mathbb{F}_2$, berechnet $A \cdot Y$ und versucht eine $n \times N$-Matrix X zu finden mit $A \cdot X = A \cdot Y$.

Dazu initialisiert man V_0 mit $A \cdot Y$ und durchläuft die Lanczos-Iteration, bis $V_i^T \cdot A \cdot V_i = 0$ für $i = m$ erreicht ist. Die gesuchte Matrix X wird dann folgendermaßen berechnet:

$$\begin{aligned}
X &= \sum_{i=0}^{m-1} W_i \cdot \left(W_i^T \cdot A \cdot W_i\right)^{-1} \cdot W_i^T \cdot V_0 \\
&= \sum_{i=0}^{m-1} V_i \cdot W_i^{inv} \cdot V_i^T \cdot V_0
\end{aligned}$$

Mit Hilfe von Gauss-Elimination kann man Linearkombinationen von $X - Y$ und V_m berechnen, um Vektoren x zu finden, für die $A \cdot x = 0$ gilt.

3.3 Bemerkungen

Der hier vorgestellte Block-Lanczos-Algorithmus ist auch als Montgomery-Methode bekannt und nicht in allen Fällen erfolgreich. Dies liegt daran, dass im Fall einer asymmetrischen Matrix B die symmetrische Matrix $A = B^T \cdot B$ als Eingabe benutzt werden muss. Falls der Rang von A kleiner ist als $rang\,(B) - N + 1$, schlägt Montgomerys Algorithmus fehl.

Trotzdem ist die hier von mir vorgestellte Variante sehr verbreitet und z.B. in LinBox[2] enthalten. In diesem Projekt sind außerdem Lookahead-Block-Lanczos-Varianten, Biorthogonal-Lanczos sowie der Wiedemann-Algorithmus implementiert. Daher werden in dieser Arbeit die Funktionen aus dem LinBox-Paket verwendet, und auf eine eigene Implementierung wird verzichtet.

[2]siehe `http://www.linalg.org`

Details zur Implementierung und eine ausführliche Beschreibung des Ablaufs des Algorithmus kann man z.B. in [28] finden.

3.4 Laufzeitanalyse

Die Laufzeit von

$$O(d \cdot n^2) + O(n^2)$$

ergibt sich dadurch, dass $O(\frac{n}{N})$ Iterationen ausgeführt werden, in denen jeweils die folgenden Berechnungen ausgeführt werden:

- Berechnung von $A \cdot V_i$ in $O(d \cdot n)$, wobei d die durchschnittliche Anzahl der Einsen in A ist.
- Berechnung von $V_i^T \cdot A \cdot V_i$ und $V_i^T \cdot A^2 \cdot V_i$ in $O(N \cdot n)$ durch Anwendung der Bitoperationen AND und XOR.
- Einige $N \times N$-Matrixoperationen die nicht ins Gewicht fallen.
- Vier Multiplikationen von $n \times N$- mit $N \times N$-Matrizen, die auch durch Bitoperationen in $O(N \cdot n)$ berechnet werden können.

Für eine genauere Herleitung der Laufzeit wird auf [25] verwiesen.

Kapitel 4

Das Quadratische Sieb (QS)

Das Quadratische Sieb ist eine Weiterentwicklung des Linearen Siebs von Richard Schroeppel durch Carl Pomerance[1] und kann als Vorläufer des GNFS angesehen werden. Es zeigt bereits die gleichen wesentlichen Merkmale, z.B. die Aufteilung der Faktorisierung in den Sieb-Schritt und den Matrix-Schritt.

Die Laufzeit der Quadratischen Siebs ist in

$$O\left(e^{(1+o(1))\cdot\sqrt{\log(n)\cdot\log(\log(n))}}\right)$$

Die wichtigsten Quellen für dieses Kapitel sind der Überblick von Lenstra in [1] und die Originalarbeit von Carl Pomerance in [9].

4.1 Die Idee des QS

Die Grundidee um n zu faktorisieren besteht darin, eine so genannte *quadratische Kongruenz* zu finden, d.h. Werte x, y mit:

$$\begin{aligned} x^2 &= y^2 \bmod n \\ \Leftrightarrow x^2 - y^2 &= 0 \bmod n \end{aligned}$$

Also ist

$$x^2 - y^2 = (x+y)(x-y)$$

ein Vielfaches von n und falls

$$x \neq \pm y \bmod n$$

kann man mit Hilfe von 2.2.1 einen Primfaktor von n durch $ggT(x+y, n)$ und $ggT(x-y, n)$ berechnen.

[1] siehe [9].

Solche Werte existieren aus folgendem Grund: Setze

$$\begin{aligned} x = \frac{p-q}{2} \quad \text{und} \quad y &= \frac{p+q}{2} \\ \Rightarrow x + y &= p \\ x - y &= q \\ \Rightarrow x^2 - y^2 &= (x+y)(x-y) \\ &= pq = n = 0 \bmod n \\ \Rightarrow x^2 &= y^2 \bmod n \end{aligned}$$

Das eigentliche Problem ist also das Suchen solcher Werte x, y, was der zentrale Teil des QS ist. Dies geschieht allerdings nicht, indem direkt x, y gesucht werden, sondern indem viele „kleinere“ Werte $v, s_v = v^2$ gesucht werden, für die gilt, dass s_v bezüglich der Faktorbasis

$$F_k = \{p_1, ..., p_k\}$$

glatt ist. Anschließend wird aus diesen Werten die gesuchte Gleichung $x^2 = y^2 \bmod n$ konstruiert.

4.2 Quadratisches Sieben

Die generelle Idee beim „Sieben“ geht auf das *Sieb des Eratosthenes* zurück. Hier werden alle Primzahlen bis zu einer Schranke S berechnet, indem nacheinander alle Vielfachen der Primzahlen $\leq \sqrt{S}$ „herausgestrichen“ werden.

Ähnlich sollen im QS glatte Zahlen gefunden werden, indem man Vielfache der Faktorbasiselemente „herausstreicht“. Dieses Verfahren ist wesentlich effizienter als die triviale Suche durch ständiges Dividieren.

4.2.1 Kleine Werte finden

Als erstes stellt man fest, dass die Wahrscheinlichkeit, dass s_v glatt ist, umso geringer ist, je größer s_v ist. Für zufällig gewählte Werte v ist es wahrscheinlich, dass $s_v = v^2$ sehr groß wird. Daher versucht man kleine s_v zu finden, die in der Größenordnung $\sqrt{n}$ liegen, indem man $v = v(i) = \sqrt{n} + i$ für kleines i wählt.

$$\begin{aligned} \Rightarrow s_{v(i)} &= v(i)^2 \bmod n \\ &= (\sqrt{n} + i)^2 - n \\ &= n + 2i\sqrt{n} + i^2 - n \\ &= 2i\sqrt{n} + i^2 \end{aligned}$$

4.2.2 Effizientes Suchen: Sieben

Eine wichtige Folgerung ist:

$$p \mid s_{v(i)} \Rightarrow p \mid s_{v(i)+tp}, \quad \forall t \in \mathbb{N}$$

Der Fokus der Suche nach B-glatten Zahlen kann also vom eigentlichen „Zahlen raten“ auf die Analyse der Faktorbasis $F_k = \{p_1, ..., p_k\}$ verschoben werden. Statt Zahlen durch alle Elemente der Faktorbasis zu teilen, geht man die Elemente p_j der Faktorbasis durch und markiert direkt diejenigen Zahlen, die durch p_j teilbar sind. Man spart sich so unzählige Divisionen.

Mit dieser Methode kann man sehr schnell viele B-glatte Werte finden. Dazu gibt man zunächst ein Intervall vor, in dem $v(i)+tp$ liegen darf. Man beachte, dass mit steigendem i die Wahrscheinlichkeit für B-glatte Werte abnimmt. In einem mit $s_{v(i)}$ initialisierten Siebarray wird anschließend für alle $p_j \in \{p_1, ..., p_k\}$, die $s_{v(i)}$ teilen, an den Stellen $i + tp$ durch die höchste vorkommende Potenz von p_j geteilt, solange $i + tp$ im gegebenen Intervall liegt. Zuletzt faktorisiert man alle $s_{v(i)}$ über der Faktorbasis, falls im Siebarray an der entsprechenden Stelle eine 1 steht. Dazu kann man entweder trivial dividieren oder die ECM benutzen.

Dieser gesamte Vorgang wird als das eigentliche „Sieben“ bezeichnet und ist wesentlich effizienter, als einfach nur alle Werte s_v über der Faktorbasis zu faktorisieren, indem man durch deren Elemente teilt.

4.3 Der Matrix-Schritt

Wir haben nun sehr viele Beziehungen der Form

$$\begin{aligned} v(i_1)^2 &= p_1^{e_{11}} p_2^{e_{12}} ... p_k^{e_{1k}} \bmod n \\ &\vdots \\ v(i_m)^2 &= p_1^{e_{m1}} p_2^{e_{m2}} ... p_k^{e_{mk}} \bmod n \end{aligned}$$

Das Ziel ist jetzt, aus diesen Beziehungen eine einzige zu berechnen, sodass eine Gleichung der Form $x^2 = y^2 \bmod n$ entsteht. Dazu werden sie in geeigneter Weise multipliziert. Dabei bleibt offensichtlich auf der linken Seite immer ein Quadrat stehen, da Produkte von Quadraten wieder Quadrate sind.

Um auf der rechten Seite ein Quadrat zu erhalten, muss eine geeignete Teilmenge gefunden werden, sodass die Exponenten des Produkts bei allen p_j gerade sind. Dazu stellt man eine $(m \times k)$-Matrix mit den Elementen $e_{ij} \in \mathbb{F}_2$ auf, indem man $e_{ij} \bmod 2$ berechnet. Anschließend sucht man darin eine Indexmenge S, die

den Nullvektor erzeugt. Dazu löst man

$$\begin{pmatrix} e_{11} & \cdots & e_{1k} \\ \vdots & \ddots & \vdots \\ e_{m1} & \cdots & e_{mk} \end{pmatrix}^T \cdot x = 0$$

Für dieses Problem existieren mindestens $m-k$ Lösungen und es gibt mehrere Algorithmen, mit denen man diese auch finden kann, z.B. den in dieser Arbeit vorgestellten Block-Lanczos-Algorithmus. Der Lösungsvektor x (oder eine Spalte der Matrix x, falls mehrere Lösungen berechnet werden) wählt dann diejenigen glatten Werte aus, die aufmultipliziert auf der rechten Seite nur gerade Exponenten haben. Mit jeder einzelnen Lösung S kann man eine Gleichung der Form $x^2 = y^2 \bmod n$ aufstellen und n faktorisieren:

$$\begin{aligned} x^2 &= (\prod_{j \in S} v(i_j))^2 \\ &= \prod_{j \in S} v(i_j)^2 \\ &= \prod_{j \in S} p_j^{2e_j} \\ &= (\prod_{j \in S} p_j^{e_j})^2 \\ &= y^2 \bmod n \end{aligned}$$

4.4 Laufzeitanalyse

Um eine Idee zur Abschätzung zur Laufzeit entwickeln zu können, sind noch ein Paar wenige Hilfsmittel nötig. Es sein nochmals angemerkt, dass die im Folgenden dargestellte Laufzeitabschätzung lediglich als Anleitung zu einer formalen Analyse dienen kann und keinesfalls vollständing oder formal korrekt ist.

Als erstes sei die Anzahl der B-glatten Werte im Intervall $[1, ..., x]$ definiert durch

$$\psi(x, B) := |\{m \mid 1 \leq m \leq x, m \text{ ist } B\text{-glatt}\}|$$

Desweiteren sei die Anzahl der Primzahlen kleiner als B definiert durch die Funktion $\pi(B)$.

Sei weiter $B = x^{\frac{1}{u}}$. Dann ist die Wahrscheinlichkeit, dass eine zufällige Zahl kleiner x B-glatt ist, gleich

$$\frac{\psi(x, x^{\frac{1}{u}})}{x}$$

und der Kehrwert davon ergibt die Anzahl der zufällig auszuwählenden Zahlen, bis eine einzige glatte gefunden ist.

Da die Faktorbasis $\pi(B)$ Elemente hat, müssen wir mindestens ebensoviele glatte Werte finden, um das Gleichungssystem aufzustellen und auch lösen zu können. Durch die Technik des Siebens sind lediglich $\log(\log(B))$ Schritte pro Kandidat notwendig.

Die Anzahl der Schritte, um alle notwendigen B-glatten Werte zu finden, ist demnach

$$\pi(B) \cdot (\log(\log(B))) \cdot \frac{x}{\psi(x, x^{\frac{1}{u}})}$$

Schätzt man die Faktoren grob ab durch

- $\pi(B) \cdot (\log(\log(B))) \approx x^{\frac{1}{u}}$
- $\frac{x}{\psi(x, x^{\frac{1}{u}})} \approx u^u$

so kann man den folgenden einfacheren Term nach u minimieren:

$$x^{\frac{1}{u}} \cdot u^u$$

Dazu minimiert man den Logarithmus des Terms

$$\frac{1}{u} \cdot \log(x) + u \cdot \log(u)$$

Die Ableitung ist gleich null, falls

$$u^2 \cdot (\log(u) + 1) = \log(x)$$

und indem man den Logarithmus dieser Gleichung nimmt, erhält man

$$u \approx \frac{1}{2} \cdot \log(\log(x))$$

Setzt man dies ein, erhält man

$$u \approx \sqrt{2 \cdot \log(x) \cdot \log(\log(x))}$$

und

$$B = e^{\left(\frac{1}{\sqrt{2}} + o(1)\right) \cdot \sqrt{\log(x) \cdot \log(\log(x))}}$$

Mit diesem Wert für B ergibt sich

$$x^{\frac{1}{u}} \cdot u^u = e^{\left(\sqrt{2} + o(1)\right) \cdot \sqrt{\log(x) \cdot \log(\log(x))}}$$

Mit $x = n^{\frac{1}{2}+o(1)}$ erhält man

$$B = e^{\left(\frac{1}{2}+o(1)\right)\cdot\sqrt{\log(n)\cdot\log(\log(n))}}$$

und

$$x^{\frac{1}{u}} \cdot u^u = e^{(1+o(1))\cdot\sqrt{\log(n)\cdot\log(\log(n))}}$$

was der Anzahl der Schritte in etwa entspricht und somit die Laufzeit bestimmt.

Zu beachten ist, dass es sich hier lediglich um den Siebschritt handelt, der Matrix-Schritt kann in ähnlicher Weise analysiert werden und führt zur gleichen Abschätzung für die Anzahl der Schritte. Allerdings ist es dazu notwendig, andere Algorithmen zu verwenden als Gauss-Elimination, z.B. den Lanczos-Algorithmus[2].

Für Genaueres siehe z.B. [10]. Hier wird auch deutlich dass eine kleinere Wahl von B zu einer größeren Laufzeit der Siebphase und einer kleineren Laufzeit des Matrix-Schrittes führt und umgekehrt.

4.5 Verbesserungen

Um das QS noch effizienter zu machen, gibt es zahlreiche Ideen. Da diese auch im GNFS angewendet werden, stelle ich sie hier schon einmal vor.

Multiple polynomial variation des QS bedeutet, dass nicht nur die Gleichung $(\sqrt{n}+i)^2-n$ benutzt wird, um glatte Zahlen zu finden, sondern eine Reihe anderer Polynome, die im Wesentlichen die gleichen Eigenschaften haben. Dadurch lässt sich das Sieben sehr leicht auf viele Rechner parallelisieren, da jeder Knoten ein unterschiedliches Polynom verwenden kann. Außerdem vermeidet man, dass mit steigendem i die Wahrscheinlichkeit für glatte Werte abnimmt, indem man das Polynom wechselt.

Nach einer Idee von Peter Montgomery[3] kann man $t := (\sqrt{n}+i)^2$ durch $a\cdot t+b$ ersetzen, wobei $b^2 = n \bmod a, b \leq \frac{a}{2}$ gelten muss. Dann sind alle Funktionswerte von $(a \cdot t + b)^2 - n$ durch a teilbar. Falls a selbst ein Quadrat ist, kann man nach glatten Funktionswerten des Polynoms

$$f_{a,b}(t) = \frac{1}{a}((a \cdot t + b)^2 - n)$$

suchen.

[2] siehe Kapitel 3.

[3] Die Idee ist zu finden unter `http://www.math.niu.edu/~rusin/known-math/98/mpqs`, veröffeltlicht ist sie in [20].

Logarithmisches Sieben spart die Divisionen beim „Markieren“ im Sieb. Im mit null initialisierten Sieb wird für alle $p_j \in \{p_1, ..., p_k\}$, die $s_{v(i)}$ teilen, an den Stellen $i + tp$ der Wert $\log p_j$ addiert, für alle $i + tp$, die im gegebenen Intervall liegen. Dadurch muss nicht durch diese Werte dividiert werden und man kann sich sehr viele dieser „teuren“ Rechenoperationen sparen.

Danach steht an i-ter Stelle im Sieb die Summe der Logarithmen der p_k, die $s_{v(i)}$ teilen. Also vergleicht man die i-te Stelle im Sieb mit $\log s_{v(i)}$ und versucht $s_{v(i)}$ über der Faktorbasis zu faktorisieren, falls die Werte nahe beieinander liegen. Da beim Addieren der Logarithmen gerundet wird, muss man eine Grenze festlegen, bis zu der sich die Werte unterscheiden dürfen.

Large primes werden benutzt, um das Sieben zu beschleunigen. Falls im Sieb Logarithmen addiert werden, die im Endeffekt gerundet werden, erhält man auch zahlreiche „fast“ glatte Zahlen, die bis auf einen relativ kleinen Faktor über der Faktorbasis faktorisiert werden können. Beim normale Sieben fällt gleichermaßen auf, dass an vielen Stellen im Sieb relativ kleine Werte übrig bleiben. Falls dieser Restfaktor prim ist, spricht man von *partial relations*, und falls er das Produkt aus zwei Primzahlen ist, spricht man von *double partial relations*. Die Primfaktoren, die nicht in der Faktorbasis sind, heißen *large primes*.

Diese Beziehungen erhält man fast ohne zusätzlichen Aufwand beim Sieben. Es ist nicht einmal notwendig einen Primzahltest durchzuführen, falls man die Grenze für large primes so wählt, dass sie zwischen p_k und p_k^2 liegt, da dann bereits durch alle Primzahlen $\leq p_k$ geteilt wurde und der large-prime-Kandidat garantiert prim ist. Für double partials muss man die Grenze höher setzen und entsprechende Tests durchführen.

Mit Hilfe von zwei *partial relations* kann man eine *full relation* konstruieren, falls sie die gleiche large prime haben, indem man diese multipliziert:

$$\begin{aligned} v(i_r)^2 &= p_1^{e_{r1}} p_2^{e_{r2}} ... p_k^{e_{rk}} q \\ v(i_s)^2 &= p_1^{e_{s1}} p_2^{e_{s2}} ... p_k^{e_{sk}} q \\ \Rightarrow (v(i_r)v(i_s))^2 &= p_1^{e_{r1}+e_{s1}} p_2^{e_{r2}+e_{s2}} ... p_k^{e_{rk}+e_{sk}} q^2 \\ \Rightarrow \left(\frac{v(i_r)v(i_s)}{q}\right)^2 &= p_1^{e_{r1}+e_{s1}} p_2^{e_{r2}+e_{s2}} ... p_k^{e_{rk}+e_{sk}} \end{aligned}$$

Dadurch bekommt man die neue Zahl $\frac{v(i_r)v(i_s)}{q}$, die komplett über der Faktorbasis faktorisierbar, also glatt ist.

Um die double partials zu verwerten, muss man diese mit partials zu full relations kombinieren, wobei wieder die large primes übereinstimmen müssen. Da-

durch erhält man analog zu oben neue glatte Zahlen mit nur geringem Mehraufwand.

Der Gewinn an Geschwindigkeit durch das Verwenden von partial relations, also die Anzahl der gefundenen glatten Werte pro Zeiteinheit, ist ein Faktor 2,5 und der Gewinn durch double partial relations nochmals ein Faktor 2 - 2,5.[4]

[4] siehe [1].

Kapitel 5

Das General Number Field Sieve (GNFS)

Das General Number Field Sieve (GNFS) oder auch Zahlkörpersieb ist die Erweiterung des Special Number Field Sieve (SNFS). Dieses wiederrum basiert auf einer Idee von John Pollard, mit der Zahlen der Form $x^3 + k$ für kleines k sehr schnell faktorisiert werden können. Durch zahlreiche mathematische Tricks wurde das NFS für die Anwendung auf beliebige Zahlen erweitert.

Die wichtigsten Quellen für dieses Kapitel sind [11], darin vor allem [15], sowie [23] und [24]. Interessanterweise lässt sich das GNFS auch zum Berechnen des Diskreten Logarithmus verwenden. Dies wird z.B. in [29] erklärt.

Die Laufzeit des GNFS ist in

$$O(e^{((\frac{64}{9})^{\frac{1}{3}}+o(1))\cdot(\log(n))^{\frac{1}{3}}\cdot(\log(\log(n)))^{\frac{2}{3}}})$$

Der signifikante Laufzeitvorteil des GNFS gegenüber dem QS ist mit der dritten Wurzel im Exponenten zu erklären. Dazu nochmal zum Vergleich die Laufzeit des QS in anderer Schreibweise:

$$O\left(e^{(1+o(1))\cdot(\log(n))^{\frac{1}{2}}\cdot(\log(\log(n)))^{\frac{1}{2}}}\right)$$

Da sich die Laufzeiten nur durch die Wurzeln und den Vorfaktor im Exponenten unterscheiden, wird in vielen Arbeiten, z.B. in [15], [23] oder [24], folgende Funktion definiert:

$$L_n(u,v) := e^{v\cdot(\log(n))^u\cdot(\log(\log(n)))^{1-u}}$$

Damit ergibt sich für das QS eine Laufzeit von

$$L_n\left(\frac{1}{2}, 1 + o(1)\right)$$

und für das GNFS eine Laufzeit von

$$L_n\left(\frac{1}{3}, (\frac{64}{9})^{\frac{1}{3}} + o(1)\right)$$

Betrachtet man die beiden Extrema $L_n(0, v)$ und $L_n(1, v)$, dann stellt man fest, dass der Ausdruck für $u = 0$ polynomiell in $\log_2(n)$, also der Bitlänge von n, ist und für $u = 1$ exponenziell. Daher ist die Laufzeit beider Algorithmen subexponenziell.

5.1 Grundidee

Die grundlegende Idee des GNFS ist die gleiche wie beim Quadratischen Sieb. Gesucht ist eine quadratische Kongruenz $x^2 = y^2 \bmod n$, die mit Hilfe vieler kleinerer Puzzleteile zusammengesetzt wird. Der wesentliche Unterschied ist die Art der Puzzleteile. Zusammenfassend geht man in 3 Schritten vor:

Schritt 1

Berechne ein über $\mathbb{Q}$ irreduzibles Polynom $f(x) = f_d \cdot x^d + ... + f_0$ vom Grad $d > 1$ und eine Zahl m mit $f(m) > 0$, aber $f(m) = 0 \bmod n$. Wähle eine Wurzel $\rho \in \mathbb{C}$ von $f(x) \Rightarrow \mathbb{Q}(\rho)$ bildet einen Zahlkörper.

Schritt 2

Berechne eine Menge S von Paaren (a, b) mit den Eigenschaften

- $\prod_{(a,b)\in S}(a + bm) = x^2 \in \mathbb{Z}$
- $\prod_{(a,b)\in S}(a + b\rho) = \delta^2 \in \mathbb{Q}(\rho)$

Dann erhält man folgende quadratische Kongruenz:

$$\begin{aligned} y^2 := \psi(\delta)^2 &= \psi(\delta^2) \bmod n \\ &= \psi\left(\prod_{(a,b)\in S} (a+b\rho)\right) \bmod n \\ &= \prod_{(a,b)\in S} \psi\left(a+b\rho\right) \bmod n \\ &= \prod_{(a,b)\in S} (a+b\cdot m) \bmod n \\ &=: x^2 \bmod n \end{aligned}$$

Die Abbildung $\psi : \mathbb{Z}[\rho] \to \mathbb{Z}_n$ mit $\psi(\rho) = m$ ist dabei der natürliche Ringhomomorphismus.

Schritt 3

Berechne die Wurzeln

$$\sqrt{\prod_{(a,b)\in S} (a+bm)} = x \in \mathbb{Z}$$
$$\sqrt{\prod_{(a,b)\in S} (a+b\rho)} = \delta \in \mathbb{Z}[\rho]$$

und damit auch $y = \psi(\delta)$. Jetzt kann man wie gewohnt die Faktorisierung von n mit Hilfe des ggT berechnen.

Das eigentliche Problem beim GNFS ist also aufteilbar in die *Polynomwahl*, die Konstruktion der *Menge von Quadraten* S sowie das Berechnen der rationalen und algebraischen *Wurzeln*. Dabei wird sich herausstellen, dass der algebraische Teil im Gegensatz zum rationalen wesentlich schwieriger ist. Im Folgenden werde ich detailliert auf die einzelnen Schritte eingehen.

5.2 Wahl des Polynoms

Die Polynomwahl ist ein wichtiger Schritt des Algorithmus. Unterschiedliche Polynome verursachen sehr unterschiedliche Laufzeiten, weshalb dieser Schritt eine wichtige Stelle für Optimierungen ist.

In diesem Abschnitt wird die einfachste und verlässlichste Methode der Polynomwahl vorgestellt, sowie zwei Möglichkeiten, dieses generierte Polynom mit

nur geringem Aufwand zu verbessern. Arbeiten, die sich ausschließlich mit der Polynomwahl beschäftigen, sind z.B. [22] oder [21].

5.2.1 Base-m-Methode

In der ursprünglichen Version[1] des GNFS wird die so genannte *base-m*-Methode benutzt.

Gegeben ist die zu faktorisierende Zahl n und gesucht ein Polynom $f \in \mathbb{Z}[x]$ sowie eine Zahl m, sodass $f(m) = 0$ gilt. Dazu wählt man den Polynomgrad $d > 1$ mit

$$d = \left(3^{\frac{1}{3}} + o(1)\right) \cdot \left(\frac{\log(n)}{\log(\log(n))}\right)^{\frac{1}{3}}$$

und rundet ihn auf die nächste ungerade Zahl. Dann setzt man $m = \lceil n^{\frac{1}{d}} \rceil$ und anschließend berechnet man n zur Basis m (daher auch der Name):

$$n = c_d \cdot m^d + c_{d-1} \cdot m^{d-1} + ... + c_0$$

wobei $0 \leq c_i < m$ gilt. Als Ergebnis erhält man das Polynom

$$f(x) = x^d + c_{d-1} \cdot x^{d-1} + ... + c_0$$

mit der Nullstelle $f(m) = 0$.

Ein erfreulicher Nebeneffekt ist, dass das Polynom mit hoher Wahscheinlichkeit irreduzibel ist. Falls nicht, hat man sogar direkt die eine Faktorisierung gefunden: Sei $f = g \cdot h$ eine Faktorisierung von f in $\mathbb{Z}[x]$. Dann ist $n = f(m) = g(m) \cdot h(m)$ eine Faktorisierung von n in $\mathbb{Z}$ und man ist damit bereits fertig. Dies ist allerdings äußerst unwahrscheinlich.

Um zu testen, ob das generierte Polynom irreduzibel ist, gibt es zahlreiche Möglichkeiten. Eine davon ist das *Irreduzibilitätskriterium von Eisenstein*, das unter 2.1.2 beschrieben ist.

Man kann auch $m = \lceil n^{\frac{1}{d+1}} \rceil$ setzen, um kleinere Koeffizienten zu erhalten, allerdings auf Kosten der Normiertheit des dadurch erzeugten Polynoms. Dies lässt sich zwar durch etwas theoretische Arbeit kompensieren, allerdings werde ich in dieser Arbeit nicht darauf eingehen. Dazu siehe z.B. [21], wo auch die Verwendung von geraden Polynomgraden beschrieben ist.

5.2.2 Kleine Koeffizienten

Ein Ziel bei der Optimierung der Polynomparameter ist fast immer, die Koeffizienten des Polynoms zu verkleinern, denn kleinere Koeffizienten bedeuten, dass

[1] siehe [15].

die (weiter unten definierte) Norm $N(a+b\cdot\rho)$ kleiner und damit wahrscheinlicher glatt ist. Die Koeffizienten c_i des Polynoms

$$f(x) = x^d + c_{d-1} \cdot x^{d-1} + ... + c_0$$

das die base-m-Methode erzeugt, haben zunächst einmal die Größe $|c_i| < m$ für $0 \leq i < d$. Dies kann man jedoch zu $|c_i| < \frac{m}{2}$ verbessern, indem man im Fall $|c_i| > \frac{m}{2}$ den Koeffizient c_{i+1} durch $c_{i+1}+1$ und c_i durch $-(m-c_i)$ ersetzt, denn:

$$\begin{aligned}(c_{i+1}+1)\cdot m^{i+1} - (m-c_i)\cdot m^i &= c_{i+1}\cdot m^{i+1} + m^{i+1} - m^{i+1} + c_i \cdot m^i \\ &= c_{i+1}\cdot m^{i+1} + c_i \cdot m^i\end{aligned}$$

Auf diese Art kann man alle Koeffizienten kleiner als $\frac{m}{2}$ bekommen, was ein wichtiger Schritt bei der Verbesserung der Laufzeit ist.

5.2.3 Addition von Polynomen

Um die Koeffizienten des Polynoms $f(x)$ noch weiter zu verkleinern, kann man ein weiteres Polynom $g(x)$ mit $g(m) = 0$ zu $f(x)$ addieren. Dazu definiert man ein oder gleich mehrere Polynome durch

$$g(x) = \sum_{i=1}^{d-1} g_i \cdot \left(x^i - m \cdot x^{i-1}\right)$$

Dann gilt $g(m) = 0$, denn

$$\begin{aligned}g(m) &= \sum_{i=1}^{d-1} g_i \cdot \left(m^i - m \cdot m^{i-1}\right) \\ &= \sum_{i=1}^{d-1} g_i \cdot \left(m^i - m^i\right) \\ &= 0\end{aligned}$$

Auch für dieses Polynom kann man die Koeffizienten, wie gerade gezeigt, auf unter $\frac{m}{2}$ verkleinern und durch Addition zu $f(x)$ möglicherweise kleinere Koeffizienten erreichen. Außerdem ist bei der Wahl von $g(x)$ offensichtlich die Eigenschaft $g(x) = 0$ unabhängig von der Wahl der g_i, sodass man sehr effizient viele Polynome erzeugen und testen kann, ob bei der Addition zu $f(x)$ die Koeffizienten auch kleiner werden.

5.3 Berechnung der Quadrate

Um die Menge S zu finden und daraus die quadratische Kongruenz $x^2 = y^2 \bmod n$ zu konstruieren, benutzt man wie üblich ein Siebverfahren.

Der wesentliche Unterschied zum QS ist, dass eine Menge T von Werten (a, b) gesucht ist, sodass $a + bm$ glatt ist und gleichzeitig $a + b\rho$ in $\mathbb{Z}[\rho]$ glatt ist, wobei diese Glätte noch definiert werden muss. Dies geschieht in zwei separaten Siebverfahren, dem *rationalen Sieb* und dem *algebraischen Sieb*. Anschließend kann man wie im QS mit linearer Algebra eine geeignete Teilmenge $S \subset T$ berechnen.

5.3.1 Das rationale Sieb

Eingabe: T_0, F_{k_1}
Ausgabe: T_1

```
for(b=1; b<=u; b++){
  for(a=-u, a<u mod n, a++){
    if(T_0[a][b]==1)T_1[a][b]=a+b*m;
    else T_1[a][b]=0;
  }//Sieb mit a+bm initialisiert
}

for(b=1; b<=u; b++){
  for(i=0, i<k_1, i++){
    a=-u +b*m % n;
    while(a mod F_k_1[i] != 0)a++;
    for(a, a<u mod n, a=a+F_k_1[i] mod n){
      if(T_0[a][b]==0) continue;
      while(T_1[a][b] mod F_k_1[i]==0)
        T_1[a][b] /= F_k_1[i];
    }
  }
}//jetzt steht bei rational glatten Werten 1

return(T_1);
```

Abbildung 5.1: rationales Sieb

Zuerst gibt man sich eine genügend große Grenze u vor, bis zu der die Werte

a und b gesucht werden. Die Menge der Kandidaten ist dann

$$T_0 = \{(a,b)|ggT(a,b) = 1; -u \leq a \leq u; 0 < b \leq u\}$$

Dies erreicht man durch sukzessives Anwenden von (2.2.1) oder durch den Algorithmus aus der Verbesserung in (5.6.3). Man beachte, dass durch die Wahl der Siebgrenzen nur positive Werte für $(a+bm)$ vorkommen, sofern $|u| \leq m$ ist. Dies wird später von Bedeutung sein.

Als nächstes wird das gleiche Siebverfahren verwendet wie beim Quadratischen Sieb. Dazu wählt man einen Glätteparameter B, die Faktorbasis

$$\mathcal{F}_{k_1} = \{p_1, ..., p_{k_1} | p_i \text{ prim }, p_i < B, p_i < p_{i-1}\}$$

und siebt die Menge

$$T_1 = \{(a,b) \in T_0 | a + bm \text{ ist } B\text{-glatt}\}$$

Wie sich später noch zeigen wird, ist die Wahl der Parameter u und B entscheidend für die Laufzeit des Algorithmus. In Kapitel 5.5 wird herausgestellt, dass die Parameter folgende Werte haben sollten:

$$B = u = e^{\left(\frac{1}{2}+o(1)\right)\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

Das eigentliche Sieben geschieht, indem man für jedes fixierte $b \in [1, ..., u]$ ein Array aufstellt, das mit $a + bm \; \forall (a,b) \in T_0$ initialisiert ist. In diesen Arrays wird jeweils für alle Elemente p_j der Faktorbasis $F_{k_1} = \{p_1, ..., p_{k_1}\}$ an allen Stellen mit $a + bm = 0 \bmod p_j$ durch die höchste enthaltene Potenz von p_j geteilt, indem man wiederholt p_j zur aktuellen Stelle addiert. Dabei nutzt man ähnlich wie im QS folgende Siebbedingung aus:

$$p_j | (a + bm) \Rightarrow p_j | (a + bm + lp_j) \quad \forall l \in \mathbb{N}$$

Nach Ablauf von Algorithmus 5.1 stehen an genau den Stellen (a, b) in T_1 Einsen, an denen $ggT(a, b) = 1$ gilt und $a + bm$ B-glatt ist, und an allen anderen Stellen steht 0.

Sind genug glatte Paare gefunden, könnte theoretisch durch lineare Algebra bereits eine Teilmenge S konstruiert werden, für die

$$\prod_{(a,b)\in S} (a + bm)$$

ein Quadrat in $\mathbb{Z}$ ist. Allerdings genügt das nicht, denn es soll zusätzlich noch das algebraische Quadrat in $\mathbb{Z}[\rho]$ konstruiert werden.

5.3.2 Das algebraische Sieb

Um die Menge der (a, b)-Paare weiter auszusieben, sodass zusätzlich die Eigenschaft erfüllt ist, dass

$$\prod_{(a,b)\in S} (a + b\rho)$$

ein Quadrat in $\mathbb{Z}[\rho]$ ist, sollen jetzt die gleichen Ideen genutzt werden wie im rationalen Sieb. Dazu muss aber zunächst einmal geklärt werden, was „glatt“ im algebraischen Sinne bedeutet.

Ein Element $\beta \in \mathbb{Z}[\rho]$ heißt B-glatt, falls seine Norm $N(\beta) \in \mathbb{Z}$ B-glatt ist. Die Norm kann für den Spezialfall $\beta = (a + b \cdot \rho)$ wie folgt berechnet werden:

$$\begin{aligned} N(a + b\rho) &= (-b)^d \cdot f(-\frac{a}{b}) \\ &= a^d - c_{d-1} \cdot a^{d-1} \cdot b + ... + (-1)^d \cdot c_0 \cdot b^d \end{aligned}$$

Die Siebidee

Um ein Sieb zu konstruieren, definiert man zunächst einmal für jedes p_j aus der Faktorbasis die Menge

$$R(p_j) = \{r \in \{0, ..., p_j - 1\} | f(r) = 0 \bmod p_j\}$$

Dann sind alle $r_j \in R(p_j)$ Wurzeln von $f(x) \bmod p_j$.

Anschließend sucht man wie im rationalen Sieb für jedes fixierte b mit $0 < b \leq u$ die Zahlen a mit $N(a + b\rho) = 0 \bmod p_j$, indem man für $r \in R(p_j)$ den Wert $a = -br \bmod p_j$ berechnet und siebt. Es gilt folgende Siebbedingung:

$$p_j | (a + br) \Rightarrow p_j | (a + br + lp_j)$$

Ein Siebarray wird für jedes fixierte b und $-u \leq a \leq u$ mit den Werten $N(a + b\rho)$ initialisiert. Für jedes p_j mit $p_j \nmid b$ und $\forall r \in R(p_j)$ identifiziert man die Stellen mit $a = -br \bmod p_j$, indem man die erste dieser Positionen sucht und dann wiederholt p_j addiert, dann wird durch die höchste enthaltene Potenz von p_j dividiert. Dadurch spart man sich wieder unzählige Divisionen, wie es für ein Sieb typisch ist.

Am Ende von Algorithmus 5.2 steht genau an den Stellen im Siebarray ± 1, an denen $a + b\rho$ B-glatt ist. Damit könnte man eine Teilmenge S finden, für die die *Norm* des Produktes

$$\prod_{a,b\in S} a + b\rho$$

Eingabe: T_1, F_{k_2}
Ausgabe: T_2

```
for(int b=1; b<=u; b++){
  for(a=-u; a<u; a++){
    if(T_1[a][b]==1){
      T_1[a][b]=N(a, b); //Norm(a+b*rho)
    }else{
      T_1[a][b]=0;
    }//kein glatter Wert oder ggT>1
  }
}//Sieb mit Norm initialisiert

for(int b=1; b<=u; b++){
  for(i=0; i<k2; i++){
    p_i=F_k_2[i][0]; //Elemente sind (p_i, r_i)
    r_i=F_k_2[i][1];
    if(b%F_k_2[i]==0)continue; //p_i teilt b
    a=0;
    while((a+b*r_i)%p_i!=0)a++; //erstes a suchen...
    for(a; a<u; a=a+p_i){ //sieben...
      if(T_1[a][b])==0){
        continue; //kein rational glatter Wert
      }
      while(T_1[a][b]%p_i==0){
        T_1[a][b]=T_1[a][b]/p_i;
      }//hoechste Potenz rausdividieren
    }
  }
}//jetzt steht bei algebraisch glatten Werten 1
```

Abbildung 5.2: algebraisches Sieb

ein Quadrat in $\mathbb{Z}$ ist. Das ist eine notwendige Bedingung dafür, dass das Produkt selbst ein Quadrat in $\mathbb{Z}[\rho]$ ist, aber keine hinreichende. Die Lösung besteht darin, sich für jedes p_j zu merken, welches $r \in R(p_j)$ dazu führt, dass p_j $N(a+b\rho)$ teilt.

Erst an dieser Stelle möchte ich analog zur rationalen Faktorbasis F_{k_1} die eigentliche algebraische Faktorbasis $\mathcal{F}_{k_2}$ einführen, da sie häufig direkt ohne Erklärung verwendet wird:

$$\mathcal{F}_{k_2} := \{(p_1, r_1), ..., (p_{k_2}, r_{k_2}) | f(r_i) = 0 \bmod p_i, i = 1, ..., k_2\}$$

Dabei sollte man anmerken, dass die Werte (p_i, r_i) jeweils eindeutig Primidealen $\mathcal{P}_i = \langle p_i \rangle$ primer Norm entsprechen[2], die noch mit Zusatzinformation ausgestattet sind.

In manchen Arbeiten, z.B. in [23], wird für r_i der Wert ∞ zugelassen falls mit nicht normierten Polynomen gearbeitet wird und $p_i \mid c_d$ gilt. Dies macht den Ablauf jedoch etwas komplizierter und daher wird in dieser Arbeit nur der Fall normierter Polynome betrachtet. Dann ist immer $c_d = 1$ und $p_i \nmid c_d$.

Die Details

Sei $ord_p(x)$ die Anzahl der Faktoren p in x. Sei $a, b \in \mathbb{Z}$ mit $ggT(a, b) = 1$, sei p prim und $r \in R(p)$. Dann ist $e_{p,r}(a + b\rho)$ definiert als

$$e_{p,r}(a + b\rho) := \begin{cases} ord_p(N(a + b\rho)) & \text{falls } a + br = 0 \bmod p \\ 0 & \text{sonst} \end{cases}$$

und es folgt

$$N(a + b\rho) = \pm \prod_{p,r} p^{e_{p,r}(a+b\rho)}$$

Die Zahl $e_{p,r}(a + b\rho)$ gibt sozusagen die Anzahl der Primidealfaktoren $\mathcal{P}$ des von $(a + b\rho)$ erzeugten Ideals an, also den Exponent in der „Primidealfaktorzerlegung“.

Falls nun ein Quadrat berechnet wurde, müssen alle Exponenten gerade sein, also muss für alle Primzahlen p und $r \in R(p)$ gelten:

$$\prod_{(a,b)\in S} (a + b\rho) = \gamma^2 \Rightarrow \sum_{(a,b)\in S} e_{p,r}(a + b\rho) = 0 \bmod 2 \tag{5.1}$$

Genau genommen ist hiervon genau die Umkehrung interessant, die so allerdings nicht gilt. Dies kann man jedoch durch das Einführen der so genannten *quadratic characters* erreichen. Dazu mehr auf Seite 51.

Ein weiteres Problem ist, dass diese Überlegungen nur für den speziellen Fall $\mathbb{Z}[\rho] = \mathcal{O}_\rho$ gelten. Daher muss noch etwas mehr algebraische Theorie eingeführt werden[3]. Für jedes Primideal $\mathcal{P}$ aus $\mathbb{Z}[\rho]$ existiert ein Gruppenhomomorphismus

$$l_\mathcal{P} : \mathbb{Q}(\rho)^* \longrightarrow \mathbb{Z}$$

mit folgenden Eigenschaften:

- $l_\mathcal{P}(\beta) \geq 0$ für alle $\beta \in \mathbb{Z}[\rho]$, $\beta \neq 0$

[2] siehe 2.1.3.

[3] siehe insb. [15]

- Wenn $\beta \in \mathbb{Z}[\rho]$ und $\beta \neq 0$ dann gilt: $l_{\mathcal{P}}(\beta) > 0 \Leftrightarrow \beta \in \mathcal{P}$
- Für jedes $\beta \in \mathbb{Q}(\rho)$ gilt: $l_{\mathcal{P}}(\beta) = 0$ für alle bis auf endlich viele $\mathcal{P}$ und

$$\prod_{\mathcal{P}} (N(\mathcal{P}))^{l_{\mathcal{P}}(\beta)} = |N(\beta)|$$

  wobei $\mathcal{P}$ die Menge aller Primideale in $\mathbb{Z}[\rho]$ durchläuft.

Seien $a, b \in \mathbb{Z}$ mit $ggT(a, b) = 1$. Wenn $\mathcal{P}$ ein Primideal nicht primer Norm ist, dann ist $l_{\mathcal{P}}(a + b\rho) = 0$. Ansonsten ist $l_{\mathcal{P}}(a + b\rho) = e_{p,r}(a + b\rho)$, wobei $\mathcal{P}$ das korrespondierende Primideal zu (p, r) ist. Damit lässt sich auch die Feststellung 5.1 oben zeigen:

$$\begin{aligned}
\prod_{(a,b)\in S} (a + b\rho) &= \gamma^2 \\
\Rightarrow \sum_{(a,b)\in S} e_{p,r}(a + b\rho) &= \sum_{(a,b)\in S} l_{\mathcal{P}}(a + b\rho) \\
&= l_{\mathcal{P}}(\prod_{(a,b)\in S} (a + b\rho)) \\
&= l_{\mathcal{P}}(\gamma^2) \\
&= 2 \cdot l_{\mathcal{P}}(\gamma) \\
&= 0 \bmod 2
\end{aligned}$$

Die vier Hindernisse

Wie oben bereits erwähnt, sind noch einige weitere Hindernisse zu überwinden, was vor allem durch Verwendung von *quadratic characters* gelingt. Doch zunächst einmal das Ziel und alle Probleme auf einen Blick. Gesucht ist eine Teilmenge $S \subset T_2$, sodass das von

$$\prod_{(a,b)\in S} (a + b\rho)$$

erzeugte Ideal ein Quadrat in $\mathbb{Z}[\rho]$ ist. Sind genug glatte Werte gefunden, kann man inzwischen eine solche Teilmenge S berechnen, sodass

$$\sum_{(a,b)\in S} l_{\mathcal{P}}(a + b\rho) = 0 \bmod 2 \quad \forall \mathcal{P}$$

Daraus folgt allerdings noch nicht unbedingt, dass das Produkt von oben ein Quadrat in $\mathbb{Z}[\rho]$ ist:

- Das von $\prod_{(a,b)\in S}(a+b\rho)$ erzeugte Ideal

$$\left\langle \prod_{(a,b)\in S} (a+b\rho) \right\rangle$$

in $\mathcal{O}_\rho$ muss nicht zwangsläufig das Quadrat eines Ideals sein, da wir mit Primidealen in $\mathbb{Z}[\rho] \subseteq \mathcal{O}_\rho$ arbeiten statt mit Primidealen in $\mathcal{O}_\rho$.

- Selbst wenn

$$\left\langle \prod_{(a,b)\in S} (a+b\rho) \right\rangle = \mathcal{I}^2$$

das Quadrat eines Ideals $\mathcal{I}$ in $\mathcal{O}_\rho$ ist, könnte es sein, dass $\mathcal{I}$ selbst kein Hauptideal ist, wodurch man kein Erzeugendes angeben kann.

- Selbst wenn

$$\left\langle \prod_{(a,b)\in S} (a+b\rho) \right\rangle = \langle \gamma^2 \rangle$$

für ein $\gamma \in \mathcal{O}_\rho$ ist, muss nicht unbedingt gelten, dass

$$\prod_{(a,b)\in S} (a+b\rho) = \gamma^2$$

- Selbst wenn

$$\prod_{(a,b)\in S} (a+b\rho) = \gamma^2$$

für ein $\gamma \in \mathcal{O}_\rho$ gilt, muss nicht unbedingt $\gamma \in \mathbb{Z}[\rho]$ sein.

Für die ersten drei Probleme werden in [15] Lösungen vorgeschlagen, die weder elegant noch effizient sind, die Laufzeit des gesamten Algorithmus verschlechtert sich dadurch! Daher werde ich diese Ideen nicht vorstellen, sondern gleich die Lösung, die seit der Veröffentlichung von [18] für die etwas komplizierteren ersten drei Probleme verwendet wird. Das letzte Problem hingegen lässt sich recht einfach lösen: Sei

$$\prod_{(a,b)\in S} (a+b\rho) = \gamma^2$$

mit $\gamma \in \mathcal{O}_\rho$. Dann ist $\gamma \cdot f'(\rho) \in \mathbb{Z}[\rho]$.[4]

[4] siehe [8]

Daher kann man die Bedingungen der quadratischen Kongruenzen wie folgt abändern: Gesucht ist eine Teilmenge S von T_2 mit den beiden Eigenschaften

$$f'(m)^2 \cdot \prod_{(a,b)\in S} (a + bm) \quad \text{ist ein Quadrat in } \mathbb{Z} \tag{5.2}$$

$$f'(\rho)^2 \cdot \prod_{(a,b)\in S} (a + b\rho) \quad \text{ist ein Quadrat in } \mathbb{Z}[\rho] \tag{5.3}$$

Man kann annehmen, dass $ggT(f'(m), n) = 1$, da sonst eine Faktorisierung von n berechnet ist. Deswegen beeinflusst die Multiplikation mit $f'(m)^2$ nicht die Wahrscheinlichkeit, eine Faktorisierung von n zu finden.

Quadratic characters

In diesem Abschnitt wird ein Hilfsmittel besprochen, das von Leonard M. Adleman in [18] entwickelt wurde: die bereits mehrmals angesprochenen *quadratic characters*. Die Kernaussage ist folgende: Sei $S \subset T_2$ wie oben mit

$$\prod_{(a,b)\in S} (a + b\rho)$$

ist ein Quadrat in $\mathbb{Q}(\rho)$. Sei $(p, r) \in \mathcal{F}_{k_2}$ mit

$$\begin{aligned} f'(r) &\neq 0 \bmod p \quad \text{und} \\ a + br &\neq 0 \bmod p \quad \forall (a, b) \in S \end{aligned}$$

Dann gilt

$$\prod_{(a,b)\in S} \left(\frac{a + br}{p}\right) = 1$$

wobei hier mit $\left(\frac{x}{y}\right)$ das Legendre-Symbol gemeint ist.

Wie in 5.1 ist davon genau die Umkehrung interessant, die aber an dieser Stelle tatsächlich gilt.

Dies bildet die Grundlage für einen probabilistischen Test, ob eine Zahl $l \in \mathbb{Z}$ ein Quadrat ist oder nicht. Dabei wird die Fehlerwahrscheinlichkeit durch die Anzahl der Tests gesteuert, indem über einer Menge von k_3 Primzahlen $p_1, ..., p_{k_3}$ getestet wird. Falls für mindestens eine dieser p_j

$$\left(\frac{l}{p_j}\right) = -1$$

gilt, dann ist l kein Quadrat. Andernfalls ist l mit der Wahrscheinlichkeit $1-(\frac{1}{2})^{k_3}$ ein Quadrat.

Um dieses Resultat zu verwenden, muss, wie gerade angedeutet, eine weitere Primzahlenmenge definiert werden, die einen Faktorbasis-ähnlichen Aufbau hat:

$$\mathcal{F}_{k_3} := \{(q_1, s_1), ..., (q_{k_3}, s_{k_3}) | q_i > p_{k_1}, s_i \in R(q_i) \forall i = 1...k_3\}$$

Es handelt sich also um eine Menge von k_3 Primzahlen, die größer sind als die Elemente der rationalen Faktorbasis. Diese dient dazu, den probabilistischen Test in den Matrix-Schritt einzubauen. Dazu wird außerdem noch folgende Abbildung benötigt:

$$\sigma(\left(\frac{x}{y}\right)) := \begin{cases} 0, & \left(\frac{x}{y}\right) = 1 \\ 1, & \left(\frac{x}{y}\right) = -1 \end{cases}$$

Sie dient im Wesentlichen dazu, aus einer multiplikativen Struktur eine additive Struktur zu machen, denn es gilt Folgendes:

$$\prod_{(a,b)\in S} \left(\frac{a+br}{p}\right) = 1 \Leftrightarrow \sum_{(a,b)\in S} \sigma(\left(\frac{a+br}{p}\right)) = 0 \bmod 2$$

Dies alles soll jetzt in eine riesige Matrix einfließen. Zur Erinnerung: T_2 ist die „gesiebte“ Menge von Paaren (a, b) sodass

- $ggT(a, b) = 1$
- $-u \leq a \leq u$
- $0 < b \leq u$
- $(a + bm)$ ist glatt bezüglich F_{k_1}
- $(a + b\rho)$ ist glatt bezüglich $\mathcal{F}_{k_2}$

Sei e die Abbildung von $T_2 \longrightarrow \mathbb{F}_2^{1+k_1+k_2+k_3}$, die wie folgt definiert ist und auch als Exponentenvektor bezeichnet wird:

- Die erste Koordinate von $e(a, b)$ ist 0, falls $a+bm > 0$, und 1, falls $a+bm < 0$
- Die nächsten k_1 Koordinaten sind

$$ord_{p_1}(a + bm) \bmod 2, ..., ord_{p_{k_1}}(a + bm) \bmod 2$$

- Die nächsten k_2 Koordinaten sind

$$e_{p_1,r_1}(a+b\rho) \bmod 2, ..., e_{p_{k_2},r_{k_2}}(a+b\rho) \bmod 2$$

- Die letzten k_3 Koordinaten sind

$$\sigma(\left(\frac{a+bs_1}{q_1}\right)), ..., \sigma(\left(\frac{a+bs_{k_3}}{q_{k_3}}\right))$$

Falls $|T_2| > k_1 + k_2 + k_3 + 1$, sind die Exponentenvektoren linear abhängig, also kann man mittels linearer Algebra eine Teilmenge $S \subset T_2$ berechnen, sodass $\sum_{(a,b)\in S} e(a,b)$ gleich dem Nullvektor ist und die gesuchte Lösung ist gefunden.

5.4 Wurzeln ziehen

Nachdem die Quadrate in $\mathbb{Z}$ und $\mathbb{Z}[\rho]$ konstruiert sind, müssen die Quadratwurzeln gezogen werden, um mit Hilfe des ggT die Zahl n zu faktorisieren.

5.4.1 Die rationale Wurzel

Die rationale Wurzel ist einfach, da die Primfaktorzerlegungen der Werte $(a+bm)$ bekannt sind. Falls also

$$\begin{aligned} x^2 &= f'(m)^2 \prod_{(a,b)\in S} (a+bm) \\ &= f'(m)^2 \prod_{p_i \in \mathcal{F}_{k_1}} p_i^{\sum_{(a,b)\in S} ord_{p_i}(a+bm)} \end{aligned}$$

ein Quadrat ist, dann muss $\sum_{(a,b)\in S} ord_{p_i}(a+bm)$ für alle $i \in \{1, ..., k_1\}$ gerade sein und die Wurzel ist gegeben durch

$$x = f'(m) \prod_{p_i \in \mathcal{F}_{k_1}} p_i^{\frac{1}{2}\sum_{(a,b)\in S} ord_{p_i}(a+bm)}$$

Da das Ergebnis nur modulo n benötigt wird, kann es sehr effizient mit dem *Square-and-multiply*-Algorithmus[5] berechnet werden.

[5] siehe 2.2.2.

5.4.2 Die algebraische Wurzel

Das GNFS ist der erste Faktorisierungsalgorithmus, der das Ziehen von algebraischen Wurzeln, d.h. von Elementen des Zahlkörpers, erfordert. Das Problem der mathematischen Standardverfahren ist, dass diese für die hier vorkommenden Koeffizienten nicht effizient genug sind, da nicht, wie bei der rationalen Wurzel, „mod n“ gerechnet werden kann.

Im Klartext bedeutet das dass die Wurzel aus

$$\gamma = f'(\rho)^2 \prod_{(a,b)\in S} (a + b\rho)$$

im Zahlkörper berechnet werden muss, also die Wurzel aus einem Polynom $\gamma(x)$ modulo $f(x)$:

$$\begin{aligned}\sqrt{\gamma(x)} &= \sqrt{f'(x)^2 \prod_{(a,b)\in S} (a + b \cdot x)} \bmod f(x) \\ &= f'(x) \prod_{(a,b)\in S} \sqrt{(a + b \cdot x)} \bmod f(x)\end{aligned}$$

und diese Berechnung liefert ein Polynom vom Grad $d - 1$ mit riesigen Koeffizienten.

Methode nach Buhler, Lenstra und Pomerance

In [15], Seite 28 steht „one has to deal with numbers of a truly gigantic size“. Daher müssen andere Methoden betrachtet werden als einfache Standardverfahren. Die dort vorgeschlagene Methode funktioniert folgendermaßen: Sei

$$\gamma = f'(\rho)^2 \prod_{(a,b)\in S} (a + b\rho)$$

das errechnete Quadrat in $\mathbb{Z}[\rho]$. Ein Ansatz zum Wurzelziehen besteht darin, das Produkt auszumultiplizieren, wobei für γ ein Polynom in ρ vom Grad $< d$ entsteht. Anschließend kann man das Polynom $X^2 - \gamma \in \mathbb{Q}(\rho)[X]$ mit einem der Algorithmen faktorisieren, auf die in [15], Seite 29 verwiesen wird.

Da sich diese Algorithmen im Wesentlichen nicht unterscheiden, falls eine Primzahl $q > 2$ existiert, sodass $f \bmod q$ irreduzibel in $\mathbb{F}_q[X]$ ist, soll genau dieser Fall betrachtet werden. Solche Primzahlen heißen übrigens *Träge Primzahlen*.

Zuerst konstruiert man das Primideal $\mathcal{Q} = q\mathbb{Z}[\rho]$ vom Grad d, das aus allen Elementen $\sum_{i=0}^{d-1} a_i \cdot \rho^i$ besteht, für die alle Koeffizienten a_i durch q teilbar sind. Dann nimmt man die Koeffizienten von γ modulo q und berechnet ein Element δ_0 sodass $\delta_0^2 \cdot \gamma = 1 \bmod \mathcal{Q}$.

Dieses δ_0 ist bis auf das Vorzeichen eindeutig. Falls $X - \gamma$ unerwarteterweise irreduzibel modulo $\mathcal{Q}$ sein sollte, dann kann kein solches δ_0 gefunden werden und γ ist kein Quadrat in $\mathbb{Z}[\rho]$. Allerdings braucht man dann nicht aufzugeben, sondern kann einfach noch ein Paar weitere Spalten mit *quadratic characters* in die oben konstruierte Matrix einfügen, um die Wahrscheinlichkeit noch weiter zu verringern, dass im Matrix-Schritt kein Quadrat berechnet wird. Genauer gesagt vergrößert man $\mathcal{F}_{k_3}$ etwas, sucht entsprechend noch einige weitere glatte Werte und führt den Matrixschritt erneut durch.

Hat man δ_0 gefunden und damit das Inverse einer Quadratwurzel von γ modulo $\mathcal{Q}$, kann man folgende Newtoniteration berechnen:

$$\delta_j = \frac{\delta_{j-1}(3 - \delta_{j-1} \cdot \gamma)}{2} \bmod \mathcal{Q}^{2^j}$$

Dabei bedeutet modulo $\mathcal{Q}^{2^j}$, dass die Koeffizienten a_i in den Ausdrücken $\sum_i a_i \cdot \rho^i$ modulo q^{2^j} berechnet werden. Dadurch erhält man Werte $\delta_1, \delta_2, ...$, für die $\delta_j^2 \cdot \gamma = 1 \bmod \mathcal{Q}^{2^j}$ ist. Dies wird solange fortgesetzt, bis q^{2^j} zumindest doppelt so groß ist wie eine Schranke, unter der die Koeffizienten einer Wurzel β von γ in $\mathbb{Z}[\rho]$ liegen müssen.

Zuletzt kann β berechnet werden durch $\beta = \delta_j \cdot \gamma \bmod \mathcal{Q}^{2^j}$. Den Test, ob $\beta^2 = \gamma$ auch wirklich zutrifft, kann man durchführen, um sich vom Probabilismus der vorherigen Schritte zu befreien, oder man versucht direkt n zu faktorisieren, da man sowieso mit signifikanter Wahrscheinlichkeit Erfolg hat.

Einige Ideen, um das Verfahren effizienter zu machen:

- Man könnte die Exponenten langsamer ansteigen lassen, z.B. $\mathcal{Q}^j$ statt $\mathcal{Q}^{2^j}$.
- Man kann in der Iteration schnelle Techniken für die Multiplikation benutzen.
- Man könnte die Iteration bereits abbrechen, falls nach einigen Schritten die Koeffizienten von $\delta_j \cdot \gamma \bmod \mathcal{Q}^{2^j}$ gleich geblieben sind.
- Man könnte sich eine Methode überlegen, die nicht direkt am Anfang das Produkt $\gamma = f'(\rho) \prod_{(a,b) \in S}(a + b\rho)$ ausmultipliziert.

Die bisherige „Einschränkung“, dass eine Primzahl $q > 2$ existiert, sodass $f \bmod q$ irreduzibel in $\mathbb{F}_q[X]$ ist, erfasst nahezu alle vorkommenden Fälle. Geht man für q sukzessive die Menge der Primzahlen, beginnend mit $3, 5, 7, ...$, durch und testet $f \bmod q$ auf Irreduzibilität, so wird man mit hoher Wahrscheinlichkeit nach wenigen Schritten bereits ein passendes q gefunden haben. Genauer gesagt ist $f(x)$ im Schnitt modulo jeder d-ten Primzahl irreduzibel.

Für die sehr selten auftretenden Spezialfälle, (z.B. wenn $d = 4$ und $n = m^4 + 1$) in denen man auf diesem Weg keinen Erfolg hat, kann man verschiedene Verfahren anwenden, um das Problem dennoch zu lösen. Am einfachsten ist es, ein Polynom zu f zu addieren, das $X - m$ teilt, oder ein anderes m in der *base-m*-Methode zu wählen. Die anderen in [15] angegebenen Verfahren sind wesentlich aufwändiger als die hier vorgestellte Newtoniteration, daher werde ich diese nicht näher betrachten. Außerdem hat dieser Algorithmus bei den zuletzt durchgeführten Iterationen enorme Laufzeitprobleme, weil die Koeffizienten sehr groß werden.

Methode nach Couveignes

Da die eben besprochene Iteration bei sehr großen Zahlen gegen Ende Laufzeitprobleme bereitet, hat Jean-Marc Couveignes in [16] einen sowohl einfachen wie auch effizienten Algorithmus entwickelt, um die Wurzel y aus γ modulo n zu berechnen. Die Idee ist, zunächst die Wurzel mit Koeffizienten modulo vieler verschiedener p_i zu berechnen, und das Ergebnis anschließend mit dem Chinesischen Restsatz zusammenzusetzen.

Die dabei verwendeten Primzahlen p_i müssen die Eigenschaft haben, dass $f(x) \bmod p_i$ irreduzibel ist, was die Frage nach einem Irreduzibilitätstest aufwirft, da z.B. der Test nach Eisenstein hierfür nicht geeignet ist. Außerdem wird ein effizienter Algorithmus benötigt, mit dem man die Wurzeln modulo der p_i berechnen kann. Um diese beiden Probleme werde ich mich gleich kümmern, doch zunächst einmal Couveignes Idee, falls dies beides gegeben ist.

Seien y_i die berechneten Wurzeln modulo p_i. Man beachte zuerst einmal, dass es sich dabei quasi um Polynome modulo $(f(x) \bmod p_i)$ handelt. Dann definiert man als Hilfsfunktionen

$$M = \prod_i p_i$$
$$M_i = \frac{M}{p_i}$$
$$a_i = M_i^{-1} \bmod p_i$$

Dann setzt man

$$z = \sum_i a_i \cdot M_i \cdot y_i$$

koeffizientenweise zusammen und erhält die Wurzel modulo M. Rundet man (wieder koeffizientenweise) den Quotienten $r = \left[\frac{z}{M}\right]$ auf eine ganze Zahl, dann gilt für die Wurzel modulo n

$$y = z - r \cdot M$$

Dabei muss außerdem beachtet werden, dass bei der Umrechnung von modulo M nach modulo n auch negative Koeffizienten entstehen können, daher müssen die einzelnen Koeffizienten noch abgeändert werden. Falls ein solcher $> \frac{M}{2}$ ist, so ist er negativ und es muss M subtrahiert werden. Dieser Hinweis ist leider weder in [16] noch in einer der anderen angegebenen Arbeiten zu finden.[6]

Die Berechnung von r sieht auf den ersten Blick recht aufwändig aus, es wird allerdings nur ein approximativer Wert benötigt. Dieser kann in Gleitkommadarstellung durch folgende Summe berechnet werden:

$$\begin{aligned}\frac{z}{M} &= \frac{\sum_i a_i \cdot M_i \cdot y_i}{M} \\ &= \sum_i \frac{a_i \cdot \frac{M}{p_i} \cdot y_i}{M} \\ &= \sum_i \frac{a_i \cdot y_i}{p_i}\end{aligned}$$

Der wichtigste Punkt bei diesem Vorgehen ist, dass alle Ergebnisse letztendlich nur modulo n oder p_i benötigt werden. Es kann also vollständig vermieden werden, mit solch gigantischen Koeffizienten rechnen zu müssen, wie sie am Ende der Methode nach Buhler, Lenstra und Pomerance vorkommen. Das Wurzelpolynom kann auch direkt durch Anwenden des Homomorphismus ψ quasi an der Stelle m modulo n ausgewertet werden. Es kommt also niemals eine Zahl größer als n vor.

Ein letztes Problem dieser Methode ist, dass immer zwei Wurzeln $\pm y$ existieren. Daher existieren auch immer zwei Wurzeln $\pm y_i \bmod p_i$, und es stellt sich die Frage, welche der beiden Wurzeln in den einzelnen Unterkörpern gewählt werden muss. Dies kann man durch die Norm beantworten.

Da in dieser Arbeit mit ungeradem Polynomgrad von $f(x)$ gearbeitet wird, hat man den Vorteil, dass für die Norm eines Elements α gilt: $N(-\alpha) = -N(\alpha)$. Da nach dem Matrix-Schritt die Norm von γ so konstruiert ist, dass in $\prod_{(a,b)\in S}(a+b\rho)$ nur Faktoren der algebraischen Faktorbasis in gerader Potenz enthalten sind, lässt

[6] Mein Dank gilt an dieser Stelle Prof. Dr. Seiler, der mir bei diesem Problem sehr geholfen hat.

sich diese Norm effizient berechnen:

$$\begin{aligned}
\gamma &= f'(\rho)^2 \cdot \prod_{(a,b)\in S} (a+b\rho) \\
\Rightarrow N(\gamma) &= N(f'(\rho))^2 \cdot \prod_{(a,b)\in S} N(a+b\rho) \\
\Rightarrow N(y) &= N(f'(\rho)) \cdot \sqrt{\prod_{(a,b)\in S} N(a+b\rho)} \\
&= N(f'(\rho)) \cdot \sqrt{\prod_{p_j \in \mathcal{F}_{k_2}} p_j^{2e_j}} \\
&= N(f'(\rho)) \cdot \prod_{p_j \in \mathcal{F}_{k_2}} p_j^{e_j}
\end{aligned}$$

An dieser Stelle wird auch klar, warum die Normen glatt und in gerader Potenz sein müssen. Ein wichtiger Punkt ist, dass die Norm jeweils nur modulo der p_i berechnet werden muss, ansonsten würden zu große Zahlen entstehen. Die Norm von $f'(\rho)$ bzw. eines beliebigen anderen Elements α des Zahlkörpers kann man durch

$$N(\alpha) = \alpha^{\frac{p_i^d - 1}{p_i - 1}} \bmod p_i$$

berechnen, denn da $f(x)$ irreduzibel ist, handelt es sich um einen Körper mit p_i^d Elementen. Dabei erhält man immer eine Zahl aus $\mathbb{F}_{p_i}$ und kein Polynom. Der Beweis dafür ist leider in keiner der angegebenen Quellen zu finden, daher wird er hier durchgeführt:[7]

$f(x) \in \mathbb{F}_{p_i}$ ist irreduzibel
$\Rightarrow \mathbb{F}_{p_i}[x] \bmod f(x)$ ist ein Körper $\mathbb{K}$ mit p_i^d Elementen
$\Rightarrow \mathbb{K}\backslash\{0\}$ ist zyklische Gruppe der Ordnung $p_i^d - 1$
$\Rightarrow \forall m \in \mathbb{N}$ mit $m | p_i^d - 1$ existiert genau eine Untergruppe
$U_m = \left\{ a^{\frac{p_i^d-1}{m} \cdot k} | k = 0, ..., m-1 \right\}$ der Ordnung m
$\Rightarrow U_m = \{b \in \mathbb{K}\backslash\{0\} | b^m = 1\}$
$\Rightarrow$ wähle $m_0 = p - 1$
$\Rightarrow \mathbb{F}_{p_i}\backslash\{0\} \subset \left\{b \in \mathbb{K}\backslash\{0\} | b^{p-1} = 1\right\} = U_{m_0}$
$U_{m_0} \subset \mathbb{F}_{p_i}\backslash\{0\}$ ist trivial

[7] An dieser Stelle danke ich Prof. Dr. Hertling für die Beweisidee.

Also kann man modulo p_i die zu α gehörende Wurzel von der von $-\alpha$ unterscheiden, indem man zuerst die Norm von $y \bmod p_i$ berechnet und anschließend eine Wurzel y_i von γ modulo p_i berechnet. Dann vergleicht man die Norm dieser Wurzel mir der Norm von y und falls diese nicht übereinstimmen, multipliziert man y_i mit -1.

Es bleibt jetzt noch zu klären, wie man die Wurzeln modulo p_i berechnet und wie der Test auf Irreduzibilität funktioniert.

Wurzeln modulo p

Um die Wurzel in $\mathbb{K} = \mathbb{Z}_p[\rho]$ zu berechnen, kann man den Algorithmus nach Shanks-Tonelli für Quadratwurzeln in $\mathbb{Z}_p$ benutzen, indem man ihn auf Zahlkörper verallgemeinert. In [23] ist dieser Weg beschrieben.

Dabei macht man sich zu Nutze, dass, falls $f(x) \bmod p$ irreduzibel ist, $\mathbb{K}$ ein Körper mit p^d Elementen ist und die Ordnung der multiplikativen Gruppe $\mathbb{K}^* = \mathbb{K}\backslash\{0\}$ gleich $p^d - 1$ ist. Dann gilt nach Euler für alle $x \in \mathbb{K}^*$

$$x^{p^d-1} = 1$$

und für jedes Quadrat $q = r^2$

$$q^{\frac{p^d-1}{2}} = (r^2)^{p^d-1} = 1$$

Da p ungerade ist, kann man $p^d - 1 = 2^k \cdot u$ für ungerades u berechnen. Dann ist q^u ebenfalls ein Quadrat mit der Wurzel r_u. Mit dieser Wurzel r_u folgt

$$\begin{aligned} q &= q^{u+1} \cdot q^{-u} \\ &= q^{u+1} \cdot r_u^{-2} \\ &= \left(q^{\frac{u+1}{2}} \cdot r_u^{-1}\right)^2 \\ &= r^2 \end{aligned}$$

Also ist eine Wurzel r von q gegeben durch

$$r = q^{\frac{u+1}{2}} \cdot r_u^{-1}$$

Sei G die Menge der Elemente aus $\mathbb{K}^*$ mit Ordnung 2^k. Dann ist G eine zyklische Untergruppe von $\mathbb{K}^*$ und $q^u \in G$. Also gibt es erzeugende Elemente w von G mit Ordnung 2^k und genau einen Exponenten e mit

$$w^e = q^u$$

Da q^u ein Quadrat ist, muss e gerade sein, und es gilt

$$r_u = w^{\frac{e}{2}}$$

Ein Erzeugendes w kann leicht gefunden werden, indem man ein Nichtquadrat a in $\mathbb{K}^*$ sucht, dessen Ordnung bekanntlich durch 2^k teilbar ist. Also hat $w = a^u$ genau Ordnung 2^k. Da die Hälfte der Elemente aus $\mathbb{K}^*$ Nichtquadrate sind, kann man durch Ausprobieren sehr schnell eines finden.

Der Exponent lässt sich bitweise bestimmen (siehe [23]) oder durch Ausprobieren berechnen, da er normalerweise eine kleine durch 2 teilbare Zahl ist.

Das Inverse eines Elements $a \in \mathbb{K}^*$ lässt sich auch einfach bestimmen, da Folgendes gilt:

$$1 = a^{p^d-1} = a^{p^d-2} \cdot a$$

und somit

$$a^{-1} = a^{p^d-2}$$

Wie im Standardfall gibt es noch drei Spezialfälle, bei denen man die Wurzel einfacher berechnen kann:

- $p^d = 3 \bmod 4$: In diesem Fall ist $\frac{p^d+1}{4}$ eine ganze Zahl und

  $$q = q^{\frac{p^d-1}{2}} \cdot q = q^{\frac{p^d+1}{2}} = \left(q^{\frac{p^d+1}{4}}\right)^2$$

  Also ist

  $$r = q^{\frac{p^d+1}{4}}$$

- $p^d = 5 \bmod 8$: Dann gilt $\frac{p^d-1}{4} = \pm 1$.

  1. Falls $\frac{p^d-1}{4} = 1$ gilt

     $$q = q^{\frac{p^d-1}{4}} \cdot q = q^{\frac{p^d+3}{4}} = \left(q^{\frac{p^d+3}{8}}\right)^2$$

     Also ist

     $$r = q^{\frac{p^d+3}{8}}$$

  2. Falls $\frac{p^d-1}{4} = -1$ gilt

     $$2^{\frac{p-1}{2}} = -1$$

     und

     $$q = -1 \cdot q^{\frac{p^d-1}{4}} \cdot q = 2^{\frac{p-1}{2}} \cdot q^{\frac{p^d+3}{4}} = \left(2^{\frac{p-1}{4}} q^{\frac{p^d+3}{8}}\right)^2$$

     Also ist

     $$r = 2^{\frac{p-1}{4}} \cdot q^{\frac{p^d+3}{8}}$$

Irreduzibilität modulo p

Da der Test nach Eisenstein für Polynome mit Koeffizienten modulo p nicht funktioniert, muss man andere Verfahren anwenden. Der Berlekamp-Algorithmus zum Faktorisieren von Polynomen[8] liefert in einer einfacheren Variante die Anzahl der Faktoren des Polynoms. Diesen möchte ich hier vorstellen.

Sei $(f(x) \bmod p)$ das zu testende Polynom vom Grad d. Dann berechnet man für $k = 0, ..., d-1$ die Polynome

$$x^{p \cdot k} = q_{k,0} + q_{k,1} \cdot x + ... + q_{k,d-1} \cdot x^{d-1} \bmod (f(x) \bmod p)$$

und konstruiert die Matrix

$$Q = \begin{pmatrix} q_{0,0} & \cdots & q_{0,d-1} \\ \vdots & \ddots & \vdots \\ q_{d-1,0} & \cdots & q_{d-1,d-1} \end{pmatrix} - I$$

wobei I die Einheitsmatrix ist.

Anschließend muss man den Rang dieser Matrix bestimmen, indem man diese in Dreiecksform bringt und die Anzahl der von Null verschiedenen Zeilenvektoren bestimmt. Die Anzahl der irreduziblen Faktoren in $f(x)$ ist gleich $d - rang(Q)$ und falls dabei 1 herauskommt, so ist $f(x)$ irreduzibel modulo p.

5.5 Laufzeitanalyse

Die ausführliche Analyse der Laufzeit des GNFS ist eine sehr mathematische und aufwändige Angelegenheit. Es soll aber trotzdem versucht werden, zumindest die Ideen zu vermitteln wie die Laufzeit von

$$e^{((\frac{64}{9})^{\frac{1}{3}} + o(1)) \cdot (\log(n))^{\frac{1}{3}} \cdot (\log(\log(n)))^{\frac{2}{3}}}$$

Schritten, bzw.

$$L_n\left(\frac{1}{3}, (\frac{64}{9})^{\frac{1}{3}} + o(1)\right)$$

ermittelt werden kann.

Dazu sei zunächst, wie im Quadratischen Sieb, die Anzahl der B-glatten Zahlen kleiner als x definiert durch

$$\psi(x, B) := |\{m \mid 1 \leq m \leq x, m \text{ ist } B\text{-glatt}\}|$$

[8] siehe z.B. [7],

Seien weiter n, d mit $n > d^{2d^2} > 1$ und $u, B \in \mathbb{R}$ mit der Eigenschaft, dass die Zahl

$$x = 2 \cdot d \cdot n^{\frac{2}{d}} \cdot u^{d+1}$$

die Bedingung

$$\frac{u^2 \cdot \psi(x, B)}{x} \geq g(B)$$

erfüllt, wobei $g(x)$ eine Funktion ist für die $g(x) \geq 1$ und $g(x) = x^{1+o(1)}$ für $x \to \infty$ gilt. Dann kann man zeigen, dass für $n \to \infty$ Folgendes gilt:

$$2\log(u) \geq (1 + o(1)) \cdot \left(d\log(d) + \sqrt{(d\log(d))^2 + 4\log(n^{\frac{1}{d}}) \cdot \log(\log(n^{\frac{1}{d}}))} \right)$$

Die Anzahl der Schritte entspricht mindestens der Größe des rationalen Siebs, also u^2. Mit der gerade festgestellten Abschätzung erhält man so direkt eine untere Schranke für die Laufzeit, nämlich

$$u^2 \geq e^{(1+o(1)) \cdot \left(d \cdot \log(d) + \sqrt{(d \cdot \log(d))^2 + 4 \cdot \log(n^{\frac{1}{d}}) \cdot \log(\log(n^{\frac{1}{d}}))} \right)}$$

Die Bedingung

$$\frac{u^2 \cdot \psi(x, B)}{x} \geq g(B)$$

ist erfüllt, da die Anzahl der Zeilen mindestens so groß ist wie die Anzahl der Spalten in der Matrix. Die Spaltenanzahl ist mindestens so groß wie die Anzahl der Elemente der Faktorbasis, also $\pi(B)$, was $B^{1+o(1)}$ für $B \to \infty$ legitimiert.

Um die Zahl der Zeilen abzuschätzen, braucht man eine Schranke für die Anzahl der im Sieb enthaltenen glatten Werte. Für $-u \leq a \leq u$ und $0 < b \leq u$ ist die Zahl $(a + bm)N(a + b\rho)$ maximal

$$\begin{aligned}
(u + um) \cdot (d + 1) \cdot m \cdot u^d &= (d + 1 + md + m) \cdot m \cdot u^{d+1} \\
&= (d + 1 + m(d + 1)) \cdot m \cdot u^{d+1} \\
&= (d + 1) \cdot (1 + m) \cdot m \cdot u^{d+1} \\
&\leq 2 \cdot d \cdot m^2 \cdot u^{d+1} \\
&\leq 2 \cdot d \cdot n^{\frac{2}{d}}
\end{aligned}$$

da die Koeffizienten von f durch m begrenzt sind und $m \leq n^{\frac{1}{d}}$. Also ist die oben definierte Zahl x eine Grenze für die im Sieb enthaltenen glatten Zahlen. Die Wahrscheinlichkeit, dass eine Zahl kleiner x glatt ist, ist wie bereits im QS

$$\frac{\psi(x, B)}{\lceil x \rceil}$$

Wir betrachten von den insgesamt $2u \cdot u$ Zahlen im Siebfeld nur Zahlen, für die $ggT(a,b) = 1$ gilt. Diese Anzahl kann mit $\frac{12}{\pi^2} \cdot u^2$ abgeschätzt werden, woraus eine Zeilenanzahl von

$$\frac{12}{\pi^2} \cdot \frac{u^2 \cdot \psi(x,B)}{x}$$

folgt. Lässt man noch die Konstante $\frac{12}{\pi^2}$ in $B^{o(1)}$ verschwinden, so ist die geforderte Bedingung erfüllt.

Als nächstes zeigt man für den Fall der Gleichheit der obigen Bedingung

$$B_0^2 = u_0^2 = e^{(1+o(1))\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

bzw.

$$B_0 = u_0 = e^{(\frac{1}{2}+o(1))\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

und

$$x_0 = 2 \cdot d \cdot n^{\frac{2}{d}} \cdot u_0^{d+1}$$

dass gilt:

$$\frac{u_0^2 \cdot \psi(x_0, B_0)}{x_0} = B_0^{1+o(1)}$$

Wählt man u und B etwas größer, also sei für ein sehr kleines $\epsilon \in \mathbb{R}$

$$B = u = e^{\left(\frac{1+\epsilon}{2}+o(1)\right)\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

und

$$x = 2 \cdot d \cdot n^{\frac{2}{d}} \cdot u^{d+1}$$

dann kann man ebenfalls zeigen, dass in diesem Fall gilt:

$$\frac{u^2 \cdot \psi(x,B)}{x} = B^{1+o(1)}$$

Also ist

$$e^{(1+o(1))\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

auch eine obere Schranke für die Anzahl der Schritte. Jetzt muss man nur noch d so wählen, dass dieser Ausdruck minimiert wird, und man erhält als Resultat

$$d = (3^{\frac{1}{3}} + o(1)) \cdot \left(\frac{\log(n)}{\log(\log(n))}\right)^{\frac{1}{3}}$$

Setzt man dies oben ein, so erhält man die Laufzeit von

$$L_n\left(\frac{1}{3}, (\frac{64}{9})^{\frac{1}{3}} + o(1)\right)$$

5.6 Verbesserungen

Selbstverständlich können die gleichen Verbesserungen wie im QS benutzt werden, um das Sieben zu beschleunigen, insbesondere das Sieben mit Logarithmen und Large Primes sollte verwendet werden. Abgesehen davon gibt es noch zahlreiche weitere Verbesserungen, von denen hier einige wenige kurz angesprochen werden sollen.

5.6.1 Homogene Polynome

Buhler, Lenstra und Pomerance schlagen in [15] vor, ein homogenes Polynom in zwei Variablen zu verwenden um die Koeffizienten kleiner zu machen. Damit soll sich die Laufzeit nochmals verbessern, da z.B. statt eines Polynoms fünften Grades zwei Polynome dritten Grades verwendet werden können.

Dazu wählt man Zahlen m_1, m_2, den Grad d und ein homogenes Polynom

$$f(x,y) = c_d \cdot x^d + c_{d-1} \cdot x^{d-1} \cdot y + \cdots + c_1 \cdot x \cdot y^{d-1} + c_0 \cdot y^d \in \mathbb{Z}[x,y]$$

mit kleinen Koeffizienten c_i und $f(m_1, m_2) = 0 \bmod n$.

Zwar werden die einzelnen Schritte des GNFS dadurch ein wenig komplizierter, im Wesentlichen bleibt der Ablauf allerdings der gleiche. Eine detaillierte Beschreibung der Methode ist z.B. in [15] zu finden.

5.6.2 Polynomgüte

Wählt man in der base-m-Methode verschiedene Werte für m, z.B. $\lceil n^{\frac{1}{d}} \rceil$, $\lceil n^{\frac{1}{d}+1} \rceil$, $\lceil n^{\frac{1}{d}+2} \rceil$, usw. dann erhält man jeweils unterschiedliche Polynome. Um von mehreren Polynomen das sinnvollste auszuwählen, kann man für diese einige Testläufe machen und das auswählen, bei dem am meisten glatte Werte erzeugt wurden. Dieses einfache Kriterium nennt man auch *Polynomgüte*.

Eine andere Idee Polynome zu bewerten ist, die mittlere Größe der Koeffizienten zu betrachten. Es gibt auch Ansätze, die die Polynome generell homogenisieren, quadrieren und integrieren, allerdings sind zu diesen eher „abenteuerlichen“ Ansätzen fast keine Arbeiten mit vernünftigem mathematischen Hintergrund zu finden.

Letztendlich ist die Polynomwahl in gewisser Weise Glückssache, denn selbst wenn ein Polynom bei einem Testlauf besonders gut abschneidet, muss das noch nicht heißen, dass nicht ein besseres existiert oder dass sich die Güte beim Sieben nicht verschlechtert. Dies ist wohl das Forschungsgebiet, auf dem noch am meisten entdeckt werden kann. Eine sehr ausführliche Arbeit, die sich ausschließlich mit der Polynomwahl beschäftigt, ist [22]. Hier werden unter anderem auch

die für bereits faktorisierten Zahlen wie z.B. RSA-155[9] verwendeten Verfahren beschrieben und verglichen.

5.6.3 Das ggT-Sieb

Um sich beim Aufstellen von T_0 unzählige ggT-Berechnungen zu sparen, kann man ein Sieb verwenden. Dazu initialisiert man wie oben das Siebfeld $T_0[a, b]$ mit 1. Für fixiertes b berechnet man dann die Primfaktoren $q_1, ..., q_k$ von b und bestimmt für alle q_j das kleinste a mit $q_j|a$. Für dieses (a, b) ist $ggT(a, b) > 1$, da sowohl a als auch b durch q_i teilbar ist. Also kann man den Wert streichen und trägt in $T_0[a, b]$ eine 0 ein. Dies kann man auch für alle $a + l \cdot q_i$ tun, analog zum Verfahren im QS.

```
Eingabe: u
Ausgabe: T_0
for(int b=1; b<=u; b++){
   q=e(b);
   for(int i=0, i<q.size, i++){
     int a=-u mod n;
     while(a mod q[i] != 0)a++;
     for(a, a<u mod n, a=a+q[i] mod n){
       T_0[a, b]=0;
     }
   }
}
return(T_0)
```

Abbildung 5.3: ggT-Sieb

Am Ende des Algorithmus in Abbildung 5.3 steht nur noch an den Stellen eine 1, für die $ggT(a, b) = 1$ gilt. Dieses Siebfeld kann man dann direkt an das rationale Sieb weiterreichen.

Zur Berechnung der Primfaktorzerlegung von b kann man z.B. die Elliptic Curves Method verwenden, da diese für kleine Faktoren am besten geeignet ist. Dies verbessert die Laufzeit gegenüber trivialer Division.

[9] siehe `http://www.rsasecurity.com/rsalabs/node.asp?id=2098`

5.6.4 Lattice Sieve

Die in 5.3.1 beschriebene Siebvariante wird auch als *Line Sieving* bezeichnet. Eine andere Möglichkeit stellt das so genannte *Lattice Sieving* dar, Details findet man in [14].

Die Idee ist, ein Element q der rationalen Faktorbasis zu fixieren und die folgende Menge zu berechnen:

$$L_q = \{(a, b) \in \mathbb{Z}^2 | a + b \cdot m = 0 \bmod q\}$$

Diese Menge stellt ein Gitter (Lattice) in $\mathbb{Z}^2$ dar, weshalb man eine minimale Basis $u = (u_1, u_2), v = (v_1, v_2)$ berechnen kann, sodass alle Elemente $(a, b) \in L_q$ geschrieben werden können als

$$(a, b) = \lambda \cdot u + \mu \cdot v$$

bzw.

$$a = \lambda \cdot u_1 + \mu \cdot v_1$$
$$b = \lambda \cdot u_2 + \mu \cdot v_2$$

Für die Elemente von L_q gilt $a + b \cdot m = 0 \bmod q$. Falls $u_1 + m \cdot u_2 \neq 0 \bmod q$ folgt

$$\begin{aligned} &\lambda \cdot u_1 + \mu \cdot v_1 + (\lambda \cdot u_2 + \mu \cdot v_2) \cdot m = 0 \bmod q \\ \Leftrightarrow\ &\lambda \cdot (u_1 + m \cdot u_2) = -\mu \cdot (v_1 + m \cdot v_2) \bmod q \\ \Leftrightarrow\ &\lambda = -\mu \cdot \frac{v_1 + m \cdot v_2}{u_1 + m \cdot u_2} \bmod q \end{aligned}$$

also ist q ein Teiler von λ. Falls $u_1 + m \cdot u_2 = 0 \bmod q$, muss $v_1 + m \cdot v_2 \neq 0 \bmod q$ sein, da $\{u, v\}$ eine minimale Basis ist, und dann ist q ein Teiler von μ. In diesem Fall vertauscht man λ und μ und rechnet damit weiter.

Die Gleichung kann als Basis für ein Sieb benutzt werden. Dazu fixiert man ein μ, berechnet den zugehörigen Wert für λ und siebt mit dem Abstand q.

Diese Idee kann außerdem auch zum Sieben über der algebraischen Faktorbasis benutzt werden. Sei (p_j, r_j) ein Element der algebraischen Faktorbasis. Dann muss man oben lediglich m durch r_j ersetzen, dann funktioniert das Sieb analog.

5.6.5 Large primes

Ähnlich wie im QS können im GNFS large primes verwendet werden, um die Suche nach glatten Werten zu beschleunigen. Dabei ist die Variante mit *partial relations* fast analog zum Verfahren im QS. Für die Variante mit *double partial*

relations wird im GNFS die so genannte *Quadruple Large Prime* (QLP)-Variante eingeführt, die wesentlich aufwändiger ist.

Um die Vorteile von large primes ausnutzen zu können, müssen Verfahren implementiert werden, die auf der Zyklensuche in gerichteten Graphen basieren. Eine interessante Eigenart der QLP ist, dass bei „relativ kleinen“ Faktorbasen ab einem gewissen Punkt eine Art Kettenreaktion ausgelöst wird, wodurch zahlreiche double partial relations auf einen Schlag zu glatten Werten kombiniert werden können. Dies lässt sich durch das Entstehen zahlreicher Zusammenhangskomponenten durch das Einfügen nur einer einzigen Komponente in den aufgebauten Graphen erklären. Ist dieser Punkt einmal erreicht, ist die Siebphase praktisch beendet.

Dieses Phänomen ist ein interessanter Ansatz, um die Laufzeit weiter zu verbessern. Wenn man erreichen könnte, dass diese Kettenreaktion bereits viel früher eintritt, dann würde der Algorithmus insgesamt wesentlich schneller ablaufen. Näheres hierzu findet man z.B. in [23].

5.6.6 Wurzeln nach Montgomery

Peter Montgomery hat in [19] einen anderen (möglicherweise schnelleren) Algorithmus vorgestellt, mit dem man die algebraische Wurzel berechnen kann. Es handelt sich um ein iteratives Verfahren, das in jedem Iterationsschritt die Wurzel von γ etwas genauer approximiert, indem ein Produkt

$$\gamma = \gamma_i \cdot \prod_i r_i^{2 \cdot c_i}$$

iterativ erweitert wird. Dabei ist γ_i ein Korrekturterm, der schrittweise einfacher wird, bis letzten Endes aus diesem direkt die Wurzel berechnet werden kann. Die endgültige Wurzel ist dann

$$r = r_n = \sqrt{\gamma_n} \cdot \prod_{i=1}^{n} r_i^{c_i}$$

Dieser Algorithmus ist im Wesentlichen der LLL-Algorithmus zur Reduktion von Gitterbasen, angewendet auf Ganzheitsbasen von Zahlkörpern. Da Couveignes Methode in den Tests letztendlich funktioniert hat, sei für weitere Details auf Montgomerys Arbeit verwisen.

5.6.7 Quadratwurzeln verkleinern

Um das Ziehen der algebraischen Wurzel effizienter zu machen, kann man γ durch ein anderes Element ersetzen, das kleinere Koeffizienten hat, wenn man es als Po-

lynom in ρ ausdrückt. Sei $S = \{(a_1, b_1), ..., (a_s, b_s)\} \subset T_2$ die errechnete Teilmenge. Wir definieren induktiv zwei Folgen $(\mu_i)_{i=0}^s$ und $(\nu_i)_{i=0}^s$ mit $\mu_0 = \nu_0 = 1$ wie folgt:

- Falls $a_i + b_i\rho | \mu_{i-1}$ in $\mathbb{Z}[\rho]$, dann setze $\mu_i = \dfrac{\mu_{i-1}}{a_i + b_i\rho}$ und $\nu_i = \nu_{i-1}(a_i + b_i\rho)$
- Sonst setze $\mu_i = \mu_{i-1}(a_i + b_i\rho)$ und $\nu_i = \nu_{i-1}$

Dann gilt folgende Gleichung:

$$\gamma = f'(\rho)^2 \prod_{i=1}^{s}(a_i + b_i\rho) = f'(\rho)^2 \mu_s \nu_s^2$$

sodass γ genau dann ein Quadrat in $\mathbb{Z}[\rho]$ ist, wenn $f'(\rho)\mu_s$ dies ist. Daher reicht es die Wurzel von $f'(\rho)\mu_s$ zu ziehen und mit ν_s zu multiplizieren.

5.6.8 Spezialhardware

Um Zahlen noch schneller faktorisieren zu können, wurde Hardware entwickelt, die speziell auf den Sieb- und den Matrix-Schritt zugeschnitten ist. Dazu wurde von Adi Shamir zunächst das so genannte TWINKLE-Device vorgestellt. Anschließend folgten weitere Verbesserungen und Vorschläge wie mesh-sieving, es entwickelte sich hier sozusagen ein neues Forschungsgebiet.

Letztendlich ist die Entwicklung derzeit beim so genannten TWIRL-Device angekommen, das in [30] vorgestellt wird, und einen 512-bit-RSA-Modulus in nur 10 Minuten bei Kosten von $10.000 faktorisieren kann.

Kapitel 6

Implementierung

Dieses Kapitel ist der Beschreibung der Implementierungen gewidmet. Es enthält einige Dinge, die mir bei den Implementierungen der Algorithmen aufgefallen sind und teilweise nicht in den angegebenen Arbeiten zu finden sind.

Die Programme sind in C++ geschrieben, da es für diese Programmiersprache die effizientesten Implementierungen der Bibliotheken gibt, die ich verwendet habe.[1]

Diese Bibliotheken sind unter anderem GMP[2] für beliebig lange Ganzzahlen und optimierte Rechenoperationen, ATLAS[3] als Grundlage für sämtliche lineare Algebra und LinBox[4] mit den Implementierungen verschiedener Algorithmen zum Lösen von Gleichungssystemen.

Außerdem habe ich mit den Entwicklern von LinBox zahlreiche Informationen ausgetauscht. Dabei konnte ich diese auf einige Fehler hinweisen und musste leider feststellen, dass die optimierte Datenstruktur $GF(2)$ vor allem mit den oben angesprochenen binären Operationen im Block-Lanczos-Algorithmus leider zum Zeitpunkt der Fertigstellung dieser Arbeit noch nicht fertiggestellt ist.

Als letztes möchte ich noch anmerken, dass ich jeweils nur die einfachste Version der Algorithmen programmiert habe, ohne jegliche der beschriebenen Verbesserungen. Ich habe mit Kommentaren an den entsprechenden Stellen angemerkt, welche Verbesserungen man jeweils noch einfügen könnte.

[1] Außerdem hat in dieser Frage letzten Endes mein persönlicher Geschmack entschieden.

[2] `http://www.swox.com/gmp/`

[3] `http://math-atlas.sourceforge.net/`

[4] `http://www.linalg.org`

6.1 Quadratisches Sieb

Als erstes stellte sich mir die Frage, wie der Parameter

$$B = e^{\left(\frac{1}{2}+o(1)\right)\cdot\sqrt{\log(x)\cdot\log(\log(x))}}$$

zu berechnen ist. Den $o(1)$-Teil kann man zwar prinzipiell als Stellvertreter für eine gegen null konvergierende Funktion auffassen, allerdings sollte er für kleine zu zerlegende Zahlen nicht vernachlässigt werden. Angenommen wir faktorisieren die 80-bit-Zahl

$$519353750868850510922311$$

Dann ist der Wurzel-Term ungefähr $14,78$. Lässt man den $o(1)$-Term einfach weg, dann ist $B = 1619$. Setzt man hingegen $0,1$ ein, so folgt $B = 7099$, was bereits bei so „kleinen“ Zahlen ein deutlicher Unterschied ist.

Eine interessante Erkenntnis bei der Implementierung des QS entstand beim Testen des Siebschritts. Da ich das Problem hatte, nicht genug glatte Werte zwischen $\sqrt{n}$ und $\sqrt{2 \cdot n}$ zu finden, habe ich das Siebarray näher betrachtet. Dabei stellte ich fest, dass zwar beim Sieben mit dem Faktorbasiselement p_j immer an allen p_j-ten Stellen im Siebarray geteilt wurde, allerdings hätte man außerdem noch zwischen diesen Stellen teilen können, da ich in meinem Fall sogar an einer Stelle im Array direkt den Eintrag p_j stehen hatte.

Daher konnte ich folgern, dass man nach dem Suchen des ersten Eintrags im Siebarray, der p_j teilt, außerdem noch so viele Einträge weiter testen sollte bis p_j erreicht ist und falls einer dieser Einträge durch p_j teilbar sein sollte kann man nochmals sieben. Dadurch erhält man ein Vielfaches an glatten Werten und kann so das Sieben noch weiter beschleunigen.

Diesen Hinweis konnte ich leider in keiner der angegebenen Arbeiten finden. Da die Anzahl der gefundenen glatten Werte dadurch aber um ein Vielfaches steigt, sollte diese Idee unbedingt benutzt werden.

Beim Testen mit immer größeren Zahlen bin ich natürlich irgendwann an die Grenzen des Hauptspeichers gestoßen. Da das Siebarray immer größer wurde, konnte es relativ schnell nicht mehr komplett im Hauptspeicher abgelegt werden. Daher habe ich beschlossen, eine Grenze einzuführen, ab der das Sieben in mehreren Phasen durch abschnittsweises Sieben durchgeführt wird. Dies würde auch eine Parallelisierung sehr einfach machen, indem die verschiedenen Siebintervalle auf unterschiedliche Rechner oder Prozessoren verteilt werden.

Zum Lösen des Gleichungssystems habe ich zwei Algorithmen aus LinBox benutzt und verglichen, den Block-Lanczos-Algorithmus nach Montgomery aus [25] und den Lookahead-based Block-Lanczos-Algorithmus von Bradford Hovinen aus [26], der um einiges schneller ist, siehe Abbildung 6.1.

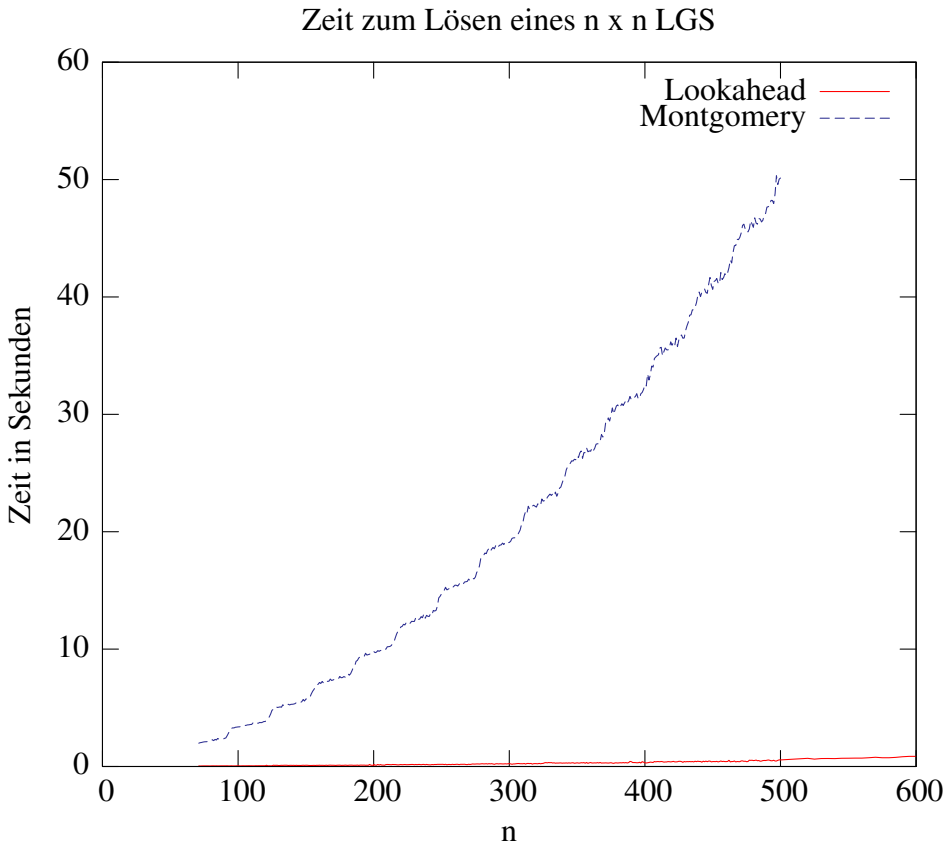

Abbildung 6.1: Montgomery vs. Lookahead-based Block-Lanczos

Dabei ist mir aufgefallen, dass bei meinen Tests der Algorithmus nach Montgomery in $98,8\%$ der Fälle fehlschlägt und keine Lösung liefert und der Lookahead-Algorithmus in $0,465\%$ der Fälle. Auf Anfrage beim LinBox-Projekt bekam ich Antwort von Bradford Hovinen, der mir mitteilte, dass dies an der Matrix liegt und dass man einen so genannten *Preconditioner* darauf anwenden muss. Am besten wäre ein Sparse Preconditioner nach Wayne Eberly, [5] allerdings kann man auch einen wesentlich einfacheren *Random Sparse Preconditioner* benutzen.

Statt das Gleichungssystem $A \cdot x = 0$ mit der $k \times k$-Matrix A zu lösen, erzeugt man zwei zufällige $k \times k$-Matrizen U und V und löst das Gleichungssystem für die Matrix $U \cdot A \cdot V$. Findet man eine Lösung x für dieses Gleichungssystem, dann ist $V \cdot x$ eine Lösung für das ursprüngliche Gleichungssystem, falls U vollen Spaltenrang hat. Die Wahrscheinlichkeit dafür lässt sich wiederum erhöhen,

[5] siehe `http://pages.cpsc.ucalgary.ca/~eberly/Research/Papers/nullspace_and_rank.pdf`

indem man für U eine zufällige $k \times (k+l)$-Matrix wählt und für V eine zufällige $(k+l) \times k$-Matrix.

Für diese Ideen existieren zur Zeit leider (noch) keine Quellen, sie stammen ausschließlich aus meinen eMail-Konversationen mit Bradford Hovinen und wurden von uns gemeinsam entwickelt.

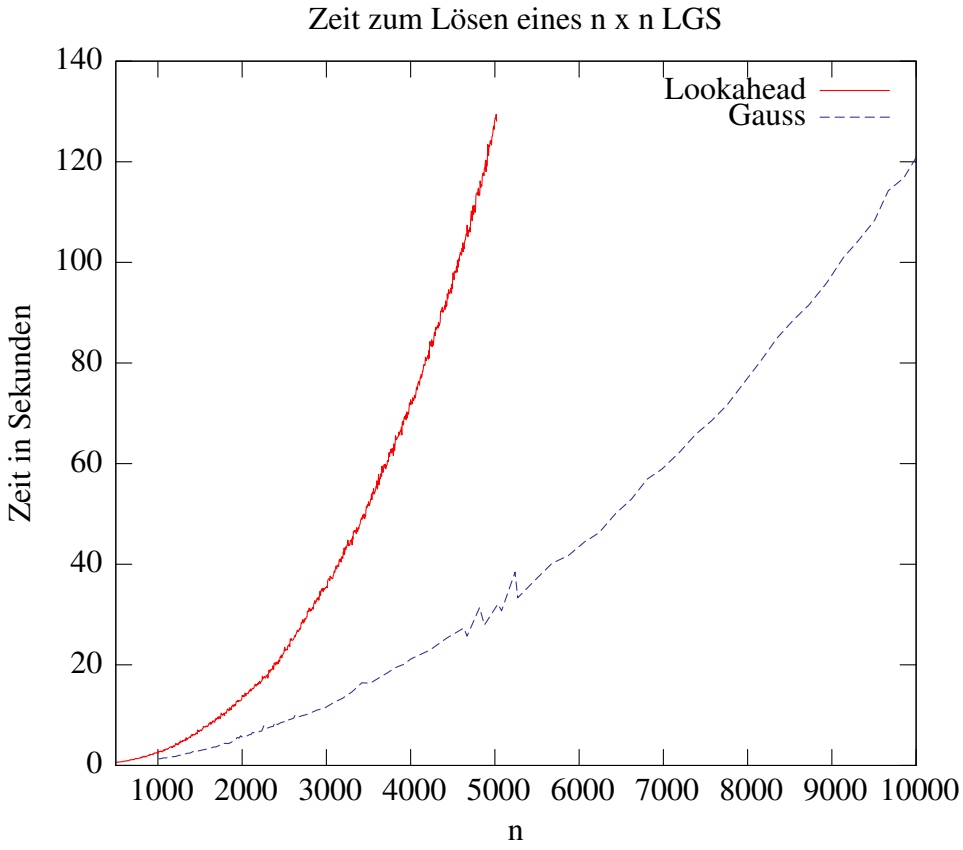

Abbildung 6.2: Gauss vs. Lookahead-based Block-Lanczos

Der Sinn dies überhaupt zu tun ist, den Aufbau der zu lösenden Matrix zu verändern. Nach Konstruktion enthält diese nämlich in den ersten Zeilen wesentlich mehr Einsen als weiter unten, da kleinere Faktorbasiselemente häufiger in der Faktorisierungen der glatten Werte vorkommen. Im mittleren Teil, genauer gesagt von Zeile k_1 auf Zeile $k_1 + 1$, ist dann wieder ein Übergang von sehr wenigen auf sehr viele Einsen, da die letzten Elemente der rationalen Faktorbasis Zeilen mit sehr wenigen Einsen erzeugen, dagegen erzeugen die ersten Elemente der algebraischen Faktorbasis wieder Zeilen mit sehr vielen Einsen. Ganz am Ende der Matrix sind einige Zeilen mit den Quadratic Characters, die zu 50% Einsen ent-

halten.

Zum Vergleich habe ich einmal die normale Gauss-Elimination implementiert, ohne irgendwelche Verbesserungen wie z.B. Matrix-Reduktion. Diese schneidet, wie man in Abbildung 6.2 sieht, im Vergleich erstaunlich gut ab, was zum einen daran liegt, dass im Lanczos-Algorithmus der Vorteil von Bitoperationen noch nicht ausgenutzt wird. Dies würde die Performance im hier vorliegenden Fall um den Faktor 64 verbessern. Zum anderen bekam ich die auf Anfrage die Auskunft, dass erst ab einer Matrixgröße von über 100000 Einträgen der Schnittpunkt erreicht ist, ab dem Lanczos schneller ist. Allerdings ist das aus der Abbildung nicht gerade ersichtlich. Die Erklärung dafür ist, dass beim Gauss-Algorithmus ab einer gewissen Größe des Gleichungssystems der Speicherplatz ausgeht. Ab diesem Punkt ist die iterative Lanczos-Methode im Vorteil.[6]

6.2 Number Field Sieve

Auch bei der Implementierung dieses Algorithmus gibt es einige Dinge zu beachten. Zuerst einmal fällt im Gegensatz zum QS der $o(1)$-Term bei der Polynomwahl nicht wirklich ins Gewicht. Der Polynomgrad ist

$$d = (3^{\frac{1}{3}} + o(1)) \cdot \left(\frac{\log(n)}{\log(\log(n))}\right)^{\frac{1}{3}}$$

und wird bei mir auf die am nächsten gelegene ungerade Zahl gerundet. Setzt man wie oben für $o(1) = 0,1$ ein und berechnet den Polynomgrad für die 200-bit-Zahl

$$92827498322909451078585756463220331362632877445417869162 5613$$

so bekommt man ohne $o(1)$ den Wert $4,385$ und mit $o(1)$ den Wert $4,689$, was beides auf 5 gerundet wird.

Bei der Wahl von B und u hat dieser Term hingegen einen größeren Einfluss, da er hier im Exponenten steckt:

$$B = u = e^{\left(\frac{1}{2}+o(1)\right)\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

Nimmt man das gleiche Zahlenbeispiel, so bekommt man ohne $o(1) = 0,1$ einen Wert von 1808035 gegenüber 32258802. Da sich ein höherer Wert für B direkt auf die Größe des zu lösenden Gleichungssystems auswirkt, habe ich mich für die kleinst mögliche Variante entschieden. Dadurch ist der leicht parallelisierbare Sieb-Schritt der dominierende Teil des Algorithmus.

[6] nach Aussage von Bradford Hovinen.

Beim Testen auf Irreduzibilität wurde schnell deutlich, dass der Test nach Eisenstein aus 2.1.2 zu viele Polynome nicht als irreduzibel erkennt. Daher ist das Problem jetzt so gelöst, dass für die Liste der Primzahlen $\{p_i | p_i < 1000\}$ der Test auf Irreduzibilität nach Berlekamp in 5.4.2 für $f \bmod p_i$ durchgeführt wird. Ist f modulo einer dieser Primzahlen irreduzibel, so folgt dies auch für f selbst.

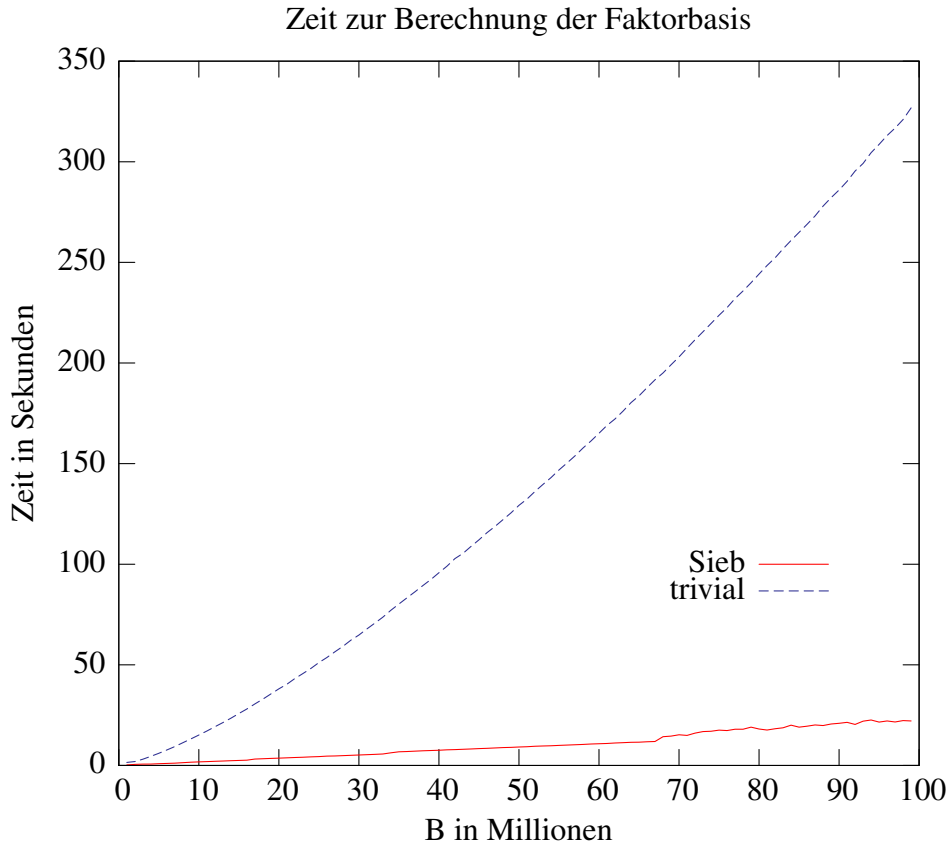

Abbildung 6.3: Trivialer Algorithmus vs. Sieb des Eratosthenes

Beim anschließenden Erstellen der rationalen Faktorbasis ist das Problem aufgetreten, dass ein trivialer Algorithmus zum Berechnen aller Primzahlen bis B viel zu lange braucht. Daher habe ich für diesen Zweck das bekannte *Sieb des Eratosthenes* implementiert. Abbildung 6.3 zeigt die Laufzeit im Vergleich.

Bei der Implementierung des Siebs habe ich primär den notwendigen Speicherplatz und die Parallelisierbarkeit beachtet. Daher ist der Siebschritt jetzt ein wenig anders als ursprünglich gedacht. Da über den Wert b am besten parallelisiert werden kann, wird dieser fixiert und in einem Siebarray über a iteriert. Dabei werden direkt hintereinander ggT-Berechnungen, rationales Sieben und algebrai-

sches Sieben für dieses eine fixierte b ausgeführt und die glatten Werte in eine Liste eingefügt. Anschließend kann dieses Siebarray wieder gelöscht werden.

Dies hat den Vorteil, dass sukzessive „Zeilen im Siebfeld“ vollständig abgearbeitet werden, was sehr einfach auf viele Rechner verteilt werden kann. Außerdem wird so der benötigte Speicherplatz minimiert, weil immer nur eine Zeile des Siebfelds abgespeichert werden muss. Desweiteren kann man abbrechen, falls genug glatte Werte gefunden sind bevor die Grenze von u erreicht ist.

Aus Gründen der Einfachheit siebe ich a nicht von $-u$ bis u, sondern nur von 0 bis $2 \cdot u$, was die Programmierung sehr vereinfacht. Ich spare mir auch den ersten Eintrag von -1 in den Exponentenvektoren, da keine negativen Werte für $a + bm$ vorkommen.

Eine interessante Feststellung ist, dass beim Sieben mit den verschiedenen Werten für b die Anzahl der glatten Werte besonders groß ist, wenn b eine Primzahl ist. Daher könnte man z.B. versuchen, erst einmal nur mit Primzahlen zu sieben, und nur falls das noch nicht reicht mit Nichtprimzahlen. Allerdings nimmt mit steigenden Primzahlen auch die Zahl der glatten Werte ab.

In Abbildung 6.4 ist die Anzahl der rational glatten Werte für $b = 1, ..., 23$ dargestellt, die bei der Faktorisierung der Zahl

$$69938810898180820962 6721$$

berechnet wurden. Die Anzahl der algebraische glatten Werte verhält sich genauso, wie man in Abbildung 6.5 sieht.

Beim anschließenden Matrix-Schritt erfolgt die Konstruktion der Matrix wie im Text oben beschrieben, erst hier werden die quadratic characters berechnet und in die Matrix mit übernommen. Anschließend wird das Gleichungssystem mit den gleichen Problemen wie im QS gelöst.

Nach dem Matrix-Schritt folgt das Berechnen der Wurzeln. Die rationale Wurzel ist wie erwartet extrem einfach zu berechnen, während die algebraische Wurzel lange und viele Nachforschungen erforderte. Das lag daran, dass kleine Teile der Idee von Couveignes einfach nicht dokumentiert waren. Daher musste ich[7] „die Idee neu zu Ende denken“ und ein Paar Details einfügen.

Außerdem hatte der Algorithmus anfangs enorme Laufzeitprobleme, weshalb ich noch ein Paar Optimierungen vornehmen musste. Die Wichtigste war das Entwickeln eines Square-and-multiply-Äquivalents für Elemente des Zahlkörpers.

Da beim Berechnen der Wurzeln modulo p für relativ kleine Primzahlen bereits Polynome $g(x)$ vom Grad 3 oder 5 mit Zahlen e der Größenordnung von Milliarden potenziert werden müssen, ist eine for-Schleife, in der immer wieder mit $g(x)$ multipliziert und das Restpolynom bei Polynomdivision durch $f(x)$ berechnet wird, nicht effizient genug. Beispielsweise musste bei der Faktorisierung

[7] zusammen mit Prof. Seiler.

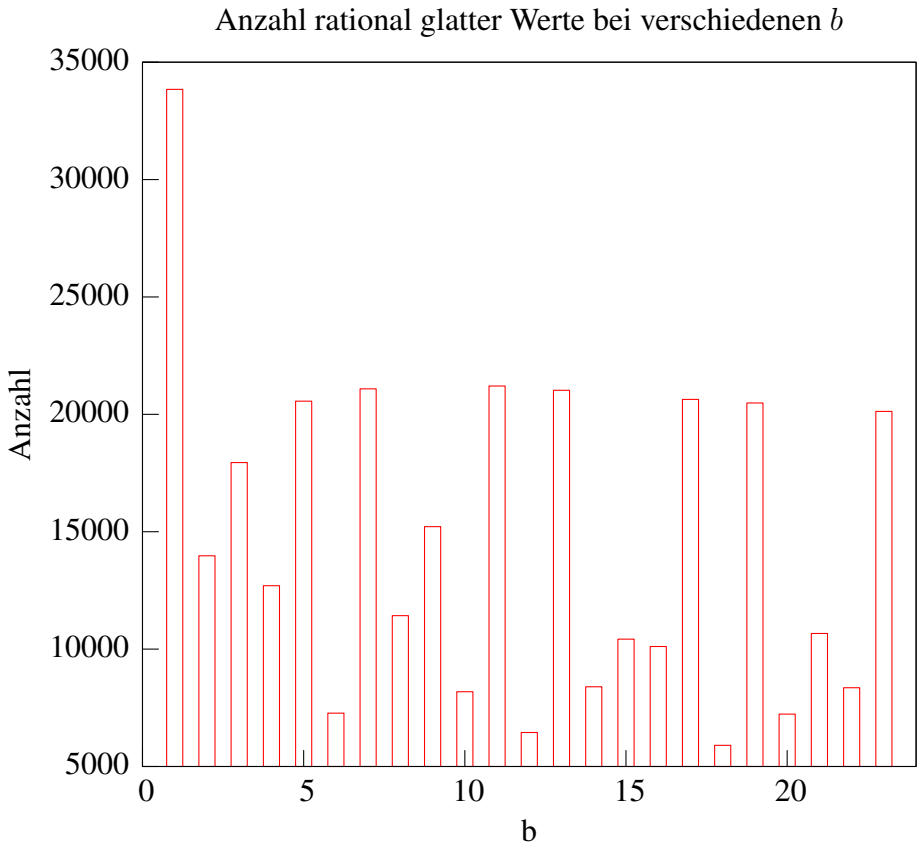

Abbildung 6.4: Verteilung der rational glatten Werte

der Zahl 980123761807 die 41526530073. Potenz von $3x^2 + 273$ berechnet werden, was 41526530073 Schleifendurchläufe mit je einer Polynommultiplikation und einer Polynomdivision zur Folge hätte. Daher die folgende Idee:

- Berechne die Binärdarstellung von $e = \sum a_i \cdot 2^i$.
- Berechne in einer Schleife für $i = 0, ..., [\log_2 e]$ die Polynome $g(x)^{2^i} \bmod (f(x) \bmod p)$ durch wiederholtes Quadrieren.
- Für jedes i mit $a_i = 1$ multipliziere das mit 1 initialisierte Ergebnispolynom mit $g(x)^{2^i}$.

Dadurch erhält man logarithmische Laufzeit im Exponenten. Diese Idee hat z.B. im Fall $n = 8051$ die Wurzelberechnung ungefähr um den Faktor 10.000 beschleunigt. Für größere Zahlen fällt das natürlich noch viel mehr ins Gewicht,

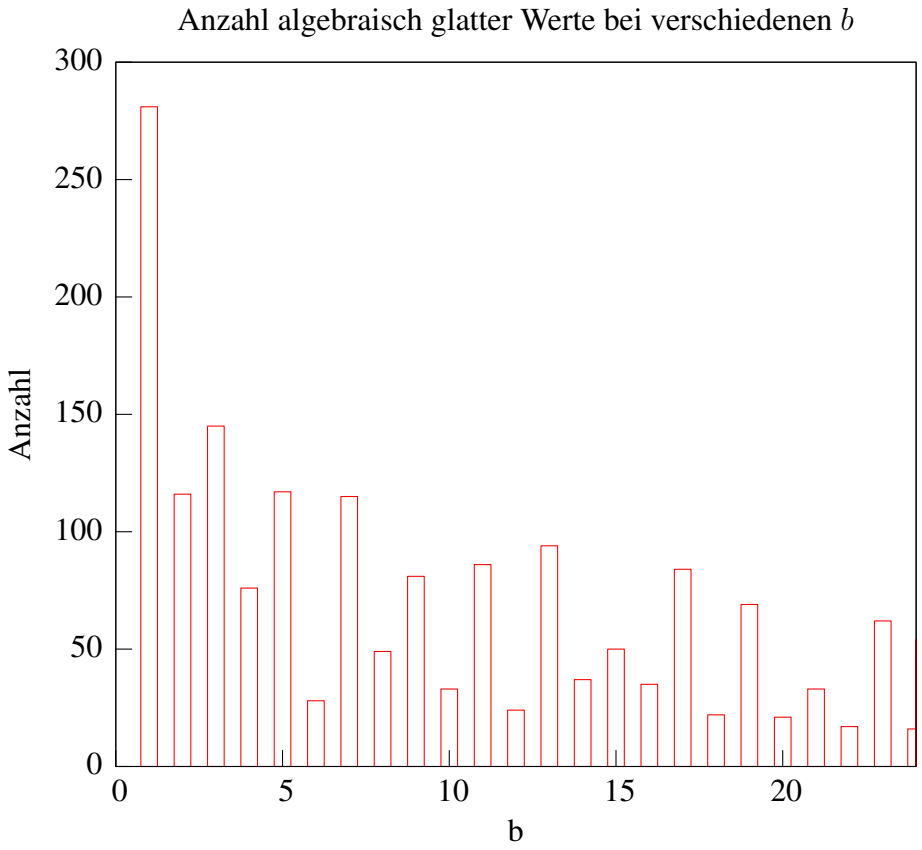

Abbildung 6.5: Verteilung der algebraisch glatten Werte

z.B. im Beispiel oben sind statt 41526530073 Durchläufen nur noch 35 Durchläufe notwendig.

Die Anzahl der Primzahlen, die zur Berechnung der Wurzel benutzt werden müssen, ist im Voraus nicht bekannt. Daher nehme ich einfach schrittweise immer wieder eine weitere hinzu und teste, ob der Chinesische Restsatz einen korrekten Wert für y liefert, also einen, für den $x^2 = y^2 \bmod n$ gilt. Ist dies nicht der Fall, waren es offensichtlich noch nicht genug und es wird eine weitere gesucht.

Man könnte auch die Restsatzberechnungen sparen, indem man eine obere Schranke für die Koeffizientengröße und damit die Anzahl der trägen Primzahlen berechnet. Dies wird in [15] getan, hat aber den Nachteil, dass dann wahrscheinlich zu viele Wurzeln berechnet werden müssen. Für dieses Dilemma konnte ich keine sinnvollere Lösung finden als meine eigene.

Die Methode nach Buhler, Lenstra und Pomerance hat bei mir übrigens selbst

bei sehr kleinen zu faktorisierenden Zahlen nie ein Ergebnis geliefert.[8]

Beim Berechnen der Norm[9] von $f'(x) \bmod p_i$ und dem anschließenden Test auf Irreduzibilität habe ich eine sehr interessante Feststellung gemacht: Immer wenn bei der Berechnung

$$N(f'(x)) = f'(x)^{\frac{p_i^d - 1}{p-1}} \bmod (f(x) \bmod p_i)$$

eine Zahl und kein Polynom berechnet wird, dann ist auch $f(x) \bmod p_i$ irreduzibel. Als dies auch noch nach Tausenden von Tests stimmte, lag die Vermutung nahe, dass hier kein Zufall vorliegt. Tatsächlich ist es aber so, dass diese Vermutung nur im Fall $p_i = 2 \bmod 3$ stimmt, für den Fall $p_i = 1 \bmod 3$ kann man Gegenbeispiele angeben.[10] Wenn aber bei der Normberechnung keine Zahl, sondern ein Polynom entsteht, dann ist f auf jeden Fall nicht irreduzibel. Dadurch konnte der Irredzibilitätstest nochmals verfeinert werden.

6.3 Tests

Die Messungen habe ich größtenteils auf einem AMD Turion 64 MT-30 mit 1GB RAM durchgeführt, bei einigen Diagrammen habe ich aber zusätzlich noch andere Rechner benutzt, z.B. falls die nicht genaue Laufzeit, sondern nur ein optimaler Parameterwert bestimmt werden sollte.

Der CD sind folgende Programme beigelegt:

- *gnfs.cpp* - Meine Implementierung des General Number Field Sieve
- *quadsv.cpp* - Meine Implementierung des Quadratischen Siebs
- *primfak.cpp* - Der triviale Faktorisierungsalgorithmus
- *bit.cpp* - Ein Programm zum Berechnen der Bitlänge einer Zahl

Die ersten beiden Programme können mit „-h“ als Parameter aufgerufen werden, um die verfügbaren anzeigen zu lassen, bei den anderen beiden Programmen ist offensichtlich was sie tun.

Um die Programme kompilieren zu können, müssen die folgenden Bibliotheken verfügbar sein, die ebenfalls auf der CD enthalten sind:

- *GNU-MP* - Die GNU-Multi-Precision-Bibliothek für beliebig lange Ganzzahlen. Dabei muss beachtet werden, dass diese mit C++ Unterstützung erstellt wurde.[11] Ich habe gmp-4.2.1 benutzt.

[8] Das mag an meiner Ungeduld gelegen haben da ich den ersten Test nach nur einer Nacht abgebrochen habe oder daran, dass ich einen Fehler bei der Implementierung gemacht habe.

[9] siehe Seite 58.

[10] Der Beweis ist nach Aussage von Prof. Dr. Hertling „kompliziert und unattraktiv“.

[11] Dazu muss man beim erstellen ./configure –enable-cxx angeben

- *ATLAS* - Die Automatically Tuning Linear Algebra Symbols enthalten die grundlegenden Datenstrukturen für alle Programme. Diese Bibliothek ist hochoptimiert und dauert sehr lange, bis sie gebaut ist. Ich habe atlas-3.6.0 verwendet.
- *LinBox* - Hierin sind zahlreiche Algorithmen aus dem Gebiet der linearen Algebra enthalten. Ich habe zuert linbox-1.0.0 benutzt, bin dann aber auf den neuesten subversion-Zweig umgestiegen, da wir noch einige Fehler bereinigen mussten.[12].

Außerdem sind auf der CD sämtliche Test-Skripte und Messergebnisse, sowie die elektronisch verfügbare Literatur abgespeichert.

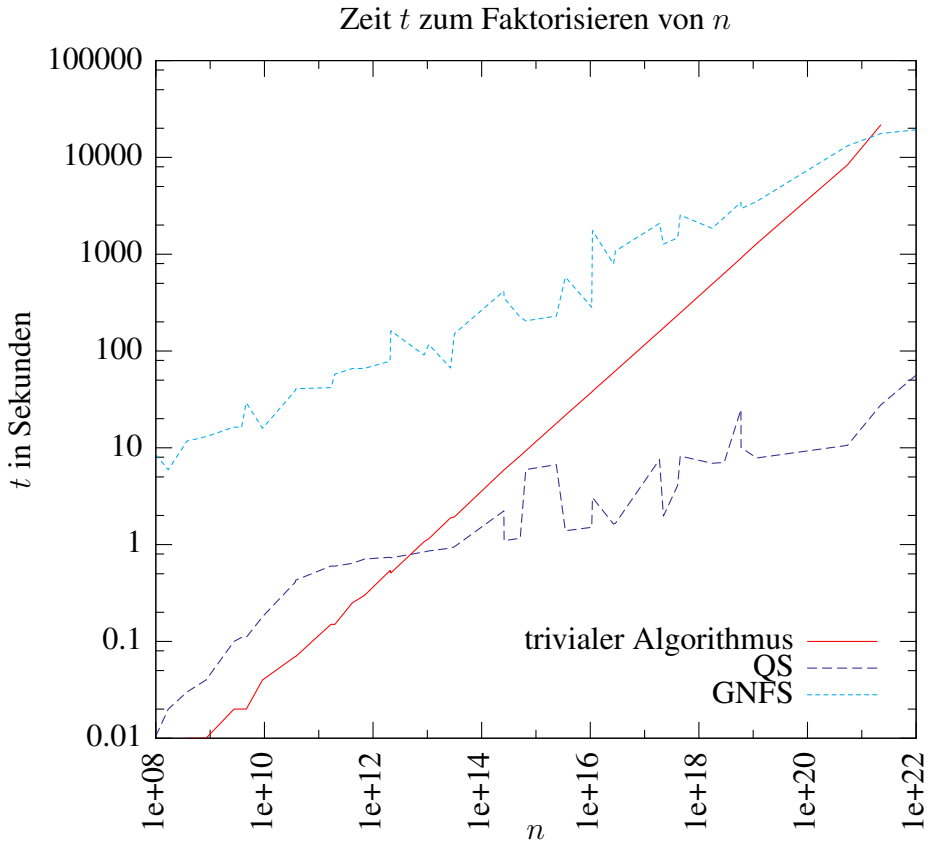

Abbildung 6.6: GNFS, QS und der triviale Algorithmus im Vergleich

Der interessanteste Test ist wahrscheinlich der direkte Vergleich der Gesamtlaufzeit des trivialen Faktorisierungsalgorithmus, der die zu faktorisierende Zahl

[12]Man kann auch die Datei *compose.h* in linbox-1.0.0 mit der Version auf der CD ersetzen

einfach durch $3, 5, 7, 9, 11, 13, 15, ...$ teilt, mit den beiden Siebalgorithmen. Daher stelle ich diesen gleich als erstes vor.

In Abbildung 6.6 sieht man deutlich, dass der triviale Algorithmus mit exponenzieller Laufzeit im Vergleich zum QS bereits bei recht kleinen Zahlen langsamer ist. Das GNFS hingegen hat eine sehr viel höhere Grundlaufzeit. Daher sind größere Zahlen als Eingabe notwendig, bis die Laufzeit zumindest besser ist als die des trivialen Algorithmus.

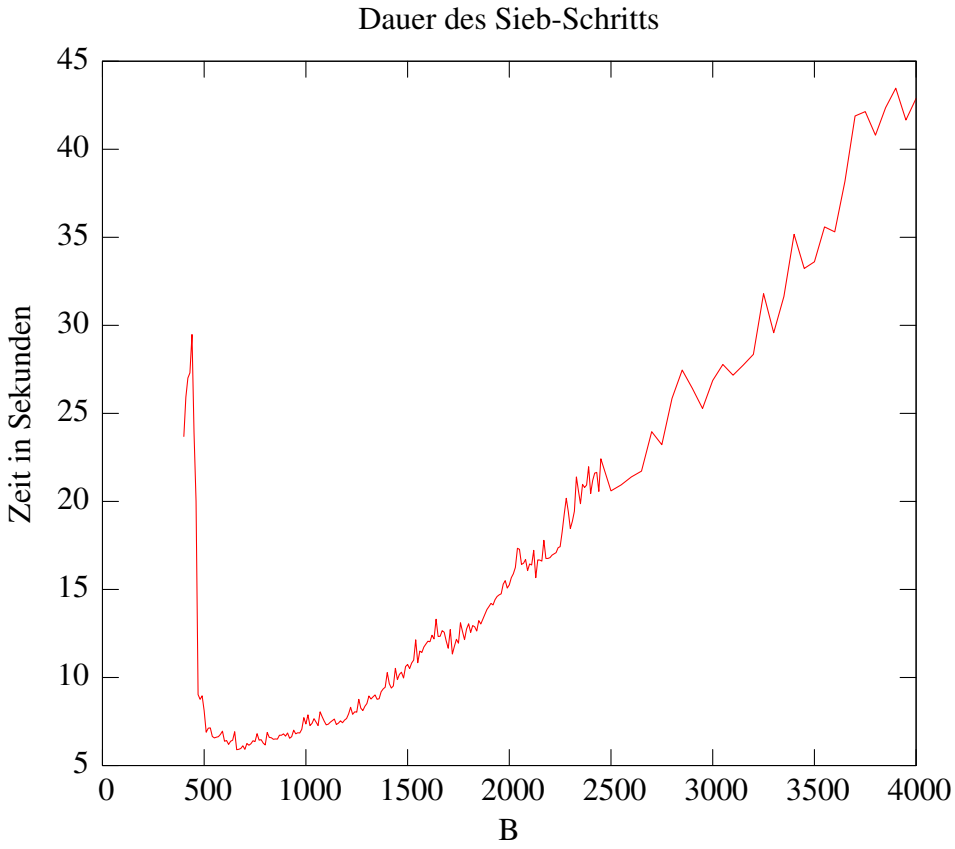

Abbildung 6.7: Dauer des Sieb-Schritts bei 640710979

Eine andere Frage ist, wie man die Algorithmen weiter beschleunigen kann. Zwar kann man beim GNFS recht viele verschiedene Parameter verändern, ich habe mich aber auf den wichtigsten konzentriert: den Parameter B.

Durch Variation dieses Wertes kann man jeden einzelnen Schritt des gesamten Algorithmus beeinflussen. Beispielsweise dauert die Berechnung der Faktorbasen umso länger, je größer der Glätteparameter B ist, da eine Faktorbasis mit mehr Elementen berechnet werden muss. Eine größere Faktorbasis erhöht die Anzahl

der glatten Werte, die bei jedem Siebvorgang gefunden werden. Dafür wird der Matrix-Schritt aufwändiger und auch die Berechnung der algebraischen Wurzel.

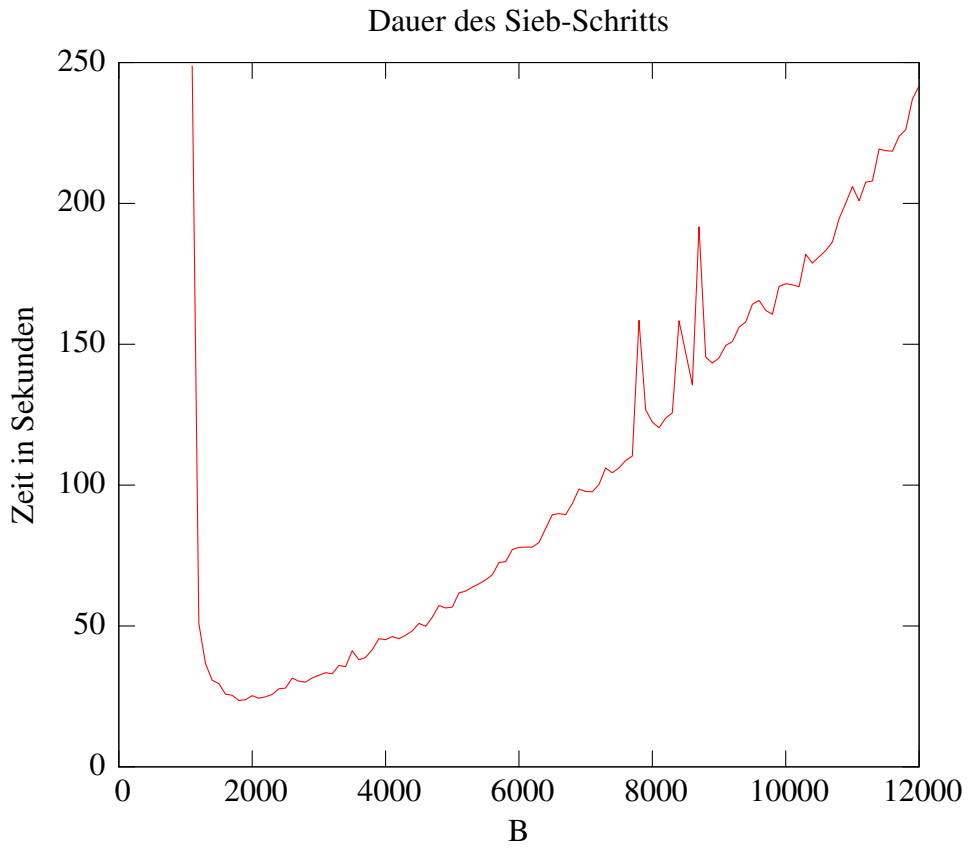

Abbildung 6.8: Dauer des Sieb-Schritts bei 506272798463

In fast allen Fällen ist allerdings der Sieb-Schritt der zeitaufwändigste Teil, daher ist es sinnvoll, B so zu wählen, dass die Dauer dieses Schritts möglichst klein wird. Dazu habe ich verschieden große Zahlen mit unterschiedlichen Werten für B faktorisiert und nur die Laufzeit des Sieb-Schritts betrachtet.

Wie man in den Abbildungen 6.7 bis 6.9 erkennen kann, ist die optimale Wahl von B ein relativ schmaler Grad. Wenn B zu klein ist, dauert der Sieb-Schritt sehr lange, da zu wenig glatte Werte gefunden werden. Ist B hingegen zu groß, so müssen beim Sieben durch zu viele Zahlen geteilt werden.

Interessanterweise ist es für die Laufzeit besonders ungünstig, wenn man den Parameter zu klein wählt. Abbildung 6.9 zeigt insbesondere, dass man ihn lieber etwas zu groß wählen sollte, da die Laufzeit “rechts vom Optimum“ sehr viel langsamer ansteigt als auf der anderen Seite.

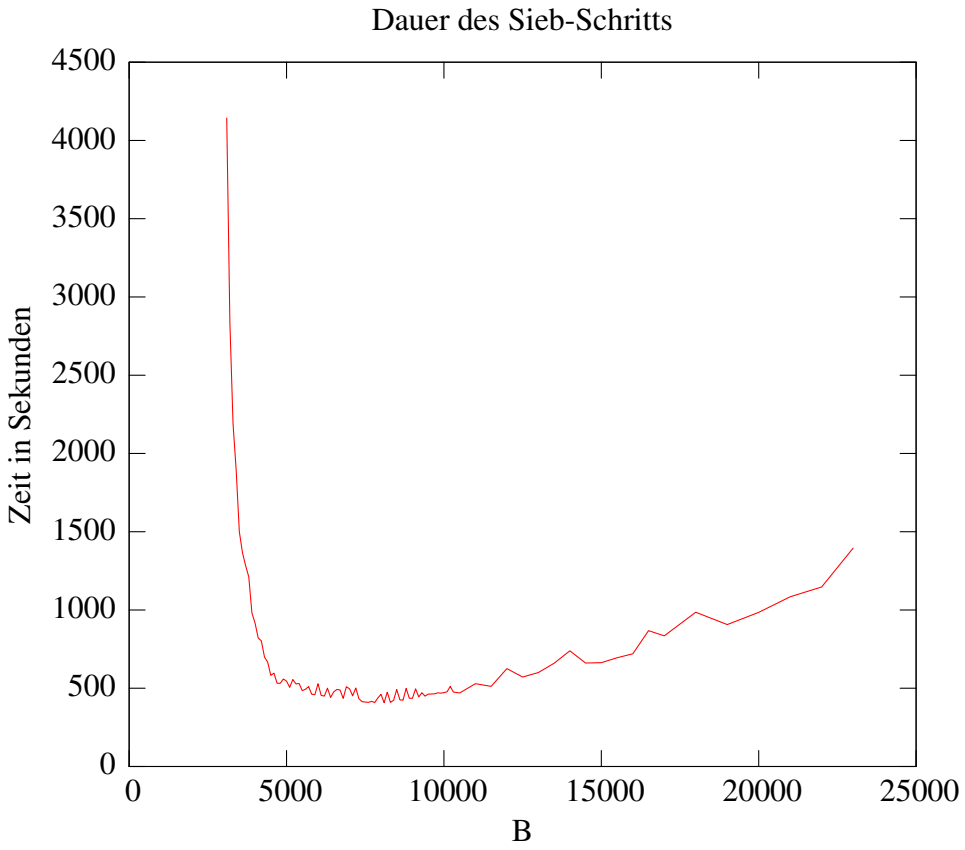

Abbildung 6.9: Dauer des Sieb-Schritts bei 579943766857501

Wählt man B so, wie es die Formel

$$B_0 = e^{\left(\frac{1}{2}\right)\cdot\left(d\cdot\log(d)+\sqrt{(d\cdot\log(d))^2+4\cdot\log(n^{\frac{1}{d}})\cdot\log(\log(n^{\frac{1}{d}}))}\right)}$$

vorschreibt, also ohne den $o(1)$-Term, so bekommt man einen zu kleinen Wert. Bei meinen Tests war es dann meistens so, dass der Wert b beim Sieben auf die Grenze u trifft und das Programm ohne Ergebnis abbricht, da zu wenig glatte Werte gefunden wurden.

Es ist also eine interessante Frage, wie dieser Term in Abhängigkeit von n gewählt werden sollte. Dazu habe ich die optimalen Werte B_{opt} für einige weitere Zahlen experimentell bestimmt und den Faktor berechnet, der zur Formel multipliziert werden muss.

In Abbildung 6.10 sieht man, dass dieser Faktor einigermaßen gleichbleibend um einen Wert zwischen 4 und 6 schwankt. Aus den oben genannten Gründen

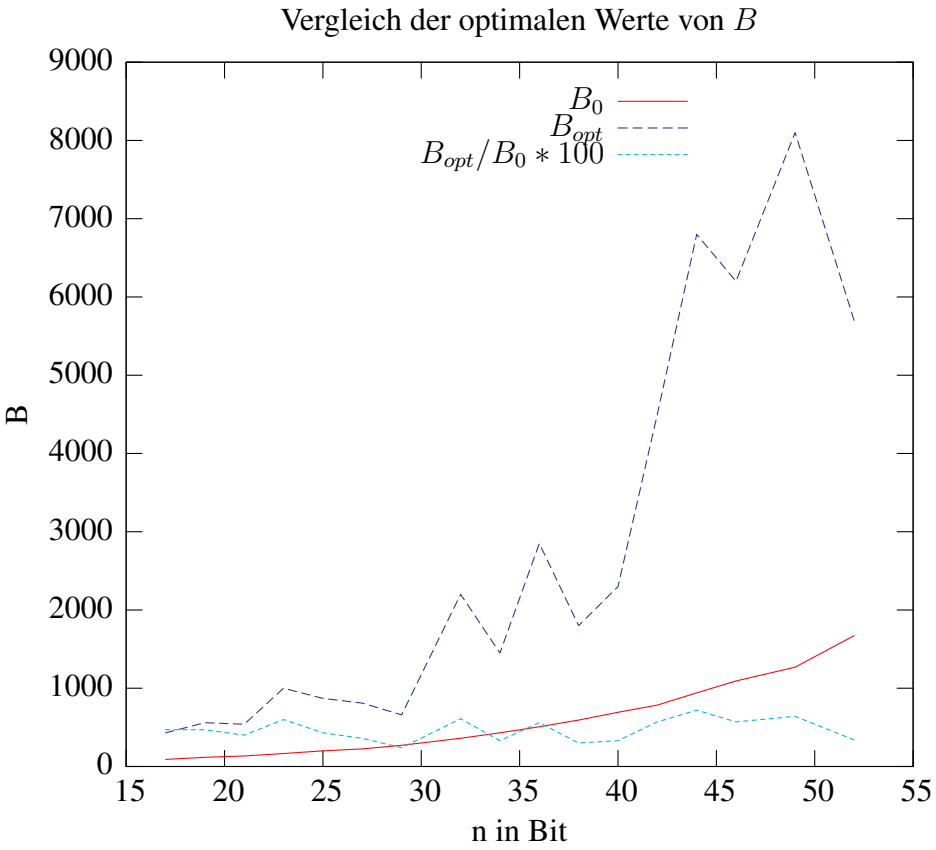

Abbildung 6.10: Optimale B-Werte im Vergleich

wird in meinen Programmen ein Faktor von 6 benutzt, sodass B eher zu groß als zu klein gewählt wird.

Dieser Faktor ist stark von der Implementierung abhängig, da, z.B. im Fall des zusätzlichen Verarbeitens von large primes, die Zahl der gefundenen glatten Werte je Siebabschnitt stark erhöht werden kann. Dadurch kann man mit kleineren Faktorbasen arbeiten und somit den Wert B kleiner setzen. Es ist außerdem unklar, ob der hier berechnete Faktor bei größeren Zahlen irgendwann vernachlässigt werden kann. Theoretisch sollte er jedenfalls gegen den Wert 1 konvergieren.

Kapitel 7

Abschließende Bemerkungen

Wie man im vorigen Kapitel gesehen hat, existiert noch einiges an Potenzial zum Verbessern des GNFS. Interessanterweise bewirkt eine unterschiedliche Wahl der Parameter unter Umständen eine enorme Veränderung der Laufzeit. Desweiteren ist der Parameter B nur einer von vielen Dingen, die man im Algorithmus variieren kann.

Ich konnte jedenfalls nachweisen, dass der Algorithmus vom Prinzip her funktioniert sowie einige Tests damit machen. Damit wäre das eigentliche Ziel der Arbeit bereits erreicht.

Es ist schade, dass der Punkt, ab dem das GNFS besser ist als das QS, schlecht zu zeigen ist, da dieser bei mehr als 110 Bit liegt. Dazu wäre eine deutlich längere experimentelle Phase notwendig gewesen.

Insgesamt betrachtet ist das Thema ein sehr mathematisches, das viele Elemente enthält, die man anfangs gar nicht erwartet hätte. Es fließen Erkenntnisse vom Jahr 1963 bis 2007 ein[1], und es wird mit Sicherheit noch weitere Entwicklungen auf diesem Gebiet geben, die zur Verbesserung beitragen können, z.B. die Untersuchung der Kettenreaktion, die in Kapitel 5.6.5 beschrieben ist.

[1] Die Verfeinerung des Irreduzibilitätstests auf Seite 78 ist vom Januar 2007.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Arjen Lenstra, *Integer Factoring*, in Designs, Codes and Cryptography Vol. 19, Kluwer Academic Publishers, 2000, http://modular.fas.harvard.edu/edu/Fall2001/124/misc/arjen_lenstra_factoring.pdf

[2] Prof. Dr. Johannes Buchmann, Volker Müller, *Algorithms for factoring Integers*, Technische Universität Darmstadt, 2005, http://www.cdc.informatik.tu-darmstadt.de/~buchmann/Lecture%20Notes/Algorithms%20for%20factoring%20integers.pdf

[3] Ian Stewart, David Tall, *Algebraic Number Theory*, 2nd edition, Chapman and Hall, 1987, ISBN 0-412-29807-8

[4] Prof. Dr. Ernst Binz, *Lineare Algebra 2*, Vorlesungsmitschrift, Universität Mannheim, SS 2003

[5] Prof. Dr. Matthias Krause, *Kryptographie 1*, Vorlesungsmitschrift, Universität Mannheim, SS 2004

[6] Prof. Dr. Johannes Buchmann, *Algebra für Informatiker*, Skript, Technische Universität Darmstadt, 2005, http://www.cdc.informatik.tu-darmstadt.de/~buchmann/Lecture%20Notes/Algebra%20f%FCr%20Informatiker.pdf

[7] Prof. Wolfgang K. Seiler, *Computeralgebra - Faktorisierung von Polynomen*, Universität Mannheim, WS 2005/06, http://hilbert.math.uni-mannheim.de/CAlg/CASkript05-3.pdf

[8] Edwin Weiss, *Algebraic Number Theory*, McGraw-Hill, 1963, ISBN 0-486-40189-8

[9] Carl Pomerance, *The Quadratic Sieve Factoring Algorithm*, in Advances in Cryptology: Proceedings of EUROCRYPT 84, Springer-Verlag, 1985, http://www.math.dartmouth.edu/~carlp/PDF/paper52.pdf

[10] Carl Pomerance, *Smooth Numbers and the Quadratic Sieve*, MRSI, 2000, http://www.math.leidenuniv.nl/~reinier/ant/sieving.pdf

[11] A.K. Lenstra, H.W. Lenstra, Jr., *The Development of the Number Field Sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, 1993, enthält [12] bis [17], ISBN 3-540-57013-6

[12] J.M. Pollard, *Factoring with cubic integers*, in The Development of the Number Field Sieve, Springer-Verlag, 1993

[13] A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J.M. Pollard, *The number field sieve*, in The Development of the Number Field Sieve, Springer-Verlag, 1993, http://www.std.org/msm/common/nfspaper.pdf

[14] J.M. Pollard, *The lattice sieve*, in The Development of the Number Field Sieve, Springer-Verlag, 1993

[15] J.P. Buhler, H.W. Lenstra, Jr., Carl Pomerance, *Factoring Integers with the Number Field Sieve*, in The Development of the Number Field Sieve, Springer-Verlag, 1993, http://openaccess.leidenuniv.nl/dspace/bitstream/1887/2149/1/346_114.pdf

[16] Jean-Marc Couveignes, *Computing a square root for the number field sieve*, in The Development of the Number Field Sieve, Springer-Verlag, 1993, http://w3.univ-tlse2.fr/grimm/algo/publi/Cou94-2.pdf

[17] *A general number field sieve implementation*, in The Development of the Number Field Sieve - Daniel J. Bernstein, A.K. Lenstra - Springer-Verlag - 1993

[18] Leonard M. Adleman, *Factoring numbers using singular integers*, ACM Press, 1991, http://portal.acm.org/citation.cfm?id=103432

[19] Peter L. Montgomery, *Square roots of products of algebraic numbers*, in Proceedings of Symposia in Applied Mathematics, Mathematics of Computation 1943-1993, American Mathematical Society, 1993, ftp://ftp.cwi.nl/pub/pmontgom/sqrt.ps.gz

[20] R.D. Silverman, *The multiple polynomial quadratic sieve*, in Proceedings of Symposia in Applied Mathematics, Mathematics of Computation 1943-1993, American Mathematical Society, 1993, ISBN 0-8218-0291-7

[21] Thorsten Kleinjung, *On Polynomial Selection for the General Number Field Sieve*, in Mathematics of Computation, Vol. 75, American Mathematical Society, 2006, http://www.ams.org/mcom/2006-75-256/S0025-5718-06-01870-9/S0025-5718-06-01870-9.pdf

[22] Brian Anthony Murphy, *Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm*, Australian National University, 1999, http://cr.yp.to/bib/1999/murphy.ps

[23] Jörg Zayer, *Faktorisieren mit dem Number Field Sieve*, Dissertation, Universität des Saarlandes, 1995, http://www.informatik.tu-darmstadt.de/ftp/pub/TI/reports/zayer.diss.ps.gz

[24] Matthew E. Briggs, *An Introduction to the GNFS*, Master's Thesis, State University of Virginia, 1998, http://www.math.uoc.gr/~marios/papers/etd.pdf

[25] Peter L. Montgomery, *A Block Lanczos Algorithm for finding Dependencies over GF(2)*, in Lecture Notes in Computer Science, Vol. 921, Springer-Verlag, 1995, http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/E95/106.PDF

[26] Bradford Hovinen, *Blocked Lanczos-style Algorithms over Small Finite Fields*, Master's thesis, University of Waterloo, 2004, http://www.math.toronto.edu/hovinen/thesis.pdf

[27] Erich Kaltofen, *Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems*, in Proceedings of the 10th International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Springer-Verlag, 1993, http://www4.ncsu.edu/~kaltofen/bibliography/95/Ka95_mathcomp.pdf

[28] Olaf Groß, *Der Block Lanczos Algorithmus über GF(2)*, Diplomarbeit, Universität des Saarlandes, 1994, http://www.cdc.informatik.tu-darmstadt.de/reports/reports/gross.diplom.ps.gz

[29] Daniel M. Gordon, *Discrete Logarithms in GF(p) using the Number Field Sieve*, University of Georgia, 1992, http://www.ccrwest.org/gordon/log.pdf

[30] Adi Shamir, Eran Tromer, *Factoring Large Numbers with the TWIRL Device*, Weizmann Institute of Science, Israel, 2003, http://www.wisdom.weizmann.ac.il/%7Etromer/papers/twirl.pdf

Index

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und dass ich keine weitere Literatur benutzt habe als die am Ende aufgeführte, auf die an den entsprechenden Stellen im Text verwiesen wird.

Mannheim, den 11. Januar 2007 Christian Stöffler