ECM

Eine FFT-Continuation für die elliptische Kurvenmethode

Diplomarbeit

von

Andreas Müller

nach einem Thema von Prof. Dr. Buchmann,
am Fachbereich 14 – Informatik –
der Universität des Saarlandes

März 1995

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken,

Mein Dank gebührt Prof. Dr. Buchmann für die Vergabe dieses interessanten Themas.

Desweiteren danke ich meinem Betreuer Franz-Dieter Berger, der mir bei der Erarbeitung der Theorie und anschließender Implementierung mit Rat und Tat zur Seite stand.

Victor Shoup stellte uns ein sehr effizientes Programmpaket zum Rechnen mit Polynomen zur Verfügung. Außerdem gab er sehr viele gute Hinweise zur Implementierung der für ECM benötigten Veränderungen.

Weiterhin möchte ich Peter L. Montgomery danken, der mir bei einem Besuch in Saarbrücken und per Email viele wertvolle Hinweise zur Theorie und Praxis des Algorithmus gegeben hat.

Heinz-Bernd Eggenstein danke ich für schlaue Antworten auf knifflige Fragen, für viele gute Ideen und interessante Diskussionen per Email.

Für viele Diskussionen und die gute Zusammenarbeit am Lehrstuhl danke ich meinen Freunden Thomas F. Denny, Frank Lehmann, Markus Maurer, Thomas Setz, Thomas Sosnowski und last but not least meinem Bruder Volker Müller.

Mein besonderer Dank gilt meinen Eltern, die mein Studium ermöglichten.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Notation

$\mathbb{N}$	Menge der natürlichen Zahlen ohne 0
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{P}$	Menge der Primzahlen
$\mathbb{Z}/m\mathbb{Z}$	Menge der Restklassen von $\mathbb{Z}$ modulo m
$(\mathbb{Z}/m\mathbb{Z})^*$	$:= \{a \in (\mathbb{Z}/m\mathbb{Z}), \mathrm{ggT}(a,m) = 1\}$
$\mathrm{ord}_G(a)$	Ordnung des Elements a in der Gruppe G (siehe Definition A.3)
$\lvert G\rvert$	Ordnung der Gruppe G (siehe Definition A.2)
$\longleftrightarrow$	Bijektive Zuordnung
$\log_2(b)$	Logarithmus von b zur Basis 2
$\varphi(x)$	Eulersche φ-Funktion (siehe Definition A.7)
$\mathrm{ggT}(a,b)$	größter gemeinsamer Teiler von a und b
$a \mid b$	a teilt b
$a \nmid b$	a teilt nicht b
$x[Q]$	x-Koordinate des Punktes Q auf einer elliptischen Kurve
$\mathrm{grad}(F)$	Grad des Polynoms $F(x)$

Einleitung

In dieser Arbeit beschäftigen wir uns mit dem Problem der Faktorisierung „großer“ Zahlen. Unter der Faktorisierung einer natürlichen Zahl größer als 1 verstehen wir die Zerlegung der Zahl in ein Produkt von Primfaktoren. Diese Zerlegung ist nach dem Fundamentalsatz der Algebra bis auf die Reihenfolge der Faktoren eindeutig bestimmt.
Dabei ist folgende Tatsache von großem Interesse: Es ist relativ leicht zu testen, ob eine Zahl eine Primzahl ist oder ob sie zusammengesetzt ist. Aus diesem Umstand können wir aber gewöhnlich nicht die Faktoren bestimmen. Die Faktorisierung von Zahlen ist deutlich schwerer als ein Test auf Primzahl.

Ursprünglich beschäftigten sich nur Zahlentheoretiker mit dem Problem der Faktorisierung großer Zahlen. Heutzutage gibt es Public Key Verschlüsselungsverfahren wie z.B. RSA, deren Sicherheit auf der Schwierigkeit beruht, Zahlen zu faktorisieren. Dadurch ist das Interesse an der Entwicklung neuer Algorithmen und Verbesserung bekannter Algorithmen gewachsen.

Zur Lösung des Problems, Zahlen zu faktorisieren, existieren verschiedene Algorithmen. Diese kann man grob in zwei Gruppen einteilen:

- Allgemeine Faktorisierungsverfahren, wie z.B.
  - Quadratisches Sieb
  - Number Field Sieve
- Spezielle Faktorisierungverfahren, wie z.B.
  - p-1 - , p+1 - Methode
  - Pollards - ρ - Methode
  - ECM — Elliptische Kurvenmethode

Die Laufzeit der allgemeinen Faktorisierungsverfahren ist im wesentlichen von der Größe der zu faktorisierenden Zahl abhängig. Bei den hier erwähnten speziellen Faktorisierungsverfahren wird die Laufzeit dagegen in hohem Maße durch die Dezimallänge der Primfaktoren bestimmt, insbesondere durch die Dezimallänge des zweitgrößten Faktors. Im allgemeinen findet man kleinere Faktoren schneller als größere, so daß man nach Bestimmung des zweitgrößten Faktors mit hoher Wahrscheinlichkeit auch die kleineren Faktoren bestimmt hat. Damit kann man durch

Division der Zahl durch alle bekannten Faktoren die Komplettfaktorisierung der Zahl berechnen. Dabei fällt der größte Faktor als Rest bei der Division ab.

Wir beschäftigen uns in dieser Arbeit mit einem speziellen Faktorisierungsalgorithmus, mit der Elliptischen Kurvenmethode (ECM). Der Grundalgorithmus wurde 1987 von H. W. Lenstra Jr. entwickelt. In der Folgezeit wurden verschiedene Verbesserungen dieses Grundalgorithmus formuliert. 1992 hat P.L. Montgomery eine neue Continuation, die sogenannte „FFT - Continuation" beschrieben.

Die Laufzeit von ECM ist in hohem Maße abhängig von der Größe der Faktoren, die man finden will. Da diese Größe von vorneherein nicht bekannt ist, kann man versuchen, z.B. 20-stellige Faktoren zu finden. Nach Ausführung der Berechnungen kann man mit hoher Wahrscheinlichkeit sagen, daß alle Faktoren mit weniger als 20 Dezimalstellen gefunden worden sind. Die Größe der Faktoren, die man finden will, ist abhängig von der Rechenzeit, die man zu investieren bereit ist.
Mit Hilfe der im Zuge dieser Arbeit implementierten Programme können wir 33-stellige Faktoren mit einem Aufwand von ungefähr 6 MIPS-Jahren finden (1 MIPS-Jahr ist die Rechenleistung eines Rechners mit 1 MIPS innerhalb eines Jahres). Kleinere Faktoren finden wir in deutlich kürzerer Zeit.

Im Rahmen dieser Arbeit stellen wir nun den Grundalgorithmus, die Verbesserungen und die FFT-Continuation vor. Dabei ist es gut, wenn der Leser gewisse Grundkenntnisse im Bereich der Faktorisierung, speziell von ECM hat. Eine ausführliche Beschreibung der Ideen und mathematischen Grundlagen des Grundalgorithmus findet man in [Be93]. Dadurch gliedert sich die Arbeit wie folgt:

Kapitel 1: Zu Beginn stellen wir die benötigten Grundlagen zu elliptischen Kurven vor. Damit können wir den Grundalgorithmus formulieren. Danach beschreiben wir eine erste Verbesserung des Grundalgorithmus durch Anfügen einer sogenannten „Continuation".

Kapitel 2: Die Laufzeit von ECM wird durch die Komplexität der Operationen auf elliptischen Kurven mitbestimmt. Wir führen eine neue Parametrisierung für elliptische Kurven ein und stellen Vor- und Nachteile dieser Parametrisierung vor. Außerdem beschreiben wir die Wahl „guter" elliptischer Kurven und Punkte auf diesen Kurven für ECM.

Kapitel 3: In der FFT-Continuation benötigen wir effiziente Algorithmen zum Rechnen mit Polynomen. Dabei gehen wir ausführlich auf die Multiplikation von Polynomen mit Hilfe der Schnellen Fourier-Transformation (FFT) ein. Durch Verwendung der FFT-Multiplikation ist erst ein effizientes Ausführen der FFT-Continuation möglich, wie es der Name auch schon sagt.

Kapitel 4: Nun haben wir die Hilfsmittel zur Ausführung der FFT-Continuation vorgestellt. Wir beschreiben nun einen einfachen Ansatz und die effiziente Ausführung dieser neuen Continuation.

Kapitel 5: In diesem Abschnitt beschreiben wir unsere Implementierung. Dabei stellen wir die verwendeten Datenstrukturen vor und geben Einblick in die mögliche und ausgeführte Parallelisierung von ECM. Die Installation und Bedienung der entstandenen Programme (sequentielle und parallele Version) wird beschrieben.

Kapitel 6: Wir haben zur Lösung von Teilproblemen verschiedene Algorithmen implementiert und verglichen. Durch die erreichten Laufzeiten begründen wir die von uns getroffene Auswahl. Zusätzlich geben wir die daraus resultierenden Laufzeiten an.

Kapitel 7: Mit den implementierten Programmen haben wir erfolgreich an zwei Wettbewerben teilgenommen, die sich mit der Faktorisierung vorgegebener langer Zahlen beschäftigen. Diese Wettbewerbe und die erreichten Ergebnisse stellen wir vor.

Kapitel 1

ECM

1.1 Grundlagen

In diesem Kapitel wollen wir die grundlegenden Begriffe und Definitionen kennenlernen. Im folgenden betrachten wir zuerst elliptische Kurven über dem Körper $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$ für eine Primzahl $p > 3$.

1.1. Definition
Eine **elliptische Kurve** *E über dem Körper $\mathbb{K}$ ist ein Paar $(a, b) \in \mathbb{K}^2$, wobei $4a^3 + 27b^2 \neq 0$ ist.*

Die **Menge aller ($\mathbb{K}$-rationalen) Punkte** *$E(\mathbb{K})$ auf E ist*

$$E(\mathbb{K}) \quad = \quad \left\{ (x : y : z) \in \mathbb{P}^2 \;\middle|\; y^2 z = x^3 + a \cdot xz^2 + bz^3 \right\}. \tag{1.1}$$

Unter $\mathbb{P}^2$ verstehen wir dabei die projektive Ebene über $\mathbb{K}$. Diese enthält die Äquivalenzklassen der Tripel $(x, y, z) \in \mathbb{K}^3, (x, y, z) \neq (0, 0, 0)$, die durch folgende Äquivalenzrelation bestimmt werden:

$$(x, y, z) \sim (x', y', z') \quad \iff \quad \exists\, c \in \mathbb{K}^* \,:\, cx = x', cy = y', cz = z'.$$

Als **Punkt** *bezeichnen wir ein Tripel $(x : y : z) \in \mathbb{P}^2$, welches die Gleichung aus (1.1), nämlich $y^2 z = x^3 + a \cdot xz^2 + bz^3$ erfüllt.*
In $E(\mathbb{K})$ existiert genau ein Punkt $(x : y : z)$ mit $z = 0$. Diesen bezeichnen wir als **Nullpunkt** *oder $\mathcal{O}$. Alle Punkte mit z-Koordinate ungleich 0 nennen wir* **endliche Punkte**.

Aus der Definition 1.1 erkennt man, daß in der Äquivalenzklasse jedes endlichen Punktes $(x : y : z) \in \mathbb{P}^2$ genau ein Tripel $(\bar{x}, \bar{y}, \bar{z}) \in \mathbb{K}^3$ mit $\bar{z} = 1$ existiert, nämlich $(xz^{-1}, yz^{-1}, 1)$. Wir identifizieren dann jeden endlichen Punkt $(x : y : z)$ aus der projektiven Ebene mit dem Punkt (xz^{-1}, yz^{-1}) in der affinen Ebene. Den Nullpunkt identifizieren wir mit (∞, ∞), hier kann man keine Invertierung der z-Koordinate durchführen.

1.2. Bemerkung
Aus den obigen Betrachtungen können wir eine kürzere Form der elliptischen Kurvengleichung einführen. Die Gleichung

$$y^2 \quad = \quad x^3 \, + \, a \cdot x \, + \, b \tag{1.2}$$

heißt Kurvengleichung von $E = E_{a,b}$ in **kurzer Weierstraß-Normalform**. Die Menge der Punkte ist dann

$$E(\mathbb{K}) \quad = \quad \left\{ (x, y) \in \mathbb{K} \times \mathbb{K} \;\mid\; y^2 = x^3 + a \cdot x + b \right\} \;\cup\; \{\mathcal{O}\}\,. \tag{1.3}$$

Diese können wir mittels einer Bijektion in eine allgemeinere Form überführen, in der auch elliptische Kurven für Körper der Charakteristik 2 bzw. 3 definiert sind (siehe [Si85]). Für die Berechnungen auf elliptischen Kurven in ECM genügt die kurze Darstellung, da wir o.B.d.A. annehmen können, daß die zu faktorisierende Zahl N weder von 2 noch von 3 geteilt wird.

Im folgenden betrachten wir nur noch elliptische Kurven und Punkte auf diesen Kurven in dieser kurzen Weierstraß-Normalform. Diese ist auch in der Implementierung auf dem Rechner umgesetzt.

In der Abbildung 1.1 auf der nächsten Seite sehen wir eine elliptische Kurve über $\mathbb{R}$ in der affinen Ebene.

Abbildung 1.1: Elliptische Kurve (Addition P + Q)

1.3. Satz
Die Menge $E(\mathbb{K})$ aus (1.3) bildet eine abelsche, im allgemeinen additiv geschriebene Gruppe mit neutralem Element $\mathcal{O}$ (siehe [Si85]). Die Addition wird dabei folgendermaßen gegeben:

- *$\mathcal{O}$ ist das Nullelement in $E(\mathbb{K})$.*
- *Für einen Punkt $P = (x, y) \in E(\mathbb{K}),\ P \neq \mathcal{O}$ ist der negative Punkt*

$$-P \quad = \quad (x, -y) \ \in \ E(\mathbb{K}).$$

- *Sind $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ endliche Punkte aus $E(\mathbb{K})$ mit $Q \neq -P$, so wird die Summe $P + Q = (x_3, y_3)$ durch folgende Formeln gegeben:*

$$\begin{aligned} x_3 &= -x_1 - x_2 + \lambda^2 \\ y_3 &= -y_1 + \lambda \cdot (x_1 - x_3) \end{aligned}$$

wobei

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2}, & \text{falls } P \neq Q \\[2ex] \dfrac{3x_1^2 + a}{2y_1}, & \text{falls } P = Q. \end{cases} \tag{1.4}$$

1.4. Beispiel
Seien $\mathbb{K} = \mathbb{Z}/13\mathbb{Z}, E = E_{4,5}$, und $P = (1, 6)$, wobei wir z.B. die Zahl 1 mit der Restklasse $[1]_{13}$ identifizieren.

Zur Berechnung der Summe $3 \cdot P$ verwenden wir die oben angegebenen Formeln. Zuerst verdoppeln wir den Punkt P und führen danach die Addition $2 \cdot P + P$ aus.

$$\begin{aligned} \lambda &\equiv (3 \cdot 1^2 + 4) \cdot (2 \cdot 6)^{-1} \equiv 7 \cdot 12^{-1} \equiv 6 \mod 13 \\ \Rightarrow x_3 &\equiv -1 - 1 + 6^2 \equiv 34 \equiv 8 \mod 13 \\ \Rightarrow y_3 &\equiv -6 + 6 \cdot (1 - 8) \equiv -48 \equiv 4 \mod 13 \end{aligned}$$

$$\Longrightarrow 2 \cdot P = (8, 4)$$

Damit erhalten wir für die anschließende Addition:

$$\begin{aligned} \lambda &\equiv (4 - 6) \cdot (8 - 1)^{-1} \equiv -2 \cdot 7^{-1} \equiv 9 \mod 13 \\ \Rightarrow x_3 &\equiv -8 - 1 + 9^2 \equiv 72 \equiv 7 \mod 13 \\ \Rightarrow y_3 &\equiv -4 + 9 \cdot (8 - 7) \equiv -4 + 9 \equiv 5 \mod 13 \end{aligned}$$

$$\Longrightarrow 3 \cdot P = (7, 5)$$

1.5. Bemerkung
Die Definition der elliptischen Kurven und die Additionsformeln über dem Körper $\mathbb{Z}/p\mathbb{Z}$ können wir analog auf elliptische Kurven über $\mathbb{R}$ übertragen. Abbildung 1.1 zeigt die Darstellung der elliptischen Kurve $E_{a,b}$ und einer Punktaddition über $\mathbb{R}$. Dabei erkennen wir die geometrische Bedeutung einer Addition.
Um zwei verschiedene, endliche Punkte zu addieren, legen wir eine Gerade durch die beiden Punkte. Diese schneidet die Kurve in einem weiteren Punkt. Das Ergebnis der Addition erhalten wir, indem wir diesen Schnittpunkt an der x-Achse spiegeln. Die Verdopplung eines Punktes P verläuft analog, hier legen wir die Tangente im Punkt P an die Kurve, es gibt wiederum einen Schnittpunkt, der gespiegelt werden muß.
Daraus ergibt sich auch die Bedeutung der Variablen λ aus (1.4), sie stellt die Steigung der Sekante bzw. Tangente dar.

Wir haben nun elliptische Kurven in Weierstraß-Form über dem Körper $\mathbb{Z}/p\mathbb{Z}$ für eine Primzahl $p > 3$ eingeführt. Im ECM-Algorithmus betrachten wir aber elliptische Kurven über dem Ring $\mathbb{Z}/N\mathbb{Z}$. N ist dabei eine zusammengesetzte Zahl

$$N \quad = \quad \prod_{i=1}^{r} p_i \qquad p_i \in \mathbb{P}_{>3}, \qquad \forall\, i = 1, \ldots, r. \tag{1.5}$$

Wir beschränken uns hier auf den Fall, daß alle Primzahlen p_i mit Vielfachheit 1 in der Zahl N als Teiler vorkommen. Durch Probedivision können wir alle Teiler von N finden, die kleiner als 10^5 sind. Es ist sehr unwahrscheinlich, daß dann noch Teiler mit Vielfachheit größer als 1 in N vorkommen.
Wir wollen nun die Definition von elliptischen Kurven in Weierstraß-Darstellung und die Regeln zum Rechnen in $E(\mathbb{Z}/p\mathbb{Z})$ auf elliptische Kurven über dem Ring $\mathbb{Z}/N\mathbb{Z}$ übertragen.

Ein Paar $(a, b) \in (\mathbb{Z}/N\mathbb{Z})^2$ mit $4a^3 + 27b^2 \in (\mathbb{Z}/N\mathbb{Z})^*$ heißt elliptische Kurve über dem Ring $\mathbb{Z}/N\mathbb{Z}$. Dabei genügt ein Punkt $P = (x, y) \in (\mathbb{Z}/N\mathbb{Z})^2$ auf dieser elliptischen Kurve der Gleichung

$$y^2 \quad = \quad x^3 \,+\, a \cdot x \,+\, b.$$

Um die Addition zweier Punkte zu definieren, müssen wir mit den allgemeineren projektiven Koordinaten arbeiten.

Wir definieren zuerst den Homomorphismus ϕ_{p_i} :

$$\begin{array}{llcl} \phi_{p_i}: & E\left(\mathbb{Z}/_{N\mathbb{Z}}\right) & \longrightarrow & E\left(\mathbb{Z}/_{p_i\mathbb{Z}}\right) \\ & \left(x : y : z\right) & \longmapsto & \Big((x \bmod p_i) : (y \bmod p_i) : (z \bmod p_i)\Big), \end{array}$$

der die Koordinaten eines Punktes modulo des Teilers p_i von N reduziert. Damit erhalten wir folgende Abbildung, die sogar eine Bijektion ist (wie man leicht zeigen kann).

$$\begin{array}{ccc} E\left(\mathbb{Z}/_{N\mathbb{Z}}\right) & \cong & E\left(\mathbb{Z}/_{p_1\mathbb{Z}}\right) \times E\left(\mathbb{Z}/_{p_2\mathbb{Z}}\right) \times \ldots \times E\left(\mathbb{Z}/_{p_r\mathbb{Z}}\right) \\ P & \longleftrightarrow & \big(\ \phi_{p_1}(P)\ ,\ \phi_{p_2}(P), \ldots, \phi_{p_r}(P)\ \big). \end{array}$$

Die Addition in $E(\mathbb{Z}/p_1\mathbb{Z}) \times E(\mathbb{Z}/p_2\mathbb{Z}) \times \ldots \times E(\mathbb{Z}/p_r\mathbb{Z})$ ist dabei komponentenweise definiert. Wir erhalten dann:

$$\begin{array}{ccc} P+Q & \longleftrightarrow & \big(\phi_{p_1}(P), \ldots \phi_{p_r}(P)\big) + \big(\phi_{p_1}(Q), \ldots, \phi_{p_r}(Q)\big) \\ & \longleftrightarrow & \Big(\big(\phi_{p_1}(P) + \phi_{p_1}(Q)\big), \cdots, \big(\phi_{p_r}(P) + \phi_{p_r}(Q)\big)\Big). \end{array}$$

Nun betrachten wir eine Punktaddition $P+Q$ in $E(\mathbb{Z}/N\mathbb{Z})$. Dabei verwenden wir die über Körpern definierten Additionsformeln aus Satz 1.3. Wir können zwei Fälle unterscheiden:

- Die Invertierung, die bei der Punktaddition auftritt, kann korrekt ausgeführt werden. Das Ergebnis der Punktaddition ist ein endlicher Punkt in $E(\mathbb{Z}/N\mathbb{Z})$. Durch Anwenden der obigen Bijektion sieht man, daß dieser Punkt dem korrekten Ergebnis modulo der Primteiler p_i von N entspricht. Mit Hilfe des Chinesischen Restsatzes folgt also, daß das Ergebnis auch modulo N korrekt ist.
- Die Invertierung von $(2 \cdot y[P])$ bzw. $(x[P] - x[Q])$ schlägt in $\mathbb{Z}/N\mathbb{Z}$ fehl. Wir finden einen Faktor p von N. Sei o.B.d.A. $p_s := p$ für ein $1 \leq s \leq r$ der gefundene Primteiler von N. Das Ergebnis der Punktaddition in $E(\mathbb{Z}/p_s\mathbb{Z})$ ist der Nullpunkt $\mathcal{O}_{p_s}$. Aus dem Chinesischen Restsatz folgt, daß die z-Koordinate der Summe in $E(\mathbb{Z}/N\mathbb{Z})$ ein Vielfaches von p_s ist. Wir können die Summe nicht in die affine Ebene transformieren. Dadurch tritt bei der Berechnung in der affinen Ebene ein Fehler auf.

Wir rechnen also in elliptischen Kurven über dem Ring $\mathbb{Z}/N\mathbb{Z}$ mit den gleichen Additionsformeln wie bei elliptischen Kurven über einem Körper. Dabei können zwei Fälle auftreten, wobei der erste der angenehme Fall zum Faktorisieren ist.

- Falls die Invertierung fehlschlägt, finden wir einen Teiler von N.
- Falls die Invertierung möglich ist, erhalten wir das korrekte Ergebnis der Punktaddition durch Anwenden der Additionsformeln aus Satz 1.3.

1.2 Der Algorithmus

1.2.1 Der Grundalgorithmus

Um die Indizierung zu vereinfachen, betrachten wir in der Folgezeit allgemein den Faktor p der Zahl N. Dieses p entspricht einem der Faktoren p_i aus (1.5).

Wie wir im vorigen Kapitel gesehen haben, finden wir einen Faktor der Zahl N, falls eine Punktaddition fehlschlägt. Dies ist genau dann der Fall, wenn das Ergebnis einer Punktaddition in $E(\mathbb{Z}/p\mathbb{Z})$ der Nullpunkt $\mathcal{O}_p$ ist. Diesen Aspekt nutzt man im ECM-Algorithmus aus. Wir berechnen ein Vielfaches $k \cdot P$ des Punktes P und hoffen, daß die Ordnung der elliptischen Kurve $E(\mathbb{Z}/p\mathbb{Z})$ für den Teiler p von N die Zahl k teilt. Wenn gleichzeitig die Ordnung von $E(\mathbb{Z}/N\mathbb{Z})$ die Zahl k nicht teilt, tritt folgende Situation auf:

1. $\forall\, c_1 \in \mathbb{N}: \quad k \neq c_1 \cdot \mathrm{ord}_{E(\mathbb{Z}/N\mathbb{Z})}(P)$,
2. $\exists\, c_2 \in \mathbb{N}: \quad k = c_2 \cdot \mathrm{ord}_{E(\mathbb{Z}/p\mathbb{Z})}(\phi_p(P))$.

Damit ist das Ergebnis der Berechnung von $k \cdot P$ — also k-faches Aufaddieren des Punktes P zu sich selbst —

$$k \cdot P \quad \neq \quad \mathcal{O}_N \qquad \text{und} \qquad k \cdot \phi_p(P) \quad = \quad \mathcal{O}_p. \tag{1.6}$$

Die Addition liefert den Faktor p. Wie wählen wir nun die Zahl k? Wir hoffen, daß die Gruppenordnung glatt ist, d.h. alle Teiler der Gruppenordnung liegen unter einer fest vorgegebenen Schranke B. Deshalb berechnen wir k als Produkt aller Primzahlen, die kleiner als diese Schranke B sind.
In der Realität führen wir die Berechnung von k nicht explizit aus, sondern verfahren wie im folgenden Algorithmus.

1.6. Algorithmus

ECM - Grundalgorithmus

(1) **do**

(2) Wähle zufällig eine elliptische Kurve E und einen endlichen Punkt P auf dieser Kurve.

(3) Multipliziere den Punkt P sukzessive mit Primzahlpotenzen $p_i^{e_i}$, wobei $p_i < B$ und $p_i^{e_i} \leq C < p_i^{e_i+1}$ für feste Schranken B, $C \in \mathbb{N}$ gilt. Falls diese Multiplikation des Punktes mit einem Skalar fehlschlägt, d.h. die Invertierung nicht ausführbar ist, finden wir einen Faktor > 1 der Zahl N.

(4) **while** (kein Faktor gefunden)

1.7. Bemerkung
Eine ausführliche Beschreibung mit Berechnung der Parameter B, C usw. finden wir in [Be93].
Neben den Verbesserungen, die wir in Kapitel 2 vorstellen werden, können wir den Algorithmus durch Anschließen einer 2. Phase, einer „Continuation", beschleunigen. Dabei nutzen wir die im Schritt (3) des obigen Algorithmus (die sogenannte „1. Phase" von ECM) berechneten Ergebnisse aus.

1.2.2 Die verbesserte Standard-Continuation

In der 1. Phase von ECM berechnen wir den Punkt

$$Q \quad = \quad k \cdot P \quad \text{mit} \quad k = \prod_{\mathbb{P} \ni q \leq B} q^{e_q}, \quad e_q = \max\{j \in \mathbb{N} : q^j \leq C\} \tag{1.7}$$

auf der elliptischen Kurve E. Angenommen, die Gruppenordnung von $E(\mathbb{Z}/p\mathbb{Z})$ hat nun folgende Primfaktorzerlegung:

$$\left| E\left({}^{\mathbb{Z}}\!/_{p\mathbb{Z}} \right) \right| \quad = \quad q_1^{e_1} \cdot q_2^{e_2} \cdot \dots \cdot q_{l-1}^{e_{l-1}} \cdot q_l^{e_l}, \quad q_i \in \mathbb{P},\ e_i \in \mathbb{N} \quad \text{für } i = 1, \dots, l \tag{1.8}$$

mit $q_1 < q_2 < \dots < q_{l-1} < q_l$. Um den Teiler p der Zahl N zu finden, müssen

1. alle Primteiler q_i der Ordnung $|E(\mathbb{Z}/p\mathbb{Z})|$ kleiner als die Schranke B sein,
2. alle Potenzen $q_i^{e_i}$ von Primteilern in der Ordnung $|E(\mathbb{Z}/p\mathbb{Z})|$ kleiner als C sein.

Wir gehen davon aus, daß sich die Gruppenordnung von $E(\mathbb{Z}/p\mathbb{Z})$ wie eine zufällige Zahl verhält. Dies konnte bisher nicht bewiesen werden! Bei heuristischen Untersuchungen hat man festgestellt, daß für zufällige Zahlen die Dezimallänge des größten Teilers ca. 63 % der Dezimallänge der Zahl beträgt, d.h. zufällige Zahlen haben häufig einen großen Teiler.

Nach der 1. Phase von ECM haben wir den Punkt Q aus (1.7) berechnet. Falls wir den Faktor p nicht gefunden haben, existiert noch mindestens ein Primteiler q_j der Ordnung $|E(\mathbb{Z}/p\mathbb{Z})|$, der größer als die Schranke B ist bzw. dessen Potenz $q_j^{e_j} > C$ ist. Damit gilt: $q_j^{e_j} \nmid k$.
Den Fall, daß nur genau ein Primteiler q_l der Ordnung mit Vielfachheit 1 größer als B ist, versuchen wir mit der Continuation abzudecken. Der Punkt Q hat also die Ordnung $\mathrm{ord}_{E(\mathbb{Z}/p\mathbb{Z})}(Q) = q_l$, somit haben wir bezogen auf (1.8) folgende Situation:

- $\forall\, i = 1, \dots l-1 : \quad q_i < B \quad \wedge \quad q_i^{e_i} < C,$
- $q_l > B \quad \wedge \quad e_l = 1.$

Die Idee für eine Continuation können wir direkt aus der obigen Betrachtung ablesen. Wir testen für alle Primzahlen q aus dem Intervall $[B, D]$ für eine neue Schranke $D > B$, ob

$$q \cdot Q \quad = \quad q \cdot k \cdot P \quad = \quad \mathcal{O}_p \qquad \text{in} \quad E\left(\mathbb{Z}/p\mathbb{Z}\right)$$

ist. Ist dies der Fall (d.h. auch $q_l \in [B, D]$), so finden wir dadurch den Faktor p.

1.8. Bemerkung

Zur Lösung dieses Problems gibt es mehrere Ansätze. Der triviale Ansatz, einfach den Punkt Q sukzessive mit allen Primzahlen aus dem Intervall $[B, D]$ zu multiplizieren, ist ineffizient. Ein besserer Ansatz, die sogenannte Standard-Continuation, nutzt aus, daß der Abstand zwischen zwei aufeinanderfolgenden Primzahlen relativ klein ist (siehe [Be93]). Durch diesen Ansatz werden die Multiplikationen eines Punktes mit einem Skalar, die im trivialen Ansatz auftreten, durch Punktadditionen ersetzt.

Wir stellen hier die verbesserte Standard-Continuation vor, weil sie in der Praxis die schnellste deterministische Variante ist. Dabei können wir die Punktaddition der Standard-Continuation durch eine Multiplikation im Ring $\mathbb{Z}/N\mathbb{Z}$ ersetzen, erhalten aber einen gewissen Overhead an Vorberechnungen. Die Idee hinter der verbesserten Standard-Continuation ist eine sogenannte Babystep-Giantstep-Strategie.

Für zwei beliebige Zahlen $q, w \in \mathbb{N}$ können wir Division mit Rest ausführen, wir erhalten

$$q = k \cdot w + r \quad \Longleftrightarrow \quad q = \underbrace{(k+1)}_{=:v} \cdot w - \underbrace{(w-r)}_{=:u}, \tag{1.9}$$

wobei

$$k = \left\lfloor \frac{q}{w} \right\rfloor, \qquad 0 \le r < w \qquad \text{und somit} \qquad 0 < u \le w$$

gilt. Daraus folgt

$$q \cdot Q \quad = \quad (vw - u) \cdot Q \quad = \quad \underbrace{vw \cdot Q}_{=:(x_1, y_1)} - \underbrace{u \cdot Q}_{=:(x_2, y_2)}.$$

Somit gilt für die Primzahl $q = q_l$ aus (1.8)

$$\begin{array}{rrcl} & q \cdot Q & = & \mathcal{O}_p \quad \text{in } E\left(\mathbb{Z}/p\mathbb{Z}\right) \\ \Leftrightarrow & vw \cdot Q & = & u \cdot Q \quad \text{in } E\left(\mathbb{Z}/p\mathbb{Z}\right) \\ \Rightarrow & x_1 & \equiv & x_2 \mod p \\ \Leftrightarrow & x_1 - x_2 & \equiv & 0 \mod p \end{array}$$

$$\begin{array}{rl} \Rightarrow & p \mid (x_1 - x_2) \\ \Rightarrow & \mathrm{ggT}(x_1 - x_2, N) \ge p. \end{array}$$

Wir stellen also jede Primzahl aus dem Intervall $[B, D]$ wie in (1.9) dar. Um die Tests durchzuführen, müssen wir zum einen die x-Koordinaten der Punkte $u \cdot Q$ für alle möglichen $u = 1, \ldots, w$ bestimmen, zum anderen die x-Koordinaten der Punkte $v \cdot w \cdot Q$ für alle $v = \lfloor (B/w) \rfloor, \ldots, \lfloor (D/w) \rfloor$ berechnen. Der Wert von w bestimmt also die Anzahl der möglichen Werte für u und v und somit die Anzahl der x-Koordinaten, die wir berechnen müssen. Als nahezu optimalen Wert für w ergibt sich

$$w = \left\lceil \sqrt{D} \right\rceil .$$

1.9. Bemerkung

1. Durch Verwendung der Primzahlen in aufsteigender Reihenfolge müssen wir nur die x-Koordinaten der Punkte $u \cdot Q$ für alle möglichen u vorberechnen und speichern, die x-Koordinaten von $vw \cdot Q$ berechnen wir jeweils sukzessive nach der Vorschrift

$$(v+1) \cdot w \cdot Q = v \cdot w \cdot Q + w \cdot Q.$$

2. Wir testen nur Primzahlen q aus dem Intervall $[B, D]$ als möglichen größten Teiler der Ordnung $|E(\mathbb{Z}/p\mathbb{Z})|$. Deshalb berechnen wir nur die x-Koordinaten von $u \cdot Q$ für alle möglichen Werte von u, die gleichzeitig die Bedingung $\mathrm{ggT}(u, w) = 1$ erfüllen.
   Wenn wir sicherstellen, daß w durch die Primzahlen 2, 3, 5, 7 teilbar ist, so verringert sich die Zahl der zu speichernden x-Koordinaten auf

$$\frac{\varphi(2 \cdot 3 \cdot 5 \cdot 7)}{2 \cdot 3 \cdot 5 \cdot 7} w = \frac{\varphi(210)}{210} w = \frac{48}{210} w \approx 0.229 w.$$

1.10. Algorithmus

Verbesserte Standard-Continuation

Wir speichern die x-Koordinaten der möglichen Werte $u \cdot Q$ im Vektor b und können mit $b[i]$ auf den i-ten Eintrag zugreifen und erhalten den Wert der x-Koordinate von $i \cdot Q$. Die Funktion $nextprime()$ liefert beim ersten Aufruf die erste Primzahl, die größer als B ist, danach jeweils die nächstgrößere Primzahl.

Eingabe: $Q = k \cdot P$, Schranken B, D
Ausgabe: Faktor p, falls gefunden, sonst 1

INITIALISIERUNG

(1) $w := \left\lfloor \left\lceil \sqrt{D} \right\rceil / 210 \right\rfloor \cdot 210$ // $210 \mid w$
(2) $v := \lfloor B/w \rfloor$ // v $\cong$ 1. Giantstep
(3) $Q_2 := 2 \cdot Q$
(4) $P := Q$

BABYSTEPS

(5) **for** $(j := 1; j \leq w; j := j + 2)$ **do**
(6)   **if** $(\mathrm{ggT}(w, j) = 1)$ **then** // Babystep speichern
(7) $b[j] := x[P]$
(8)   **fi**
(9) $P := P + Q_2$ // $j \to j + 2$
(10) **od**
(11) $Q := P - Q$

GIANTSTEPS

(12) $P := v \cdot Q$ // 1. Giantstep
(13) $q := nextprime()$
(14) **while** $(q < D)$ **do**
(15)   **if** $((u := vw - q) < 1)$ **then** // nächster Giantstep
(16) $v := v + 1$
(17) $P := P + Q$
(18) $u := vw - q$
(19)   **fi**
(20) $c := x[P] - b[u]$
(21)   **if** $(\mathrm{ggT}(c, N) > 1)$ **then** // Faktor gefunden
(22)   **return** $(\mathrm{ggT}(c, N))$
(23)   **fi**
(24) $q := nextprime()$
(25) **od**
(26) **return** (1)

1.11. Bemerkung

Da ggT-Berechnungen auf langen Zahlen ziemlich teuer sind, können wir in Schritt (19) die Differenzen aufmultiplizieren und den ggT z.B. nur alle 500 Schritte berechnen. Damit erhalten wir für jede Primzahl aus dem Intervall $[B, D]$ asymptotisch eine Multiplikation mit Reduktion.

Kapitel 2

Verbesserungen der Grundversion

2.1 Die Montgomery-Chudnowsky-Form

Neben der schon vorgestellten Weierstraß-Parametrisierung für elliptische Kurven und Punkte auf einer Kurve gibt es noch weitere Parametrisierungen. Wir stellen hier die Montgomery-Chudnowsky-Parametrisierung vor. Dabei geben wir Regeln zum Rechnen in dieser Parametrisierung an und zeigen eine Transformation auf die Weierstraß-Parametrisierung.

2.1.1 Kurven in Montgomery-Chudnowsky-Form

Die Montgomery-Chudnowsky-Parametrisierung hat die zur Anwendung im ECM-Algorithmus schöne Eigenschaft, daß man Punktadditionen und -verdoppelungen ohne Invertierung durchführen kann, wie wir im folgenden sehen werden. Außerdem können wir elliptische Kurven und Punkte aus der Montgomery-Chudnowsky-Form auf die Weierstraß-Form transformieren. Daraus ergibt sich schon der Einsatzbereich für diese Parametrisierung. Wir generieren eine Kurve in Montgomery-Chudnowsky-Form und rechnen die 1. Phase von ECM. Nach der Transformation von elliptischer Kurve und Punkt führen wir die Berechnungen in der Continuation dann mit Punkten auf einer Kurve in Weierstraß-Form aus.

2.1. Definition
Seien $\hat{a}, \hat{b} \in \mathbb{Z}/p\mathbb{Z}$, so daß $\mathrm{ggT}((\hat{a}^2-4)\hat{b}, p) = 1$. *Eine* **elliptische Kurve** $\hat{E}_{\hat{a},\hat{b}}$ *oder kürzer $\hat{E}$ in Montgomery-Chudnowsky-Form über dem Körper $\mathbb{Z}/p\mathbb{Z}$ wird beschrieben durch die Gleichung*

$$\hat{b}y^2z = x^3 + \hat{a}x^2z + xz^2. \tag{2.1}$$

Die **Menge aller Punkte** $E(\mathbb{Z}/p\mathbb{Z})$ *ist*

$$\hat{E}\left({\mathbb{Z}}/{p\mathbb{Z}}\right) \quad = \quad \left\{(x:y:z) \in \mathbb{P}^2 \;\middle|\; \hat{b}y^2z = x^3 + \hat{a}x^2z + xz^2\right\}. \tag{2.2}$$

Wie in der Weierstraß-Darstellung repräsentiert ein **Punkt** $(x : y : z)$ *wiederum eine Äquivalenzklasse. Die Äquivalenzrelation ist*

$$(x, y, z) \sim (x', y', z') \iff \exists\, c \in \mathbb{K}^* : cx = x', cy = y', cz = z'.$$

Der **Nullpunkt** *ist der Punkt* $(0 : 1 : 0)$*, der einzige Punkt mit z-Koordinate* 0.

2.2. Bemerkung
Bei der folgenden Vorstellung der arithmetischen Operationen, also Punktaddition und -verdoppelung, sieht man, daß nur der Koeffizient $\hat{a}$ aus der Kurvendefinition (2.1) benötigt wird. Bei Punkten genügt die Berechnung von x- und z-Koordinate. Die y-Koordinate geht weder bei der Addition von Punkten, noch bei der Transformation auf die Weierstraß-Form ein. Deshalb identifizieren wir im folgenden jeden Punkt $(x : y : z)$ mit $(x : -- : z)$, d.h. der Wert der y-Koordinate wird bei allen Berechnungen nicht näher spezifiziert.

Mit Hilfe der in [Mo92/2] angegebenen Transformationen kann man die Addition von Punkten auf die Montgomery-Chudnowsky-Parametrisierung übertragen. In den folgenden Abschnitten stellen wir die Additionsformeln vor. Weil dabei keine Invertierung ausgeführt werden muß, können wir diese Operationen direkt für Kurven über dem Ring $\mathbb{Z}/N\mathbb{Z}$ einführen.

2.1.2 Arithmetik in Montgomery-Chudnowsky-Form

Verdoppelung von Punkten

Sei $P = (x : -- : z) \in E(\mathbb{Z}/N\mathbb{Z})$. Dann ist

$$2 \cdot P = \left((x+z)^2 (x-z)^2 : -- : 4xz \left((x-z)^2 + 4xz \frac{\hat{a}+2}{4} \right) \right). \tag{2.3}$$

Dabei können wir den Term $(\hat{a}+2)/4$ vorberechnen, weil er für eine feste Kurve konstant ist, durch Nachrechnen erhalten wir $4xz = (x+z)^2 - (x-z)^2$.

Wir benötigen also für eine Punktverdopplung 4 Additionen, 3 Multiplikationen und 2 Quadrierungen, alle im Ring $\mathbb{Z}/N\mathbb{Z}$.

Addition bei bekannter Differenz

Um 2 verschiedene Punkte P, Q in Montgomery-Chudnowsky-Parametrisierung zu addieren, muß die Differenz $D := P - Q$ bekannt sein. Sei $P = (x_1 : -- : z_1), Q = (x_2 : -- : z_2)$ und $D = (x_d : -- : z_d)$. Dann ist

$$P + Q = \left(4z_d \left(x_1 x_2 - z_1 z_2 \right)^2 : -- : 4x_d \left(x_1 z_2 - x_2 z_1 \right)^2 \right)$$

Wir betrachten die Terme $t_1 = (x_1 - z_1)(x_2 + z_2)$ und $t_2 = (x_1 + z_1)(x_2 - z_2)$. Dann gilt:

$$\begin{aligned} t_1 + t_2 &= (x_1 - z_1)(x_2 + z_2) + (x_1 + z_1)(x_2 - z_2) \\ &= x_1x_2 + x_1z_2 - x_2z_1 - z_1z_2 + x_1x_2 - x_1z_2 + x_2z_1 - z_1z_2 \\ &= 2(x_1x_2 - z_1z_2). \end{aligned}$$

Analog kann man nachrechnen, daß $t_1 - t_2 = 2(x_1z_2 - x_2z_1)$ ist und damit gilt

$$P + Q = \left(z_d\left(t_1 + t_2\right)^2 : -- : x_d\left(t_1 - t_2\right)^2\right). \tag{2.4}$$

Für eine Addition von zwei verschiedenen Punkten sind 4 Additionen, 4 Multiplikationen und 2 Quadrierungen notwendig.

Multiplikation eines Punktes mit einem Skalar

Sei $P = (x, --, z)$ ein Punkt in Montgomery-Chudnowsky-Form auf der Kurve $\hat{E}_{\hat{a},\hat{b}}(\mathbb{Z}/N\mathbb{Z})$ und $m \in \mathbb{N}$. Wir wollen den Punkt $Q = m \cdot P$ berechnen. Falls dabei m gerade ist, gilt $m = 2^k \cdot m'$ für $k \geq 1$ und ungerades m'. Wir verdoppeln den Punkt P k-mal und berechnen danach $m' \cdot P$. Somit müssen wir die Multiplikation eines Punktes mit einem Skalar nur für ungerade Skalare angeben. Im folgenden Algorithmus führen wir die Bearbeitung des geraden Anteils des Skalars in der Initialisierungsphase durch. Damit muß nicht gefordert werden, daß der Skalar ungerade ist.

Aus den Betrachtungen in den vorigen Abschnitten ist ersichtlich, daß bei der Addition zweier Punkte die Differenz der Punke bekannt sein muß. Dies müssen wir im Algorithmus sicherstellen. Dazu betrachten wir bei der Multiplikaton o.B.d.A. einen ungeraden Skalar m.
Sei $m = [b_d b_{d-1} \cdots b_1 b_0]_2$ die Binärdarstellung dieses Skalars. Wir führen nun eine schnelle Exponentiation von links nach rechts durch. Dabei berechnen wir $m \cdot P$ und $(m + 1) \cdot P$ parallel, erhalten also eine Folge von Paaren (P_1, P_2). Diese erfüllt im Algorithmus folgende Invariante:

1. In jedem Schritt gilt : $P_2 - P_1 = P$.
2. In der i-ten Iteration gilt: $P_1 = [b_d \cdots b_{d-i}]_2 \cdot P \ \wedge \ P_2 = [b_d \cdots b_{d-i}]_2 \cdot P + P$

Mit dieser Invariante stellen wir sicher, daß nach dem d-ten Schritt in P_1 das Ergebnis $m \cdot P$ steht. Nach dem Algorithmus beweisen wir, daß diese Invariante für den angegebenen Algorithmus gilt.

2.3. Algorithmus

Multiplikation eines Punktes in Montgomery-Chudnowsky-Darstellung mit einem Skalar mit Hilfe eines binären Algorithmus

EINGABE: $m \in \mathbb{N}$, Punkt P
AUSGABE: $m \cdot P$

<u>INITIALISIERUNG</u>

$P_1 := P$
while $(m \equiv 0 \bmod 2)$ **do** // m gerade
    $P_1 := 2 \cdot P_1$
    $m := m/2$
od
$(P_1, P_2) := (P_1, 2 \cdot P_1)$
$d := \lfloor \log_2(m) \rfloor$

<u>BERECHNUNG FÜR BITS $d - 1 \cdots 1$</u>

for $(i := d-1; i > 0; i--)$ **do**
    $Q := P_1 + P_2$
    if $(b_i = 1)$ **then**
        $P_1 := Q$ // i-tes Bit von $m = 1$
        $P_2 := 2 \cdot P_2$
    else
        $P_1 := 2 \cdot P_1$ // i-tes Bit von $m = 0$
        $P_2 := Q$
    fi
od

<u>BERECHNUNG FÜR DAS LETZTE BIT</u>

$P_1 := P_1 + P_2$
return (P_1)

Beweis:
Die Invariante gilt bei der Skalarmultiplikation des Punktes mit einer ungeraden Zahl. Wir führen den Beweis für eine ungerade Zahl m aus. Die Korrektheit für gerade Zahlen folgt direkt aus diesem Beweis unter Beachtung der Vorbemerkungen. Dazu führen wir eine vollständige Induktion über i aus.

Sei $i = 0$:
Das vorderste oder d-te Bit der Zahl m ist immer 1, wir initialisieren das Paar (P_1, P_2) mit $(P, 2 \cdot P)$. Damit ist die Behauptung erfüllt.

Sei die Behauptung für alle $i \leq k$ erfüllt. Wir zeigen nun den Schritt $k \rightarrow k+1$:
Im k-ten Schritt haben wir $P_1 = [b_d \cdots b_{d-k}]_2 \cdot P$ und $P_2 = [b_d \cdots b_{d-k}]_2 \cdot P + P$ berechnet. Wir betrachten nun das Bit b_{d-k-1} des Skalars m. Dabei führen wir den Fall $b_{d-k-1} = 1$ näher aus, der Fall $b_{d-k-1} = 0$ ergibt sich analog.
Sei $b_{d-k-1} = 1$. Die Bedingung in der if-Schleife im obigen Algorithmus (Schritt (10)) ist erfüllt, also berechnen wir:

$$\begin{aligned} P_1 &= P_1 + P_2 \\ &= [b_d \cdots b_{d-k}]_2 \cdot P + [b_d \cdots b_{d-k}]_2 \cdot P + P \\ &= 2 \cdot [b_d \cdots b_{d-k}]_2 \cdot P + P \\ &= [b_d \cdots b_{d-k}0]_2 \cdot P + P \\ &= [b_d \cdots b_{d-k} b_{d-k-1}]_2 \cdot P \end{aligned}$$

und

$$\begin{aligned} P_2 &= 2 \cdot P_2 \\ &= 2 \cdot \left([b_d \cdots b_{d-k}]_2 \cdot P + P \right) \\ &= [b_d \cdots b_{d-k}0]_2 \cdot P + 2 \cdot P \\ &= [b_d \cdots b_{d-k} b_{d-k-1}]_2 \cdot P + P. \end{aligned}$$

■

2.4. Bemerkung
Für das letzte oder 0.te Bit müssen wir keine Überprüfung durchführen. In der Initialisierung führen wir die Multiplikation des Punktes mit dem geraden Anteil des Skalars aus. Danach müssen wir den Punkt nur noch mit einem ungeraden Skalar multiplizieren. Weil für ungerade Zahlen das letzte Bit immer 1 ist, tritt der erste Fall ein. Wir müssen nur eine Punktaddition ausführen und sparen eine Punktverdoppelung ein.

Im Algorithmus 2.3 haben wir die Invariante, daß die Differenz der beiden Punkte (P_1, P_2) immer konstant P ist. Durch diese Einschränkung erhalten wir für viele Multiplikationen eines Punktes mit einem Skalar „schlechte" Folgen. Dies illustrieren wir in folgendem Beispiel.

2.5. Beispiel
Wir berechnen die Folge der Paare (P_1, P_2) für die Multiplikation mit dem Skalar $23 \cdot P = [10111]_2 \cdot P$. Als Initialisierung erhalten wir

$d = 4$ und $(P_1, P_2) = (P, 2 \cdot P)$.

Daraus ergibt sich die Folge

$$\begin{array}{lll} i = 3: & 3.Bit = 0 \implies & (2P, 3P) \\ i = 2: & 2.Bit = 1 \implies & (5P, 6P) \\ i = 1: & 1.Bit = 1 \implies & (11P, 12P). \end{array}$$

Das korrekte Ergebnis ist nun die Summe $11P + 12P = 23P$. Für diese Berechnung müssen wir 4 Punktadditionen und 4 Punktverdoppelungen ausführen.

Nun geben wir eine zweite Folge zur Berechnung von $23 \cdot P$ an. Bei dieser müssen nur 6 Punktadditionen und 1 Punktverdoppelung ausgeführt werden. Dabei ist das 3. Element jeweils die Punktdifferenz der beiden ersten Punkte.

(2P, 3P , P)
(3P, 5P , 2P)
(5P, 8P , 3P)
(5P,13P, 8P)
(5P,18P,13P)

Durch die Punktaddition $5P + 18P$ erhalten wir das Ergebnis. Wir sehen, daß diese zweite Folge länger als die erste Folge ist. In jedem Schritt muß wir aber jeweils nur ein Eintrag neuberechnet werden. Die beiden anderen ergeben sich durch Kopieren aus den Punkten im vorigen Folgenglied.

Wir führen nun einen Algorithmus ein, der solch eine bessere Folge von Punkten berechnet. Diese Variante ist ca. 12 % schneller als die binäre Methode, wie sich in der Praxis herausgestellt hat (siehe Tabelle 6.3). Die Methode basiert auf Lukas - Ketten. Eine ausführliche Beschreibung des Algorithmus steht in [Mo92/1].

Wir berechnen im folgenden Algorithmus eine Folge von Tripeln von Punkten $(P_1,\ P_2,\ P_3)$. P_3 entspricht dabei in jedem Schritt der Differenz $P_1 - P_2$. In unserer speziellen Anwendung der Lukas-Ketten gilt, daß die Differenzen $P_1 - P_2$ und $P_2 - P_1$ bis auf die y-Koordinate gleich sind. Da wir in unseren Berechnungen die y-Koordinate nicht benötigen, können wir bei der Punktaddition $P_1 + P_2$ die gleiche Punktdifferenz verwenden wie bei der Punktaddition $P_2 + P_1$.

Im Algorithmus und in der anschließenden Tabelle 2.1 verwenden wir die Notation

$$(d, e, P_1, P_2, P_3) \leftarrow (d', e', P_1', P_2', P_3').$$

Wir verstehen darunter eine Auswertung der Terme auf der rechten Seite und dann komponentenweise Zuweisung. Falls sich ein Wert in einem Schritt nicht ändert, so lassen wir diesen einfach weg.

2.6. Algorithmus

Multiplikation eines Punktes in Montgomery-Chudnowsky-Darstellung mit einem Skalar

EINGABE: $m, t \in \mathbb{N}, t|m$, Punkt P
AUSGABE: $m \cdot P$

INITIALISIERUNG

```
(d, P1) ← (m, P)
```

d DURCH 2 BZW. 3 TEILBAR?

```
while (d ≡ 0 mod 2) do                      // m gerade
   (d, P1) ← (d/2, 2 · P1)
od
while (d ≡ 0 mod 3) do                      // m durch 3 teilbar
   (d, P1) ← (d/3, 3 · P1)
od
```

BERECHNUNG

```
while (d ≠ 1) do
   if (t > d) then                          // t teilt d
      t := d
   fi
   r := (d/t) · ⌊t/α⌋                       // α = (1 + √5)/2
   e := 2 · r − d
   d := 2 · d − 3 · r
   (P1, P2, P3) ← (2 · P1, 3 · P1, P1)
   while (e ≠ d) do
      if (d < e) then
         (d, e, P1, P2) ← (e, d, P2, P1)
      Suche Fall aus Tabelle 2.1
   od
   P1 := P1 + P2
od
return (P1)
```

Nun geben wir in einer Tabelle die Bedingungen und dazugehörige Operationen an, die wir für die speziellen Werte von e, d ausführen müssen. Dabei unterscheiden wir zehn Fälle. Diese prüfen wir im Algorithmus in aufsteigender Reihenfolge ab. Falls ein Fall erfüllt ist, so führen wir die angegebenen Operationen aus und durchlaufen den nächsten Schleifendurchgang.

Fall	Bedingungen	Operationen
1	$d \leq 1.25 \cdot e$ $(d+e) \equiv 0 \bmod 3$	h := (d + e) / 3 $(d, e) \leftarrow (d-h, e-h)$ $(P_1, P_2) \leftarrow (2 \cdot P_1 + P_2, P_1 + 2 \cdot P_2)$
2	$d \leq 1.25 \cdot e$ $(d-e) \equiv 0 \bmod 6$	$d := (d-e)/2$ $(P_1, P_2) \leftarrow (2 \cdot P_1, P_1 + P_2)$
2a	$d \leq 1.25 \cdot e$	$d := d - e$ $(P_2, P_3) \leftarrow (P_1 + P_2, P_2)$
3	$d \leq 4 \cdot e$	$d := d - e$ $(P_2, P_3) \leftarrow (P_1 + P_2, P_2)$
4	$(d-e) \equiv 0 \bmod 2$	$d := (d-e)/2$ $(P_1, P_2) \leftarrow (2 \cdot P_1, P_1 + P_2)$
5	$d \equiv 0 \bmod 2$	$d := d/2$ $(P_1, P_3) \leftarrow (2 \cdot P_1, P_1 + P_3)$
6	$d \equiv 0 \bmod 3$	$d := d/3 - e$ $(P_1, P_2, P_3) \leftarrow (3 \cdot P_1, 3 \cdot P_1 + P_2, P_2)$
7	$(d+e) \equiv 0 \bmod 3$	$d := (d+e)/3 - e$ $(P_1, P_2) \leftarrow (3 \cdot P_1, 2 \cdot P_1 + P_2$
8	$(d-e) \equiv 0 \bmod 3$	$d := (d-e)/3$ $(P_1, P_2, P_3) \leftarrow (3 \cdot P_1, P_1 + P_2, P_1 + P_3)$
9	$e \equiv 0 \bmod 2$	$e := e/2$ $(P_2, P_3) \leftarrow (2 \cdot P_2, P_1)$

Tabelle 2.1: Bedingungen für Algorithmus 2.6

Wie wir gesehen haben, wird bei den arithmetischen Operationen beim Rechnen in der Montgomery-Chudnowsky-Darstellung keine Invertierung durchgeführt. Deshalb kann bei den Berechnungen kein Fehler auftreten, es wird aber auch kein Faktor gefunden. Die z-Koordinate des Nullpunktes über dem Körper $\mathbb{Z}/p\mathbb{Z}$ ist kongruent $0 \bmod p$. Damit erhalten wir durch eine ggT-Berechnung der z-Koordinate und der zu faktorisierenden Zahl N möglicherweise den Faktor p.

Transformation auf Weierstraß-Form

In diesem Abschnitt wollen wir die Transformation von elliptischen Kurven aus der Montgomery-Chudnowsky-Form auf die Weierstraß-Form vorstellen. Um auch einen Punkt zu transformieren, muß die z-Koordinate des Punktes gleich 1 sein. Dabei nutzt man aus, daß nach Definition 2.1 gilt:

$$(x, --, z) \quad \sim \quad (xz^{-1}, --, 1).$$

Sei nun $(\hat{x}, --, 1) \in \hat{E}_{\hat{a},\hat{b}}(\mathbb{Z}/N\mathbb{Z})$ ein Punkt auf der Kurve $\hat{E} : \hat{b}\hat{y}^2 = \hat{x}^3 + \hat{a}\hat{x}^2 + \hat{x}$, wobei $\hat{y}$ einen nicht näher spezifizierten Wert hat. Durch die Transformation (siehe [BoLe93])

$$\begin{aligned} t &= \hat{x}^3 + \hat{a}\hat{x}^2 + \hat{x} \\ a &= \left(1 - \frac{\hat{a}^2}{3}\right) t^{-2} \\ b &= \frac{1}{3}\left(\frac{2}{9}\hat{a}^3 - \hat{a}\right) t^{-3} \end{aligned} \tag{2.5}$$

erhalten wir die Kurve $E_{a,b}$ in Weierstraß-Form, der Punkt auf der elliptischen Kurve E in Weierstraß-Form hat folgende Gestalt

$$(x, y) \quad = \quad \left(\left(\hat{x} + \frac{\hat{a}}{3}\right)t^{-1}, t^{-1}\right). \tag{2.6}$$

2.2 Kurvengeneratoren

Wir finden einen Faktor p in einer zusammengesetzten Zahl N, falls die Gruppenordnung der gewählten elliptischen Kurve E über dem Körper $\mathbb{Z}/p\mathbb{Z}$ in kleine Faktoren, d.h. Faktoren unter einen bestimmten Schranke, zerfällt.

Diese Gruppenordnung $|E(\mathbb{Z}/p\mathbb{Z})|$ liegt nach dem Satz von Hasse (siehe Satz A.5) im Intervall

$$\left[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}\right].$$

Um die Wahrscheinlichkeit zu erhöhen, einen Faktor zu finden, kann man durch geschickte Wahl einer elliptischen Kurve und eines Punktes auf dieser Kurve sicherstellen, daß die Gruppenordnung $|E(\mathbb{Z}/p\mathbb{Z})|$ für alle Primteiler p der Zahl N immer von vorher bestimmten Zahlen geteilt wird. Wir stellen nun zwei Algorithmen vor, die beide elliptische Kurven und Punkte auf dieser Kurve über dem Körper $\mathbb{Z}/p\mathbb{Z}$ in Montgomery-Chudnowsky-Parametrisierung generieren. Dabei ist die Gruppenordnung der elliptischen Kurve durch 12 bzw. 16 teilbar.

2.2.1 Der 12 - Generator

Wir wollen eine elliptische Kurve $\hat{E}$ generieren. Die Gruppenordnung $|\hat{E}(\mathbb{Z}/p\mathbb{Z})|$ soll für jeden Teiler p der Zahl N durch 12 teilbar sein. Dazu betrachten wir

$$\hat{a} = \frac{-3u^4 - 6u^2 + 1}{4u^3}, \quad \hat{b} = \frac{(u^2-1)^2}{4uv^2}$$

für zufällige Zahlen $u, v \in \mathbb{Z}$. Falls die Bedingung an $\hat{a}, \hat{b}$ aus Definition 2.1, also $\mathrm{ggT}((\hat{a}^2 - 4)\hat{b}, N) = 1$, erfüllt ist, definiert dies eine elliptische Kurve in Montgomery-Chudnowsky-Parametrisierung. Die Gruppenordnung $|\hat{E}(\mathbb{Z}/p\mathbb{Z})|$ ist für jeden Primteiler p von N durch 12 teilbar (siehe [AtMo92]). Wir müssen jetzt Bedingungen an u, v stellen, so daß wir $\hat{a}, \hat{b}$ berechnen können und in diesem Fall die Bedingung aus der Kurvendefinition überprüfen. Dies führt uns zu beiden Bedingungen

$$\mathrm{ggT}\left(u \cdot v \cdot \left(u^2 - 1\right), N\right) \quad = \quad 1$$

und

$$\mathrm{ggT}\left(\left(\frac{-3u^4 - 6u^2 + 1}{4u^3} - 2\right)\left(\frac{-3u^4 - 6u^2 + 1}{4u^3} + 2\right), N\right) \quad = \quad 1.$$

Durch die erste Bedingung ist schon sichergestellt, daß u und damit auch $4u^3$ modulo N invertierbar ist. Wir rechnen nun die zweite Bedingung nach:

$$\begin{aligned}
& \left(\frac{-3u^4 - 6u^2 + 1}{4u^3} - 2\right) \cdot \left(\frac{-3u^4 - 6u^2 + 1}{4u^3} + 2\right) && \equiv 0 \mod N \\
\iff & \left(\frac{3u^4 + 8u^3 + 6u^2 - 1}{4u^3}\right)\left(\frac{3u^4 - 8u^3 + 6u^2 - 1}{4u^3}\right) && \equiv 0 \mod N \\
\iff & (3u^4 + 8u^3 + 6u^2 - 1) \cdot (3u^4 - 8u^3 + 6u^2 - 1) && \equiv 0 \mod N \\
\iff & (3u - 1) \cdot (u + 1)^3 \cdot (3u + 1) \cdot (u - 1)^3 && \equiv 0 \mod N.
\end{aligned}$$

Folgende Bedingung an u und v deckt alle vorne angegebenen Fälle ab, die nicht auftreten dürfen:

$$\mathrm{ggT}\left(uv(u^2 - 1)(9u^2 - 1), N\right) \quad = \quad 1.$$

Als endlichen Punkt auf $\hat{E}$ wählen wir $P = (\hat{x}_0, --, 1)$ mit $\hat{x}_0 = (3u/4)$. Setzen wir diesen Punkt P in die Kurvengleichung (2.1) ein, so erhalten wir:

$$\frac{(u^2-1)^2}{4v^2} y^2 = u \cdot \left(\hat{x}_0^3 + \frac{-3u^4 - 6u^2 + 1}{4u^3}\hat{x}_0^2 + \hat{x}_0\right).$$

Weil der Wert von y nicht bestimmt wird, muß nur sichergestellt sein, daß auf der rechten Seite der obigen Gleichung ein Quadrat steht. Durch Einsetzen von $\hat{x}_0$ und

Ausrechnen erhalten wir, daß $(9-6u^2)/64$ ein Quadrat sein muß. Dies erreichen wir, indem wir $u = 6s/(s^2+6)$ wählen. s ist dabei eine zufällige Zahl aus $\mathbb{Z}$. Beim Einsetzen dieses speziellen Wertes für u gilt:

$$\begin{aligned}(9-6u^2)/64 &= \left(9-6\left(\frac{6s}{s^2+6}\right)^2\right)\cdot 64^{-1} = \frac{9(s^2+6)^2-36s^2}{64(s^2+6)^2} \\ &= \frac{9s^4+108s^2+324-36s^2}{64(s^2+6)^2} = \frac{(3s^2+18)^2}{64(s^2+6)^2}.\end{aligned}$$

Wir sehen, daß sich beim Einsetzen von $u = 6s/(s^2+6)$ in $(9-6u^2)/64$ für jeden beliebigen Wert $s \in \mathbb{Z}$ per Konstruktion ein Quadrat ergibt und unsere Bedingung erfüllt ist.
Weil man für ECM den Wert der b-Koordinate der generierten elliptischen Kurve nicht benötigt, muß er auch nicht berechnet werden. Deshalb wählen wir nur ein $s \in \mathbb{Z}$ zufällig und bestimmen daraus den Wert von u. Der Wert der zufälligen Zahl v wird nicht berechnet.

2.7. Algorithmus

Generierung einer elliptischen Kurve E und eines Punktes P in Montgomery-Chudnowsky-Form
Alle Berechnungen werden modulo N durchgeführt. Falls eine Invertierung fehlschlägt, finden wir schon hier einen Faktor.

AUSGABE: Kurve E, Punkt P oder Faktor von N

(1) Wähle $s \in \mathbb{Z}$ zufällig.
(2) $u :=\ 6s/(s^2+6)$
(3) $d :=\ \text{ggT}(u(u^2-1)(9u^2-1), N)$
(4) **if** $(d > 1)$ **then** // Faktor gefunden
(5)   **return** (d)
(6) **fi**
(7) $a :=\ (-3u^4-6u^2+1)/(4u^3)$
$E :=\ (a, --)$
(8) $x :=\ (3u/4)$
$P :=\ (x, --, 1)$
(9) **return** (E, P)

2.2.2 Der 16 - Generator

Nun stellen wir noch einen Algorithmus vor, der eine elliptische Kurve in Montgomery-Chudnowsky-Parametrisierung und einen Punkt generiert, wobei die Ordnung der elliptischen Kurve durch 16 teilbar ist (siehe [AtMo92]).

Wir wählen die Kurve E mit der Kurvengleichung $E : T^2 = S^3 - 8S - 32$ in Weierstraß-Form. Diese Kurve hat über $\mathbb{Q}$ die Torsionsgruppe $\mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$. Zu jedem rationalen Punkt auf dieser Kurve können wir durch eine Transformation eine elliptische Kurve in der sogenannten Kubert-Form erzeugen. Dabei entspricht jeder rationale Punkt P von E genau einer Kurve in Kubert-Form. Elliptische Kurven in Kubert-Form haben folgende Gestalt:

$$E(b,c) : Y^2 + (1-c)XY - bY = X^3 - bX^2.$$

Außerdem kennen wir den Punkt $P = (12, 40)$, der auf der oben angegebenen Kurve liegt. Dieser Punkt P ist kein Torsionspunkt. Durch Multiplikation dieses Punktes mit einem Skalar können wir uns weitere Nichttorsionspunkte auf der Kurve E bestimmen. Diese führen uns mit folgender Transformation (siehe [AtMo92]) auf die Kurve in Kubert-Form:

$$\alpha = \left(\frac{t+25}{s-9} + 1\right)^{-1}$$

und

$$d = \frac{2\alpha(4\alpha+1)}{8\alpha^2 - 1} \qquad c = \frac{(2d-1)(d-1)}{d} \qquad b = cd.$$

Neben der Kurve benötigen wir auch noch einen endlichen Punkt auf dieser Kurve. Dieser hat in unserem Fall die x-Koordinate $-(2d-1)/4$.

Nun müssen wir noch den Punkt und die elliptische Kurve in die Montgomery-Chudnowsky-Form transformieren. Mit der Substitution (siehe [Mo92/2])

$$\begin{aligned} Y &= (d-1)^3 y - \frac{(1-c)x}{2} + \frac{b}{2} \\ X &= (d-1)^2 x + d^2 - d, \end{aligned}$$

erhalten wir als x-Koordinate eines Punktes in Montgomery-Chudnowsky-Form den Wert

$$x = \frac{X - d^2 + d}{(d-1)^2}.$$

Die transformierte y-Koordinate müssen wir nicht berechnen, die z-Koordinate ist gleich 1. Durch Einsetzen der obigen Substitution berechnen wir die Koordinate a der Kurve durch

$$a = \left(\frac{(2d-1)^2}{2d(d-1)}\right)^2 - 2.$$

Wir sehen, daß wir zur Berechnung der x-Koordinate des Punktes und der a-Koordinate der Kurve nur den Wert von d benötigen, wir müssen also nicht explizit die Kurve in Kubert-Form berechnen. Die Variablen b, c werden nur zur Herleitung der verschiedenen Transformationen benötigt.

2.8. Algorithmus

Generierung einer elliptischen Kurve E und eines Punktes P in Montgomery-Chudnowsky-Form
Alle Berechnungen werden modulo N durchgeführt. Falls eine Invertierung fehlschlägt, finden wir schon hier einen Faktor. Außerdem kann bei der Multiplikation des Punktes P mit dem Skalar m in Schritt (2) ein Faktor gefunden werden.

EINGABE: $m \in \mathbb{N}$
AUSGABE: Kurve $\hat{E}$, Punkt $\hat{P}$ oder Faktor von N

INITIALISIERUNG

(1) $E : T^2 = S^3 - 8S - 32$
$P := (12, 40)$
(2) $P := m \cdot P$

GENERIERUNG DER KUBERT-KURVE

(3) $\alpha := \left(\dfrac{y[P] + 25}{x[P] - 9} + 1 \right)^{-1}$
(4) $d := 2\alpha(4\alpha + 1)/(8\alpha^2 + 1)$
(5) $X := -\dfrac{(2d-1)}{4}$

TRANSFORMATION AUF MONTGOMERY-CHUDNOWSKY-FORM

(6) $\hat{x} := (X + d - d^2)/(d-1)^2$
$\hat{P} := (\hat{x}, --, 1)$
(7) $\hat{a} := \dfrac{(2d-1)^2}{2d(d-1)} - 2$
$\hat{E} := (\hat{a}, --)$
(8) **return** $(\hat{E}, \hat{P})$

2.3 Die Wahl der Parameter

In diesem Abschnitt beschäftigen wir uns kurz mit der Wahl der Parameter. Eine ausführliche Beschreibung finden wir in [Be93]. Wir gehen hier auf die Veränderungen durch die Verwendung der Montgomery-Chudnowsky-Parametrisierung ein.

Wir betrachten dabei den 16-Generator, die Gruppenordnung der Kurve ist also für jeden Primteiler der zu faktorisierenden Zahl durch 16 teilbar.

In der 1. Phase führen wir alle Berechnungen in der Montgomery-Chudnowsky-Parametrisierung aus. Wir berechnen

$$Q \quad = \quad k \cdot P \quad \text{mit} \quad k = \prod_{\mathbb{P} \ni q \leq B} q^{e_q}, \quad e_q = \max\{j \in \mathbb{N} : q^j \leq C\}.$$

Dabei wählen wir die Schranke $C = B$. Es ist sehr unwahrscheinlich, daß bei Primzahlen mit z.B. 33 Dezimalstellen ein Primteiler größer als 10^5 der Gruppenordnung in der Vielfachheit größer als 1 auftritt. Diese Einschränkung spart Rechenzeit und war bei allen Tests ausreichend. Als Algorithmus zur Multiplikation eines Punktes mit einem Skalar wenden wir dabei den Algorithmus 2.6 an.
Nach der Transformation auf Weierstraß-Koordinaten führen wir die verbesserte Standard-Continuation aus. Dabei berechnen wir zuerst alle möglichen Babysteps vor. Dazu benötigen wir $\sqrt{D}/2$ Punktadditionen. In der anschließenden Giantstep-Phase berechnen wir $\sqrt{D}$ Giantsteps, also $\sqrt{D}$ Punktadditionen. Diese fassen wir zusammen und erhalten für die Continuation $\frac{3}{2} \cdot \sqrt{D}$ Punktadditionen.
Außerdem müssen wir für jede Primzahl einen ggT mit N berechnen. Mit Hilfe von Bemerkung 1.11 können wir diesen ggT durch eine Multiplikation mit Reduktion modulo N ersetzen. Dabei ist die Anzahl der Primzahlen im Intervall $[B, D]$ ungefähr

$$\frac{D}{\ln D} \quad - \quad \frac{B}{\ln B}.$$

Bisher haben wir nur die Anzahl der Operationen betrachtet, die wir bei einem Durchlauf von ECM ausführen müssen. Gleichzeitig müssen wir auch die Wahrscheinlichkeit abschätzen, daß die Ordnung so gut zerfällt, daß wir mit den gewählten Parametern eine Chance haben. Dazu verwenden wir die μ-Funktion (siehe [Kn81]). Sei $M \in \mathbb{N}$ fest. Für eine Zahl $q = q_1 \cdot q_2 \cdot \dots \cdot q_r$, $q_1 \leq q_2 \leq \dots \leq q_r$, $q_i \in \mathbb{P}$ für $i = 1, \dots, r$ in der Nähe von M und $\alpha - 1 \geq \beta \geq 1$ definieren wir

$$\mu(\alpha, \beta) := \lim_{M \to \infty} P\left(q_{r-1} < M^{\frac{1}{\alpha}}, q_r < M^{\frac{\beta}{\alpha}}\right).$$

Diese Funktion beschreibt also die Wahrscheinlichkeit, daß der größte Primteiler der Zahl q unter der Schranke $M^{\frac{\beta}{\alpha}}$ liegt. Für den zweitgrößten Faktor soll die Schranke $M^{\frac{1}{\alpha}}$ gelten. Wie wir diese Funktion berechnen, ist in [Be93] ausgeführt.
Um jetzt optimale Parameter zu berechnen, haben wir nicht die theoretisch erwartete Anzahl von Operationen genommen, sondern Tests mit verschiedenen Parametern durchgeführt und dabei die Laufzeiten der einzelnen Schritte gemessen. Mit Hilfe der Mathematica-Funktion *ecmpar2* (siehe [Be93]) haben wir dann die optimalen Parameter berechnet. Auf der folgenden Seite sind die Parameter, die wir für die aktuelle Implementierung von ECM berechnet haben. Dabei generieren wir eine Kurve mit dem 16-Generator, führen die erste Phase mit Punkten in Montgomery-Chudnowsky-Form aus. Nach einer Transformation auf Weierstraß-Form schließen wir die verbesserte Standard-Continuation an.

Tabelle 2.2: optimale Parameter für ECM mit verbesserter Std.Continuation

#Stellen	B	D	#Kurven
10	411	8 858	6
11	657	17 982	8
12	997	31 494	10
13	1 439	54 615	13
14	2 106	89 500	17
15	3 071	144 949	21
16	4 387	245 713	27
17	6 281	382 890	33
18	8 839	577 722	42
19	12 241	845 946	54
20	16 337	1 285 635	68
21	21 982	1 788 859	88
22	29 860	2 508 237	111
23	41 006	3 550 150	137
24	53 214	5 139 913	172
25	71 535	6 996 392	215
26	97 494	9 650 918	261
27	125 654	13 767 225	325
28	168 272	18 372 001	398
29	223 559	24 108 064	489
30	280 894	33 036 146	612
31	376 514	43 533 099	736
32	489 249	54 671 451	909
33	631 461	77 351 879	1 088
34	830 049	97 901 685	1 315
35	1 031 865	131 338 190	1 621
36	1 374 222	167 878 973	1 914
37	1 730 979	228 723 578	2 305
38	2 281 220	285 997 636	2 730
39	2 842 365	383 172 401	3 296
40	3 709 051	469 274 438	3 914
41	4 574 575	618 960 601	4 738
42	6 070 570	769 184 676	5 487
43	8 241 002	1 147 978 630	5 960
44	12 096 136	1 612 280 756	6 191

Kapitel 3

Schnelle Polynomarithmetik

In den beiden ersten Kapiteln haben wir den ECM - Grundalgorithmus vorgestellt und verschiedene Verbesserungen angegeben. Peter L. Montgomery beschreibt eine schnellere, probabilistische Continuation, die sogenannte FFT-Continuation. Dazu benötigt man eine schnelle Arithmetik zum Rechnen mit Polynomen über dem Ring $\mathbb{Z}/N\mathbb{Z}$.

Wir stellen nun Algorithmen zur Lösung der für die FFT-Continuation interessanten Probleme vor, so z.B. Multiplikation von Polynomen oder Aufbau eines Polynoms aus seinen Wurzeln. Dabei führen wir die Algorithmen zuerst über dem Körper $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$ für eine Primzahl p ein und geben dann die Erweiterungen auf den Ring $\mathbb{Z}/N\mathbb{Z}$ an.

3.1 Polynommultiplikation mit FFT über $\mathbb{Z}/p\mathbb{Z}$

Seien $F(x)$, $G(x)$ Polynome vom Grad $n-1$ über dem Körper $\mathbb{Z}/p\mathbb{Z}$,

$$F(x) = \sum_{i=0}^{n-1} f_i\, x^i\ , \qquad G(x) = \sum_{j=0}^{n-1} g_j\, x^j.$$

Die Koeffizienten f_i, g_j liegen in $\mathbb{Z}/p\mathbb{Z}$, diese Darstellung nennt man Standarddarstellung. Mit der Schulmethode zur Multiplikation erhalten wir dann als Produkt

$$H(x) = F(x) \cdot G(x) = \sum_{i=0}^{2n-2} \left(\sum_{j=0}^{i} f_j g_{i-j} \right) x^i.$$

Dieser Algorithmus zur Multiplikation von $F(x)$ mit $G(x)$ ist in der Praxis relativ langsam. So benötigen wir auf einer "SUN SPARC station ELC" zur Multiplikation von zwei Polynomen vom Grad 1000 über einem Körper mit ungefähr 10^{100} Elementen 238 Sekunden.

Eine schnellerer Algorithmus zur Polynommultiplikation basiert auf der schnellen Fourier-Transformation (FFT). Um FFT bei der Polynommultiplikation anzuwenden, muß die Anzahl der Koeffizienten des Polynoms eine 2er-Potenz sein, d.h. der

Grad muß gleich $2^k - 1$ für ein $k \in \mathbb{N}$ sein. Im weiteren gehen wir davon aus, daß diese Bedingung erfüllt ist. Falls der Grad eines Polynoms diese Bedingung nicht erfüllt, kann man das Polynoms durch Hinzufügen von Koeffizienten mit Wert 0 auf den benötigten Grad erweitern.

Um Polynome mit FFT zu multiplizieren, müssen gewisse Anforderungen an den Grundkörper $\mathbb{K}$ erfüllt sein:

- Es existiert ein multiplikativ Inverses von 2, d.h. $p > 2$. (3.1)
- Es existiert $w \in \mathbb{K}$ mit $w^{n/2} = -1$.

Beide Voraussetzungen sind für Polynome über dem Körper $\mathbb{Z}/p\mathbb{Z}$ für Primzahlen $p > 2$ erfüllt, wenn $p \equiv 1 \mod n$ ist. w kann dann als $\frac{p-1}{n}$-te Potenz eines primitiven Elements in $\mathbb{Z}/p\mathbb{Z}$ gewählt werden.

Der FFT-Algorithmus transformiert die Standarddarstellung der Polynome in eine point-value-Darstellung. Wir repräsentieren also ein Polynom durch die Funktionswerte an n verschiedenen Stützstellen. Diese Darstellung bestimmt die Polynome eindeutig (siehe Satz A.9). Wir wählen speziell die Stützstellen $w^0, w^1, \ldots, w^{n-1}$ und erhalten damit

$$\begin{aligned} \hat{f} &= \left[F(1),\ F(w),\ F(w^2),\ \cdots,\ F(w^{n-1})\right], \\ \hat{g} &= \left[G(1),\ G(w),\ G(w^2),\ \cdots,\ G(w^{n-1})\right]. \end{aligned}$$

Die Vektoren $\hat{f}$ und $\hat{g}$ heißen Diskrete Fourier-Transformation (DFT) von F bzw. G. Durch ein komponentenweises Produkt ergibt sich

$$\hat{h} = \left[F(1) \cdot G(1),\ F(w) \cdot G(w),\ \cdots,\ F(w^{n-1}) \cdot G(w^{n-1})\right].$$

Dann ist $\hat{h}$ die DFT von $H(x) = F(x) \cdot G(x) \bmod (x^n - 1)$, wie wir durch Nachrechnen zeigen können. Durch Interpolation der Funktionswerte in $\hat{h}$ erhält man die Funktion $H(x) \bmod (x^n - 1)$ in Standarddarstellung.

Wir geben nun zuerst kurz an, wie man diese Algorithmen zum Multiplizieren von Polynomen nutzen kann. Danach beschreiben wir die Auswertung des Polynoms an den Stellen $1, w, w^2, \ldots, w^{n-1}$ mittels FFT. Anschließend folgt die Anwendung dieser Idee bei der Interpolation. Damit kann dann die Interpolation effizient durchgeführt werden.

Um ein vollständiges Ergebnis $H(x) = F(x) \cdot G(x)$ der Multiplikation mit FFT zu erhalten, setzen wir $m = \text{grad}(F) + \text{grad}(G) + 1$ und berechnen die DFT von F und G der Länge m. Damit erhalten wir nach der Interpolation von $\hat{h}$

$$H(x) = F(x) \cdot G(x) \bmod (x^m - 1) = F(x) \cdot G(x). \tag{3.2}$$

Diese Gleichheit ist gegeben, weil der Grad des Produktes maximal der Summe der Grade der Ausgangspolynome entspricht.

Wir gehen kurz auf die Auswertung des Polynoms mit FFT ein. Eine ausführliche Beschreibung findet man in [CoLeRi89]. Wir wollen das Polynom $F(x)$ aus der Standarddarstellung in die DFT transformieren. Das Polynom $F(x)$ hat $n = 2^k$ Koeffizienten für ein $k \in \mathbb{N}$.
Dazu verwenden wir eine Divide-and-Conquer Strategie. Wir teilen das Polynom $F(x) = \sum_{i=0}^{n-1} f_i \cdot x^i$ in zwei Teilpolynome $F_1(x)$ und $F_2(x)$ auf, wobei gilt:

$$\begin{aligned} F_1(x) &= f_0 + f_2 x + f_4 x^2 + \cdots + f_{n-2} x^{n/2-1}, \\ F_2(x) &= f_1 + f_3 x + f_5 x^2 + \cdots + f_{n-1} x^{n/2-1}. \end{aligned}$$

Damit erhalten wir

$$F(x) = F_1(x^2) + x \cdot F_2(x^2).$$

Wir ersetzen die Auswertung des Polynoms $F(x)$ vom Grad $n-1$ durch die Auswertung der Polynome $F_1(x), F_2(x)$, die Grad $(n/2-1)$ haben. Diese Strategie können wir fortsetzen und die Polynome weiter unterteilen. Auf der untersten Ebene erhält man Polynome vom Grad 1, die leicht ausgewertet werden können.

Die vorgestellte Rekursion kann man auflösen. Wir berechnen im voraus, welche Polynome vom Grad 1 ausgewertet werden müssen. Diese werten wir aus und bauen dann die Ergebnisse wie in der Rekursion zusammen. Damit erhalten wir nach $\log_2 n$ Schritten die komplette DFT von $F(x)$.

Die Interpolation können wir mit Hilfe der Lagrange-Formel durchführen. Die Laufzeit dieses Algorithmus macht die Laufzeitgewinne durch die Multiplikation in der point-value-Darstellung zunichte. Wir müssen also eine andere Strategie wählen.

Sei f der Koeffizientenvektor des Polynoms $F(x)$. Dann entspricht die vorgestellte FFT-Auswertung einer Multiplikation $\hat{f} = V_n \cdot f$, wobei V_n eine Vandermonde-Matrix (siehe Definition A.8) ist. Durch $V_n^{-1} \cdot \hat{f} = V_n^{-1} \cdot V_n \cdot f = f$ können wir die DFT in die Standarddarstellung transformieren. Durch Nachrechnen erhält man, daß diese Transformation genau einer FFT entspricht. Diese müssen wir an den Stellen $1, w^{-1}, w^{-2}, \ldots, w^{-n+1}$ durchführen. Man kann also den gleichen Algorithmus wie bei der Auswertung des Polynoms mit FFT verwenden, hat aber andere Stützstellen.

3.2 Übertragung auf $\mathbb{Z}/N\mathbb{Z}$

Wir wollen die Methoden aus dem letzten Abschnitt auch für Polynome über dem Ring $\mathbb{Z}/N\mathbb{Z}$ nutzen. Dafür müssen beide Bedingungen (3.1) erfüllt sein. Die 1. Bedingung ist erfüllt, wenn N ungerade ist, was wir o.B.d.A. annehmen können. Die 2. Bedingung ist meistens aber nicht gewährleistet, so daß wir eine andere Strategie wählen müssen.

Wir interpretieren die Polynome $F(x), G(x)$ als Polynome über dem Ring $\mathbb{Z}$, haben also Polynome in $\mathbb{Z}[x]$. Zum Rechnen über $\mathbb{Z}$ existiert eine Standardmethode. Wir berechnen das Ergebnis modular und setzen dann mit Hilfe des Chinesischen

Restsatzes die modularen Ergebnisse zusammen. Falls wir genügend viele modulare Ergebnisse berechnet haben, erhalten wir das Ergebnis in $\mathbb{Z}$. Dieses reduzieren wir modulo N und haben das korrekte Ergebnis im Ring $\mathbb{Z}/N\mathbb{Z}$.
In unserer Arithmetik verwenden wir als Moduln 26-Bit-Zahlen. Dadurch können wir alle modularen Berechnungen in den Datentypen *float* und *double* durchführen. Diese Berechnungen laufen damit sehr schnell ab.

Bevor wir die Anwendung des Chinesischen Restsatzes beschreiben, wollen wir an einem Beispiel seine Wirkungsweise illustrieren.

3.1. Beispiel
Sei $N = 524230993452474 50539$. Wir wollen die Polynome

$$F(x) = x - 655904991418877 \quad \text{und} \quad G(x) = x - 485644284403531 13$$

über $\mathbb{Z}/N\mathbb{Z}$ multiplizieren, beschränken uns hier im Beispiel auf die Berechnung des letzten Koeffizienten. Mit den Algorithmen aus Abschnitt 3.1 haben wir das Ergebnis h modulo der Primzahlen $p_1 = 66551809, p_2 = 66813953$ und $p_3 = 67043329$ bestimmt. Dabei bezeichnet h_i den Rest modulo p_i:

$$\begin{aligned} h_1 &\equiv 34239960 \quad \bmod 66551809, \\ h_2 &\equiv 55399418 \quad \bmod 66813953, \\ h_3 &\equiv 36196085 \quad \bmod 67043329. \end{aligned}$$

Mit Hilfe des Chinesischen Restsatzes wollen wir den Wert von $h \bmod P = p_1 \cdot p_2 \cdot p_3 = 298114158659380167442433$ berechnen. Dazu bestimmen wir Werte y_i, für die

$$y_i \equiv \left(\frac{P}{p_i}\right)^{-1} h_i \bmod p_i$$

gilt. In unserem Beispiel erhalten wir die Ergebnisse

$$\begin{aligned} y_1 &\equiv 18754848 \quad \bmod 66551809, \\ y_2 &\equiv 12326216 \quad \bmod 66813953, \\ y_3 &\equiv 35785230 \quad \bmod 67043329. \end{aligned}$$

Damit ergibt sich h als

$$\begin{aligned} h &\equiv \sum_{k=1}^{3} \frac{P}{p_i} y_i \quad \bmod P \\ &\equiv 298131037765819777567862 \bmod P \qquad (3.3) \\ &\equiv 16879106439610125429 \bmod P. \end{aligned}$$

Daraus folgt dann das Ergebnis $h = 16879106439610125429$.

Wir sehen, daß bei der Berechnung von $h \bmod P$ in (3.3) eine Reduktion modulo P durchgeführt werden muß. Um dies zu vermeiden, berechnen wir beim Zusammensetzen eine Approximation, wie oft P subtrahiert werden muß. Dadurch kann die Reduktion durch eine Multiplikation mit anschließender Subtraktion ersetzt werden. Im Beispiel ergibt sich über $\mathbb{R}$ für die Approximation

$$\left\lfloor \frac{1}{2} + \sum_{i=1}^{3} \frac{y_i}{p_i} \right\rfloor .$$

der Wert 1, wir ersetzen die Reduktion in (3.3) durch eine Subtraktion.

Bevor wir eine allgemeine Version zur Polynommultiplikation über dem Ring $\mathbb{Z}/N\mathbb{Z}$ formulieren, wollen wir noch einen Blick auf die Schranken werfen, die beachtet werden müssen. Wir wollen die Polynome $F(x)$ und $G(x)$ über dem Ring $\mathbb{Z}/N\mathbb{Z}$ multiplizieren. Sei $m = \mathrm{grad}(F) + \mathrm{grad}(G) + 1$. Dazu wählen wir verschiedene Primzahlen $p_i, i = 1, \ldots, k$, wobei $p_i \equiv 1 \bmod m$ gilt. Für das Produkt der p_i gilt folgende Schranke:

$$P = \prod_{i=1}^{k} p_i > mN^2/(1-\varepsilon).$$

Der Wert mN^2 entspricht dem größten möglichen Wert, der bei der Berechnung auftreten kann, ε ist die Präzision der floating-point-Arithmetik, die bei der Berechnung der Approximation eine Rolle spielt.

Wir wollen das Produkt

$$F(x) \cdot G(x) \quad \equiv \quad \sum_{j=0}^{m} h_j x^j \mod (x^m - 1)$$

bestimmen. Mit dem Algorithmus aus (3.1) können wir

$$F(x) \cdot G(x) \quad \equiv \quad \sum_{j=0}^{m} h_{ij} x^j \mod (p_i, x^m - 1)$$

für die verschiedenen Primzahlen p_i berechnen. Mit den Betrachtungen in obigem Beispiel erhalten wir nun als Formel für die verschiedenen Werte y_{ij}

$$y_{ij} \quad \equiv \quad \left(\frac{P}{p_i}\right)^{-1} h_{ij} \mod p_i. \tag{3.4}$$

Daraus berechnen wir einen Koeffizienten h_j des Polynoms $H(x)$ mittels

$$h_j \quad \equiv \quad \sum_{i=1}^{k} \left(\frac{P}{p_i}\right) y_{ij} - P \cdot \left\lfloor \frac{\varepsilon}{2} + \sum_{i=1}^{k} \frac{y_{ij}}{p_i} \right\rfloor . \tag{3.5}$$

3.2. Bemerkung

Man kann in einer Implementierung dieses Algorithmus die Werte $(P/p_i)^{-1} \bmod p_i$ aus (3.4) und $(P/p_i \bmod N)$ sowie $(-P \bmod N)$ aus (3.5) vorberechnen und abspeichern, weil diese Berechnungen bei einer Polynommultiplikation für jeden Koeffizienten benötigt werden. Wenn der Modul N und der Grad der Polynome $F(x)$ und $G(x)$ zu Beginn der Berechnungen bekannt ist, kann man diese Vorberechnungen am Anfang durchführen und die berechneten Ergebnisse für alle später auftauchenden Polynommultiplikationen verwenden.

3.3. Bemerkung
In unserer Implementierung beschränken wir uns auf 26-bit Primzahlen. Dadurch können wir alle Operationen bei den modularen Berechnungen mit einer single-precision-Arithmetik ausführen. Außerdem berechnen wir die 26-bit-Primzahlen p_i vor und bestimmen gleichzeitig die Wurzel w zu jeder Primzahl p_i.

3.3 Aufbau eines Polynoms aus den Nullstellen

Sei n eine Potenz von 2 und $a_i \in \mathbb{Z}/N\mathbb{Z}$ für $0 \leq i < n$. Wir wollen nun die Koeffizienten des Polynoms

$$F(x) = \prod_{i=0}^{n-1}(x - a_i) \tag{3.6}$$

in Standarddarstellung bestimmen. Als Notation wählen wir

$$F_{i,d}(x) = \prod_{j=0}^{d-1}(x - a_{i+j}) \tag{3.7}$$

für $\quad d \,|\, n \quad$ und $\quad 0 \leq i \leq n - d.$

Für $d = 1$ sind die Polynome $F_{i,1}(x) = (x - a_i)$ genau die Linearfaktoren des aufzubauenden Polynoms. Für $d > 1, d = 2^k$ ist das Polynom $F_{i,d}$ das Produkt der d Linearfaktoren zu den Nullstellen $a_i, a_{i+1}, a_{i+2}, \ldots, a_{i+d-1}$. d ist also jeweils der Grad des Polynoms $F_{i,d}$.

Wir können nun eine rekursive Formel angeben, die aus zwei Polynomen vom Grad d eine Polynom vom Grad $2 \cdot d$ zusammenbaut. Diese lautet:

$$F_{i,2d}(x) = F_{i,d}(x) \cdot F_{i+d,d}(x) \quad \text{für} \quad i = 0, 2d, \ldots, n - 2d.$$

Diese Formel kennzeichnet schon die Vorgehensweise im Algorithmus. Wir konstruieren aus den Linearfaktoren mit dieser Formel Polynome vom Grad 2, indem wir die Formel für alle möglichen Werte von i und den Wert $d = 1$ anwenden. Durch wiederholtes Anwenden der Formel für $d = 2, 4, \ldots, n/2$ erhält man nach $\log_2 n$ Schritten das komplette Polynom $F(x)$.

3.4. Algorithmus

Aufbau eines Polynoms aus seinen Nullstellen

Der folgende Algorithmus berechnet $F(x) = \prod_{i=0}^{n-1}(x - a_i)$ aus den Nullstellen $a_i \in \mathbb{Z}/N\mathbb{Z}$ für $0 \leq i < n$ für eine Zweierpotenz n und liefert das entstandene Polynom $F(x)$ zurück..

EINGABE: $a_i \in \mathbb{Z}/N\mathbb{Z}$ für $0 \leq i < n$
AUSGABE: $F(x) = \prod_{i=0}^{n-1}(x - a_i)$

```
for (i := 0; i< n; i++) do                  // Linearfaktoren
    F_{i,1}(x) := (x − a_i)
```

<u>BERECHNUNG</u>

```
for (j := 1; j≤ n/2; j := 2j) do
    for (i := 0; i≤ n − 2j; i := i + 2j)
        do
        F_{i,2j}(x) := F_{i,j}(x) · F_{i+j,j}(x)
return (F_{0,n})
```

3.5. Beispiel
Sei $\mathbb{K} = \mathbb{Z}/91\mathbb{Z}, n = 8, a_i = 2 \cdot i$ für $0 \leq i \leq 7$.
Wir berechnen die Koeffizienten von $F(x) = \prod_{i=0}^{7}(x - a_i)$. Den Aufbau der Polynome stellen wir als Baum dar, in jeder Stufe verdoppelt sich der Grad der Polynome, auf der untersten Ebene erhalten wir das Polynom $f(x)$.

Abbildung 3.1: Aufbau eines Polynoms aus seinen Nullstellen (1)

Im Algorithmus verwenden wir als Notation $F_{i,j}(x)$ für die einzelnen Polynome. Wir müssen aber nicht diese vielen verschiedenen Polynome allokieren und speichern. Die Polynome sind immer normiert. Somit müssen wir den Koeffizienten der höchsten Potenzen jeweils nicht speichern. Damit bleibt die Gesamtzahl der Koeffizienten auf jeder Stufe konstant. In der Realität genügt es also, ein Feld der Länge n zu allokieren und in diesem jeweils alle Koeffizienten abzuspeichern. Abbildung 3.2 zeigt den beschriebenen Aufbau.

Abbildung 3.2: Aufbau eines Polynoms aus seinen Nullstellen (2)

Dabei zeigen die gestrichelten Linien an, daß die davon getrennten Einträge zu einem Polynom gehören. In der obersten Zeile haben wir nach der Initialisierung 8 Polynome vom Grad 1. Daraus berechnen wir 4 Polynome vom Grad 2, z.B. $x^2 + 77x + 48$. Durch Fortführen des Algorithmus ergeben sich zwei Polynome vom Grad 4. In der letzten Zeile sehen wir dann das Ergebnis der Berechnung. Dieses Polynom hat den Grad 8, alle Einträge sind Koeffizienten des gleichen Polynoms.

3.4 Berechnung des ggT von Polynomen

Der Standardalgorithmus zur Berechnung des ggT von Polynomen ist der euklidische Algorithmus. Wir stellen hier eine Modifikation dieses Algorithmus vor. Dabei versuchen wir, die im euklidischen Algorithmus auftretenden Divisionen mit Rest durch Multiplikationen zu ersetzen. Diese können mit Hilfe der vorne beschriebenen FFT-Multiplikation schnell ausgeführt werden.

Wir geben nun eine rekursive Prozedur an. Die Bedingungen an die Eingabe sind

- U, V sind Polynome mit $\mathrm{grad}(U) > \mathrm{grad}(V)$.
- $0 \leq d_{red} \leq \lceil \mathrm{grad}(U)/2 \rceil$.

Beim Aufruf der im folgenden Algorithmus beschriebenen Funktion ist die Parameterübergabe „call by reference". Zur besseren Verständlichkeit haben wir die Parameter, deren Wert in der Funktion verändert wird, durch das tiefergestellte *out* gekennzeichnet. Damit erhalten wir als Ausgabe der Funktion

- Polynome U_{out} und V_{out}, für die gilt:
$$\mathrm{ggT}(U, V) \quad = \quad \mathrm{ggT}(U_{out}, V_{out}).$$
- $\mathrm{grad}(V_{out}) \leq \mathrm{grad}(U) - d_{red} \leq \mathrm{grad}(U_{out}) - 1$.
Dieser Aspekt zeigt den eigentlichen Nutzen der Funktion. Durch den Aufruf können wir den Grad des Polynoms V auf $\mathrm{grad}(U) - d_{red}$ drücken. Durch eine anschließende Reduktion $U_{out} \bmod V_{out}$ haben wir den Grad beider Polynome um mindestens d_{red} erniedrigt.
- eine Matrix M_{out}. M_{out} ist eine 2×2-Matrix. Die Einträge in dieser Matrix sind Polynome. Dabei erfüllt sie folgende Bedingung:
$$\begin{pmatrix} U_{out} \\ V_{out} \end{pmatrix} = M_{out} \cdot \begin{pmatrix} U \\ V \end{pmatrix}.$$
Wir betrachten die Matrix $M_{out} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$ und erhalten
$$\begin{aligned} U_{out} &= m_{11} \cdot U + m_{12} \cdot V \\ V_{out} &= m_{21} \cdot U + m_{22} \cdot V \end{aligned}$$
Diese Darstellung gleicht der Darstellung im erweiterten ggT-Algorithmus. Auf der linken Seite steht aber nicht der ggT von U und V, sondern ein Vielfaches des ggT. Darin zeigt sich die Wirkungsweise dieser Matrix M_{out}. Die Polynome $m_{11}, m_{12}, m_{21}, m_{22}$ stellen die Transformationen dar, die auf den Polynomen U und V ausgeführt werden. Wenden wir also den Algorithmus nur auf Teile der Ausgangspolynome an, so können wir mit Hilfe der Marix M_{out} diese Transformationen von den Teilen auf die kompletten Polynome übertragen.

Wir geben nun den Algorithmus an und erläutern danach seine Wirkungsweise. Ein Beweis der Korrektheit ist in [Mo92/2] ausgeführt.

3.6. Algorithmus

halfgcd

<u>$halfgcd(U, V, d_{red}, M_{out}, U_{out}, V_{out})$</u>

if $(deg(V) \leq deg(U) - d_{red})$ **then**

$\quad M_{out} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} U_{out} \\ V_{out} \end{pmatrix} := \begin{pmatrix} U \\ V \end{pmatrix}$

else

$\quad n := deg(U) - 2 \cdot d_{red} + 2$

$\quad d' := \lfloor (d_{red} + 1)/2 \rfloor$

$\quad halfgcd\left(\left\lfloor \frac{U}{x^n} \right\rfloor, \left\lfloor \frac{V}{x^n} \right\rfloor, d', M', U', V' \right)$

$\quad d'' := deg(V') - deg(U) + n + d_{red}$

$\quad$ **if** $(d'' \leq 0)$ **then**

$\quad\quad M_{out} := M'$

$\quad\quad \begin{pmatrix} U_{out} \\ V_{out} \end{pmatrix} := x^n \cdot \begin{pmatrix} U' \\ V' \end{pmatrix} + M_{out} \cdot \begin{pmatrix} U \bmod x^n \\ V \bmod x^n \end{pmatrix}$

$\quad$ **else**

$\quad\quad Q := \lfloor (U'/V') \rfloor$

$\quad\quad W' := U' - Q \cdot V'$

$\quad\quad halfgcd(V', W', d'', M'', V'', W'')$

$\quad\quad M_{out} := M'' \cdot \begin{pmatrix} 0 & 1 \\ 1 & Q \end{pmatrix} \cdot M'$

$\quad\quad \begin{pmatrix} U_{out} \\ V_{out} \end{pmatrix} := x^n \cdot \begin{pmatrix} V'' \\ W'' \end{pmatrix} + M_{out} \cdot \begin{pmatrix} U \bmod x^n \\ V \bmod x^n \end{pmatrix}$

$\quad$ **fi**

fi

Der Wert der Zahl d_{red} bestimmt bei jedem Aufruf, auf welchen Grad wir das Polynom V reduzieren wollen. Die Idee des Algorithmus ist dann eine Divide and Conquer Strategie. Wir berechnen den ggT der Polynome U und V, indem wir den ggT von Teilpolynomen $\left\lfloor \frac{U}{x^n} \right\rfloor$ und $\left\lfloor \frac{V}{x^n} \right\rfloor$ bestimmen. Die auf diesen Teilpolynomen durchgeführten Operationen merken wir uns in der Matrix M_{out}. Damit können wir das korrekte Ergebnis auf den kompletten Polynomen rekonstruieren. Dabei unterscheiden wir folgende Fälle:

- Wenn der Grad des Polynoms V genügend klein ist, müssen wir keine Berechnungen durchführen, wir geben die Eingabepolynome als Ergebnis zurück (Zeilen (1) - (2)).
- Im ersten rekursiven Aufruf betrachten wir nur noch die ersten $(\text{grad}(U) - n)$ bzw. $(\text{grad}(V) - n)$ Koeffizienten der Polynome. In der Matrix M_{out} liefert uns dieser Aufruf (Zeile (6)) alle Operationen, die wir auf diesen Teilpolynomen ausführen. Wenden wir diese Matrix dann auf den hinteren Teil der Polynome an, so erhalten wir Polynome U_{out}, V_{out}, die die oben angegebenen Bedingungen erfüllen.
- Mit dem ersten rekursiven Aufruf konnten wir die Teilpolynome aus dem letzten Fall nicht genügend reduzieren. Wir erhalten als Ergebnis U', V'. Nun schließen wir einen weiteren rekursiven Aufruf mit V' und $U' \bmod V'$ an. Dieser reduziert die Ausgangspolynome.

Wir sehen im Algorithmus, daß alle Operationen immer nur auf dem vorderen Teil der Polynome ausgeführt werden. Der hintere Teil geht nur bei der Komplettierung des Ergebnispolynoms ein. Dies ist immer beim Abschluß einer Rekursionsstufe und damit Rücksprung zur aufrufenden Prozedur der Fall. Bei dieser Komplementierung führen wir Polynommultiplikationen aus, die mit FFT sehr schnell sind. Durch diese Polynommultiplikationen sparen wir Reduktionen ein. Im Algorithmus führen wir nur eine Reduktion aus, dabei ist der Grad der Polynome immer kleiner als der Ausgangsgrad.

3.7. Bemerkung
Die vorgestellte Funktion *halfgcd* reduziert mit einem Aufruf den Grad des zweiten Polynoms V auf maximal $\text{grad}(U) - d_{red}$. Wir müssen diese Funktion also in eine aufrufende Funktion einbetten, die dann die Reduktion ausführt. Außerdem muß für jeden Aufruf der Parameter d_{red} bestimmt werden und überpüft werden, ob nicht schon der ggT berechnet ist. Dies leistet die im folgenden Algorithmus vorgestellte Steuerfunktion.

Dabei werden nur die Ausgangspolynome U, V und die berechneten Polynome U_{out}, V_{out} benötigt. Wir können also auf der obersten Aufrufebene der Funktion *halfgcd* die Berechnung der Matrix M_{out} einsparen. Dadurch sinkt zum einen der Speicherplatzbedarf, zum anderen müssen die Matrixmultiplikationen zum Berechnen der Matrix M_{out} nicht durchgeführt werden.

3.8. Algorithmus

ggT - Berechnung von Polynomen

EINGABE: Polynome U, V
AUSGABE: $\mathrm{ggT}(U, V)$

if $(\mathrm{grad}(U) > \mathrm{grad}(V))$ **then**
  $U' := U; \quad V' := V$
else
  $U' := V; \quad V' := U$
while $(V' \neq 0)$ **do**
  halfgcd $(U', V', \lceil \mathrm{grad}(U')/2 \rceil, M_{out}, U'', V'')$
  if $(V'' = 0)$ **then**
    $U' := U''; \quad V' := V''$
  else
    $U' := V''; \quad V' := U \bmod V''$
  fi
od
return (U')

Da wir in ECM Polynome über dem Ring $\mathbb{Z}/N\mathbb{Z}$ betrachten, wollen wir nun noch auf die Eigenheiten bei der ggT-Berechnung über $\mathbb{Z}/N\mathbb{Z}$ eingehen. Dazu führen wir zunächst das Beispiel 3.5 fort.

3.9. Beispiel

Sei wieder $N = 91, F(x) = x^8 + 19x^7 + 21x^5 + 7x^4 + 63x^3 + 20x^2 + 79x + 63$.
Als Polynom G wählen wir $G(x) = x^2 + x$ und wollen nun $\mathrm{ggT}(F(x), G(x))$ bestimmen. Dazu verwenden wir in diesem Beispiel den euklidischen Algorithmus und erhalten

$$F(x) = \left(x^6 + 18x^5 + 73x^4 + 39x^3 + 59x^2 + 4x + 16\right) \cdot G(x) + \left(63x + 63\right).$$

Damit gilt

$$\mathrm{ggT}\left(F(x), G(x)\right) = \mathrm{ggT}\left(x^2 + x,\ 63x + 63\right).$$

Um den euklidischen Algorithmus fortzusetzen, müssen wir nun $G(x) \bmod (63x + 63)$ bestimmen. Dazu versuchen wir, 63 modulo 91 zu invertieren, was nicht möglich ist. Die Invertierung liefert den Faktor $p = 7$ der Zahl $N = 91$.

Wir betrachten nun die ggT-Berechnung in den Körpern $\mathbb{Z}/7\mathbb{Z}$ und $\mathbb{Z}/13\mathbb{Z}$.

$$\begin{aligned}\mathrm{ggT}\left(F(x),(G(x)\right) &= x^2+x = x\cdot(x+1) \quad \text{in } \left(\mathbb{Z}/_{7\mathbb{Z}}\right)\\ \mathrm{ggT}\left(F(x),(G(x)\right) &= 11\cdot(x+1) \quad \text{in } \left(\mathbb{Z}/_{13\mathbb{Z}}\right)\end{aligned}$$

Der Grad des ggT ist also einmal 2, beim zweiten Teiler 1. Nun können wir keine normiertes Polynom in $\mathbb{Z}/91\mathbb{Z}$ konstruieren, das beide Bedingungen erfüllt. Die Berechnung in $\mathbb{Z}/91\mathbb{Z}$ schlägt fehl.

Mit Hilfe der ggT-Berechnung über dem Ring $\mathbb{Z}/N\mathbb{Z}$ können wir zwei Polynome auf gemeinsame Nullstellen modulo eines Teilers p der Zahl N testen. Falls solch eine gemeinsame Nullstelle existiert, schlägt im Ablauf des Algorithmus eine Invertierung fehl, wir finden den Faktor p. Falls der Algorithmus terminiert, erhalten wir das korrekte Ergebnis modulo N, also auch modulo jeden Teilers von N (Beweis siehe [Mo92/2]).

Kapitel 4

Die FFT-Continuation

In diesem Kapitel zeigen wir, wie man die schnellen Algorithmen für die Polynomarithmetik zum Faktorisieren mit ECM nutzen kann. Die Idee basiert auf dem Geburtstags-Paradoxon, das jetzt vorgestellt wird.

4.1 Die Geburtstags-Paradoxon-Continuation

Beim Geburtstags-Paradoxon dreht sich die Frage darum, wieviele zufällig ausgewählte Personen zusammentreffen müssen, so daß mit Wahrscheinlichkeit größer als $\frac{1}{2}$ mindestens zwei am gleichen Tag Geburtstag haben.
Die Lösung des Problems erhalten wir, wenn wir die Ungleichung

$$\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)\cdot\left(1-\frac{3}{365}\right)\cdots\left(1-\frac{n-1}{365}\right)<\frac{1}{2}$$

nach n auflösen. Dann erhalten wir, daß die Gleichung für alle $n > 23$ gilt.

Wir betrachten nun dieses Phänomen etwas allgemeiner. Wieviele Zahlen muß man zufällig wählen, damit mit Wahrscheinlichkeit größer als $\frac{1}{2}$ mindestens zwei kongruent modulo einer fest vorgegebenen Zahl p sind? Die Gleichung

$$\left(1-\frac{1}{p}\right)\cdot\left(1-\frac{2}{p}\right)\cdot\left(1-\frac{3}{p}\right)\cdots\left(1-\frac{q-1}{p}\right)<\frac{1}{2}$$

liefert uns die Lösung

$$\Rightarrow q \approx \sqrt{2\ln 2}\cdot\sqrt{p} \;\approx\; 1.18\sqrt{p} \quad \text{(siehe [Ri85]).} \tag{4.1}$$

Auf diesem Umstand basiert **Pollards ρ - Methode**, die wir nun vorstellen. Danach übertragen wir diese Idee auf ECM.

4.1.1 Pollards ρ - Methode

Aus der obigen Betrachtung ist ersichtlich, daß wir nur $O(\sqrt{s})$ zufällige Zahlen unter einer festen Schranke s wählen müssen, um mit Wahrscheinlichkeit größer als $\frac{1}{2}$ ein Paar x_1, x_2 zu finden, so daß gilt

$$x_1 \equiv x_2 \mod s \iff (x_1 - x_2) \mid s.$$

Auf dieser Eigenschaft beruht die Pollard ρ - Methode. Wir generieren uns eine Folge zufälliger Zahlen $x_i \in \mathbb{Z}/N\mathbb{Z}$ suchen Indizes $i \neq j$, so daß gilt:

$$x_i \equiv x_j \mod p \quad \text{und} \quad x_i \not\equiv x_j \mod N.$$

Dann liefert uns die Berechnung $\mathrm{ggT}(x_i - x_j, N)$ einen Faktor $\geq p$. Bei einer deterministischen Konstruktion der Folgenglieder ist dies gleichbedeutend mit dem Finden eines Zyklus in der Folge. Dazu gibt es mehrere Varianten. Wir können alle x_i abspeichern und dann jeweils ein neues Folgenglied mit allen vorher berechneten Folgengliedern vergleichen. Bei langen Periodenlängen haben wir einen immensen Speicherplatzbedarf und müssen zum Testen viele Multiplikationen mit Reduktion bzw. ggT-Berechnungen durchführen. Hier bietet sich eine andere Lösung an. Bei Floyds Zyklus-Finde-Algorithmus nutzt man die Tatsache aus, daß man die Periode einer Folge finden kann, indem man jeweils nur das i-te und das $2i$-te Folgenglied vergleicht (siehe [Ri85]). Wir berechnen also parallel x_i und x_{2i} und führen den Test aus.

4.1. Algorithmus

Pollards ρ - Algorithmus

Wir verwenden als Übergangsfunktion $g(x) = x^2 + a$.

EINGABE: zu faktorisierende Zahl N, Startwert x_0, a
AUSGABE: Faktor p

<u>INITIALISIERUNG</u>

(1) $x := x_0, \quad y := x_0$

<u>SCHLEIFE</u>

(2) **repeat**
(3) $\quad x := x^2 + a \mod N$
(4) $\quad y := y^4 + 2y^2a + a^2 + a \mod N \qquad // y := y^2 + a, y := y^2 + a$
(5) **until** $(\mathrm{ggT}(x - y, N) > 1)$
(6) **return** $(\mathrm{ggT}(x - y, N))$

4.2. Bemerkung

- Es ist möglich, bei der ggT-Berechnung als Faktor die Zahl N selbst zu finden. Wir haben also Paar x_1, x_2 gefunden, das schon in $\mathbb{Z}/N\mathbb{Z}$ kongruent ist. In diesem Fall kann man die Berechnung für einen anderen Wert von a noch einmal durchführen.

- Falls die Primteiler von N sehr groß sind, erhalten wir eine große Laufzeit. Deshalb fügen wir eine zweite Bedingung zum Abbruch des Pollard ρ - Algorithmus hinzu, nämlich die maximale Anzahl von Versuchen, die durchgeführt werden soll. Falls danach kein Faktor gefunden wurde, können wir mit anderen Algorithmen versuchen, die Faktoren zu finden.

4.3. Beispiel
Sei $N = 31613 = 101 \cdot 313$, $a = 1$. Wir verwenden also die Übergangsfunktion $g(x) = x^2 + 1$, als Startwert x_0 wählen wir 5.

i	x_i	y_i	$\mathrm{ggT}(x_i - y_i, N)$
0	5	5	–
1	26	677	1
2	677	27 133	1
3	15 748	26 820	1
4	27 133	19 308	313

Diesen Algorithmus wollen wir nun auf ECM übertragen. Wir beschreiben nun einen simplen Ansatz für die Geburtstags-Paradoxon-Continuation.

4.1.2 Einfacher Algorithmus

Die Situation ist die gleiche wie in Kapitel 1.2.2. Sei

$$N \quad = \quad p_1 \cdot p_2 \cdot \dots \cdot p_r, \quad p_i \in \mathbb{P}_{>3} \quad \forall\, i = 1, \dots, r$$

die zu faktorisierende Zahl. Wir betrachten nun einen Primteiler $p := p_i$, für den gilt:

$$\left| E\left(\mathbb{Z}/p\mathbb{Z} \right) \right| \quad = \quad q_1^{e_1} \cdot q_2^{e_2} \cdot \dots \cdot q_{l-1}^{e_{l-1}} \cdot q_l^{e_l}, \quad q_i \in \mathbb{P},\ e_i \in \mathbb{N} \quad \forall\, i = 1, \dots, l \tag{4.2}$$

mit $\quad q_1 < q_2 < \dots < q_{l-1} < q_l$.

In der 1. Phase haben wir

$$Q \quad = \quad k \cdot P \quad \text{mit} \quad k = \prod_{\mathbb{P} \ni q \leq B} q^{e_q}, \quad e_q = \max\{j \in \mathbb{N} : q^j \leq C\}$$

berechnet, den Faktor p aber nicht gefunden. Wiederum betrachten wir den Fall, daß $q_l > B$ ist. Zusätzlich soll gelten:

$$q_l \cdot \phi_p(Q) \quad = \quad \mathcal{O}_p \qquad \text{in} \quad E\left(\mathbb{Z}/_{p\mathbb{Z}}\right).$$

Wir arbeiten nun in der von Q erzeugten Untergruppe $\langle Q \rangle$ von $E(\mathbb{Z}/N\mathbb{Z})$. In dieser Untergruppe bestimmen wir zufällige Vielfache des Punktes Q. Dabei hoffen wir, daß zwei dieser Vielfachen auf der Kurve $E(\mathbb{Z}/p\mathbb{Z})$ gleich sind. Wir haben also wieder die Situation des Geburtstags-Paradoxons. Falls wir genügend viele zufällige Vielfache berechnen, so sollten wir mit Anwendung des Geburtstags-Paradoxons einen Faktor $\geq p$ finden.
Formal erhalten wir folgenden Algorithmus:

4.4. Algorithmus

Geburtstags-Paradoxon-Continuation

EINGABE: Punkt $Q = k \cdot P$ als Ergebnis der 1. Phase,
Anzahl der gewünschten zufälligen Vielfachen d

AUSGABE: Faktor p der zu faktorisierenden Zahl N

(1) Berechne d zufällige Vielfache Q_i des Punktes Q. Die Methoden dafür stellen wir in Abschnitt 4.4 vor.

(2) Finde zwei Indizes $i \neq j$ mit $Q_i = Q_j$ in $E(\mathbb{Z}/p\mathbb{Z})$. Nach (4.1) benötigt man dazu $O(\sqrt{q_l})$ Schritte.

(3) Berechne $f :=\ \mathrm{ggT}(x[Q_i] - x[Q_j], N)$

(4) **return** (Faktor f)

Dies ist ein probabilistischer Algorithmus. Wir müssen nicht notwendigerweise einen Faktor finden, falls auch die am Anfang dieses Abschnitts vorgestellte Situation gegeben ist und wir ausreichend viele Vielfache von Q berechnen. Falls z.B. „schlechte“ Vielfache generiert werden, finden wir möglicherweise keinen Faktor. Neben dieser Einschränkung ist es natürlich unwahrscheinlich, daß überhaupt diese Situation auftritt, so daß meistens die Zahl 1 als Faktor gefunden wird.

Falls zwei Indizes $i \neq j$ mit obiger Bedingung in der Folge auftreten, so liefert die ggT-Berechnung in Schritt (3) einen Faktor $\geq p$. In diesem Algorithmus müssen wir noch die einzelnen Schritte präzisieren, z.B. wie finden wir Indizes $i \neq j$ mit $Q_i = Q_j$ in $E(\mathbb{Z}/p\mathbb{Z})$. Dies wollen wir in den folgenden Abschnitten tun.

4.1.3 Praktischer Ansatz

Wir betrachten wir zwei beliebige verschiedene Zahlen $s_1 \neq s_2$. Wenn

$$\begin{array}{lrcl}
 & s_1 & \equiv & s_2 \mod q_l \\
\Rightarrow & s_1 & = & s_2 + m \cdot q_l\ , \quad m \in \mathbb{Z} \\
\Rightarrow & s_1 - s_2 & = & m \cdot q_l\ , \quad m \in \mathbb{Z} \\
\Rightarrow & (s_1 - s_2) \cdot Q & = & m \cdot q_l \cdot Q \\
\Rightarrow & s_1 \cdot Q - s_2 \cdot Q & = & \mathcal{O}_p \quad \text{in} \quad E\,(\mathbb{Z}/p\mathbb{Z}) \\
\Rightarrow & x[s_1 \cdot Q] - x[s_2 \cdot Q] & \equiv & 0 \mod p \\
\Rightarrow & \multicolumn{2}{r}{\mathrm{ggT}(x[s_1 \cdot Q] - x[s_2 \cdot Q],\ N) \quad \geq} & p.
\end{array}$$

Bisher haben wir immer eine Folge berechnet und gleiche Punkte $Q_i = Q_j$ in dieser Folge gesucht. Dies ist nicht notwendig. Wir können auch zwei Folgen Q_i und $\tilde{Q}_j$ generieren und dann die Folgenglieder der einen Folge mit den Folgengliedern der anderen Folge vergleichen. Für die Anwendung des Geburtstags-Paradoxons spielt dies keine Rolle. Entscheidend ist nur die Anzahl der möglichen Vergleiche von Punkten, die wir durchführen. Wir betrachten nun zwei Folgen

$$(Q_i) \quad \text{für } i = 1, \ldots, d_1 \qquad \text{und} \qquad (\tilde{Q}_j) \quad \text{für } j = 1, \ldots, d_2. \tag{4.3}$$

Dabei entspricht x_i der x-Koordinate des i-ten Punktes der ersten Folge, $\tilde{x}_j$ analog. In einem einfachen Ansatz berechnet man

$$d = \prod_{i=1}^{d_1} \prod_{j=1}^{d_2} (x_i - \tilde{x}_j) \mod N$$

und danach $\mathrm{ggT}(d, N)$. Diese Methode hat den Nachteil, daß zum einen $(d_1 + d_2)$ x-Koordinaten abgespeichert werden müssen. Damit haben wir einen großen Speicherplatzbedarf. Zur Berechnung von d benötigen wir $(d_1 \cdot d_2)$ Multiplikationen mit Reduktion, was diesen Ansatz deutlich langsamer als die verbesserte Standard-Continuation macht.

Brent schlägt einen anderen Ansatz vor. Wir betrachten wieder zwei Folgen wie in (4.3), $d_1 < d_2$. Aus den x-Koordinaten der ersten Folge baut man ein Polynom über dem Ring $\mathbb{Z}/N\mathbb{Z}$ auf, d.h. wir berechnen

$$f(x) = \prod_{i=1}^{d_1} (x - x_i).$$

Dieses Polynom f werten wir nun an den Stellen $\tilde{x}_j, j = 1, \ldots, d_2$ aus. Dabei entspricht eine Auswertung

$$f(\tilde{x}_j) = \prod_{i=1}^{d_1} (\tilde{x}_j - x_i) \mod N.$$

Damit berechnen wir also das Produkt der Differenzen $(\tilde{x}_j - x_i)$ für ein festes j und alle $i = 1, \ldots, d_1$. Für diese Auswertung benötigt man $d_1/2 + \log_2 d_1$ Multiplikationen mit Reduktion und einige Additionen (siehe [BoLe93]) — eine deutliche Verbesserung gegenüber d_1 Multiplikation im ersten Ansatz. Der Speicherplatzbedarf sinkt auf d_1 Einträge im Polynom und eine konstante Anzahl, die bei der Berechnung der Vielfachen der zweiten Folge benötigt werden.

Im folgenden Algorithmus haben wir als Länge der Folgen $d_1 < d_2$, wobei $d_1 \cdot d_2 \approx D$ für eine feste Schranke D ist. D ist abhängig von der Größe der Faktoren, die wir finden wollen (vergleiche Tabelle 2.2).
Zur Berechnung der Vielfachen $Q_{i+1} = g(i) \cdot Q_i$ bzw. $\tilde{Q}_{j+1} = \tilde{g}(j) \cdot \tilde{Q}_j$ nutzen wir zwei hier nicht näher spezifizierte verschiedene Funktionen. Mögliche Funktionen stellen wir in Abschnitt 4.4 vor.

4.5. Algorithmus

Brent's Geburtstags-Paradoxon-Continuation

Eingabe: elliptische Kurve E, Punkt $Q = k \cdot P$, Schranken d_1, d_2
Ausgabe: Faktor von N, 1 möglich

(1) $d := 1$

Berechnung von $f(x)$

(2) **for** $(i := 1; i \leq d_1; i++)$ **do**
(3) $\quad x_i := x[g(i) \cdot Q]$
(4) **od**
(5) $f(x) := \prod_{i=1}^{d_1} (x - x_i) \bmod N$

Berechnung und Auswertung der zweiten Folge

(6) **for** $(j := 1; j \leq d_2; j++)$ **do**
(7) $\quad \tilde{x} := x[\tilde{g}(j) \cdot Q]$
(8) $\quad c := f(\tilde{x}) \bmod N$
(9) $\quad d := c \cdot d \bmod N$
(10) **od**

ggT-Berechnung

(11) $d := \mathrm{ggT}(d, N)$
(12) **return** (d)

Bei dieser Variante können wir schon die schnelle Polynomarithmetik nutzen, z.B. beim Aufbau des Polynoms aus seinen Nullstellen. Wir arbeiten hier mit einem Polynom, welches an vielen, verschiedenen Stellen ausgewertet wird. Dies legt die Idee nahe, auch die x-Koordinaten der zweiten Folge in einem Polynom zu entwickeln und dann eine Auswertung eines Polynoms an einem Polynom durchzuführen. Damit erhalten wir einen Ansatz für die FFT-Continuation.

4.2 Die FFT - Continuation

Sei N wie in Abschnitt 4.1.2. Wir haben in der ersten Phase $Q = k \cdot P$ berechnet, aber keinen Faktor von N gefunden. Wir wählen nun zwei Mengen

$$M := \{m_i : i = 1, \ldots, d_1\} \quad \text{und} \quad N := \{n_j : j = 1, \ldots, d_2\},$$

die disjunkt sind und berechnen damit die Mengen

$$\begin{aligned} A &:= \{a_i := x[m_i \cdot Q] \mid i = 1, \ldots, d_1\}, \\ B &:= \{b_j := x[n_j \cdot Q] \mid j = 1, \ldots, d_2\}. \end{aligned}$$

Nun haben wir die Situation der Geburtstags-Paradoxon-Continuation. Wir besitzen zwei Mengen von x-Koordinaten von zufälligen Vielfachen von Q und wollen prüfen, ob Indizes i_0, j_0 existieren, so daß gilt:

$$a_{i_0} \equiv b_{j_0} \bmod p.$$

Dies ist genau dann der Fall, falls schon für die zufälligen Skalare der Mengen M und N gilt:

$$m_{i_0} \equiv n_{j_0} \bmod q_l.$$

Wir verwenden die Algorithmen aus Kapitel 3 und konstruieren aus beiden Mengen die Polynome

$$F(X) = \prod_{i=1}^{d_1} (X - a_i) \quad \text{und} \quad G(X) = \prod_{j=1}^{d_2} (X - b_j). \tag{4.4}$$

Um gemeinsame Nullstellen modulo p zu finden, gibt es zwei verschiedene Algorithmen. Zum einen können wir das Polynom $F(X)$ an $G(X)$ auswerten [Mo92/2]. Diesen Algorithmus stellen wir hier nicht vor, weil zur effizienten Ausführung sehr viel Speicher benötigt wird. Eine zweite Lösung bietet eine ggT-Berechnung von $F(X)$ und $G(X)$. In Abschnitt 3.4 haben wir gesehen, daß bei der Berechnung von $\mathrm{ggT}(F(X), G(X)) \bmod N$ ein Fehler auftritt, falls gemeinsame Nullstellen modulo eines echten Teilers von N existieren. Damit erhält man den folgenden Algorithmus:

4.6. Algorithmus

FFT-Continuation

Als Funktion zur Generierung der zufälligen Vielfachen verwenden wir die Funktionen g bzw. $\tilde{g}$.

Eingabe: Punkt $Q = k \cdot P$, Schranken d_1, d_2
Ausgabe: Faktor der Zahl N

Berechnung des Polynoms F

(1) **for** $(i := 1; i \leq d_1; i++)$ **do**
(2) $a_i := x[g(i) \cdot Q]$
(3) $F(x) := \prod_{i=1}^{d_1}(x - a_i)$

Berechnung des Polynoms G

(4) **for** $(j := 1; j \leq d_2; j++)$ **do**
(5) $b_j := x[\tilde{g}(j) \cdot Q]$
(6) $G(x) := \prod_{j=1}^{d_2}(x - b_j)$

ggT-Berechnung

(7) **if** (Berechnung ggT $\big(F(x), G(x)\big)$ mod N ausführbar) **then**
(8) **return** (1)
(9) **else**
(10) **return** (Faktor(bei ggT-Berechnung gefunden))

Bei diesem Algorithmus gibt es zwei Stellen, wo wir einen Faktor finden können. Zum einen kann die Multiplikation des Punktes Q mit dem Skalar $g(i)$ bzw. $\tilde{g}(j)$ fehlschlagen. Dies ist genau dann der Fall, falls gilt: $q_l \mid g(i)$ bzw. $q_l \mid \tilde{g}(j)$.
Den zweiten Fall zum Finden eines Faktors haben wir schon in Kapitel 3.4 ausgeführt. Falls eine gemeinsame Nullstelle modulo eines Faktors p von N existiert, so können wir die ggT-Berechnung nicht komplett durchführen. Eine Invertierung schlägt fehl, wir finden einen Faktor $\geq p$.

Diese beiden Eigenschaften spielen natürlich auch eine bedeutende Rolle bei der Generierung der Vielfachen in der Continuation. Wir versuchen bei der Generierung beide Möglichkeiten auszuschöpfen. Bevor wir auf die Generierung der zufälligen Vielfachen eingehen, wollen wir uns noch mit der Bestimmung der Parameter B, d_1, d_2 beschäftigen.

4.3 Die Wahl der Parameter

Bei der Bestimmung der Parameter d_1, d_2 gibt es gewisse Einschränkungen, die wir beachten müssen.

- Die Anzahl der Punktvielfachen, die berechnet werden, ist d_1+d_2. Gleichzeitig erhalten wir $d_1 \cdot d_2$ Möglichkeiten für einen Treffer.
- Um die Anzahl der Punktvielfachen zu minimieren, muß man $d_1 = d_2$ wählen. Dies hat den Nachteil, daß die Polynome sehr groß werden. Um den Speicherplatzbedarf in Grenzen zu halten, wählen wir also $d_1 < d_2$.
- Um die FFT-Multiplikation bei Polynomen optimal anwenden zu können, sollten die Parameter d_1, d_2 jeweils Zweier-Potenzen sein.

Diese letzte Eigenschaft, daß die Schanken d_1 und d_2 jeweils Zweier-Potenzen sein sollen, ist bei der Berechnung der Parameter nicht einzuhalten. Wir haben in Abschnitt 3.1 beschrieben, was man für diesen Fall bei der Polynommultiplikation mit FFT tun muß.
Bei verschiedenen asymptotischen Analysen von ECM (siehe [Br86], [Mo87]) hat sich herausgestellt, daß die Parameter nahezu ideal sind, wenn die Laufzeiten von 1. Phase und Continuation etwa im Verhältnis 2:1 bis 3:2 stehen. Diese Tatsache hat sich auch bei der Berechnung der Parameter für die ECM-Version mit verbesserter Standard-Continuation und bei vielen Tests bestätigt.
Peter L. Montgomery hat in seiner Arbeit [Mo92/2] und bei persönlicher Kommunikation eine ausführliche Beschreibung zur Berechnung der Parameter angegeben. Wir haben unsere Implementierung mit diesen berechneten Parametern getestet und gesehen, daß unsere Ergebnisse das oben angegebene Verhältnis erfüllen und nah an den von Peter L. Montgomery erreichten Ergebnissen liegen. Aus diesem Grund haben wir seine Parameter übernommen. In der folgenden Tabelle geben wir Parameter für verschiedene Faktorlängen an:

#Stellen	B	d_1	d_2	#Kurven
20	18 000	160	1 000	51
25	88 000	540	3 700	120
30	390 000	1 700	12 000	280
33	880 000	3 300	24 000	450
35	1 500 000	5 000	38 000	610
38	3 200 000	9 200	72 000	960
40	5 300 000	14 000	110 000	1 300

Tabelle 4.1: optimale Parameter für ECM mit FFT-Continuation

In Tabelle 6.4 geben wir an, welche Laufzeiten und welchen Speicherplatzbedarf wir bei den in ECM benötigten Operationen mit Polynomen haben. Dabei wächst der Speicherplatz bei steigendem Polynomgrad sehr stark an. Deshalb sind wir bei unserer verteilten Version (siehe Abschnitt 5.3.3) gezwungen, mit einem geringeren Wert für d_1 zu rechnen. Um die gleiche Wahrscheinlichkeit zum Finden eines Faktors zu erhalten, müssen wir den Parameter d_2 entsprechend erhöhen.

Wie wir gesehen haben, wählen wir die Parameter $d_1 \ll d_2$. Dadurch bietet sich die Möglichkeit, das Polynom $G(x)$ in Teilpolynomen vom Grad d_1 aufzubauen und die Zwischenergebnisse jeweils zu reduzieren. Im folgenden Algorithmus verwenden wir zum Aufbau der Einzelpolynome Algorithmus 3.4. Damit erhalten wir zum Aufbau der Polynome $F(x)$ und $G(x)$ folgenden Algorithmus.

4.7. Algorithmus

Der folgende Algorithmus berechnet $F(x) = \prod_{i=1}^{d_1}(x - a_i)$ und $G(x) = \prod_{j=1}^{d_2}(x - b_j) \bmod F(x)$

EINGABE: $a_i \in \mathbb{Z}/N\mathbb{Z}$ für $1 \leq i \leq d_1$
$b_j \in \mathbb{Z}/N\mathbb{Z}$ für $1 \leq j \leq d_2$
AUSGABE: $F(x), G(x)$

<u>INITIALISIERUNG</u>

(1) $F(x) := \prod_{i=1}^{d_1}(x - a_i)$
(2) $G(x) := 1$

<u>BERECHNUNG</u>

(3) **for** $(j := 1; j \leq \frac{d_2}{d_1}; j++)$ **do**
(4) $\quad H(x) := \prod_{i=jd_1+1}^{jd_1}(x - b_i)$
(5) $\quad G(x) := G(x)H(x) \bmod F(x)$
(6) **od**
(7) **return** $(F(x), G(x))$

4.4 Generierung der Vielfachen

In diesem Abschnitt beschäftigen wir uns mit der Generierung der Punktvielfachen in der FFT-Continuation. Dabei stellen wir mehrere Anforderungen vor, die wir an die Vielfachen stellen und geben mögliche Lösungen an. In der ersten Phase von ECM haben wir $Q = k \cdot P$ berechnet, die Ordnung des Punktes Q sei nun $ord_{E(\mathbb{Z}/p\mathbb{Z})}(Q) = q_l, q_l \in \mathbb{P}$. Da q_l in der Regel nicht bekannt ist, führen wir die folgenden Betrachtungen für eine beliebige Primzahl q durch.

4.4.1 Anforderungen

Zur Anwendung des Geburtstags-Paradoxons wird gefordert, daß man $\sqrt{q}$ viele zufällige Vielfache von Q berechnen muß, um mit Wahrscheinlichkeit größer $\frac{1}{2}$ einen Teiler von N zu finden. Für die Berechnung dieser zufälligen Vielfachen benötigen wir also $\sqrt{q}$ Punktmultiplikationen, was gegenüber der Standard-Continuation und erst recht gegenüber der verbesserten Standard-Continuation einen deutlichen Effizienzverlust darstellt. In der Praxis hat sich herausgestellt, daß auch eine Generierung mit pseudozufälligen Vielfachen genügend zufällig ist, so daß wir das Geburtstags-Paradoxon anwenden dürfen.

Wir stellen nun Anforderungen an die Mengen $M = \{m_i : i = 1, \ldots, d_1\}$ und $N = \{n_j : j = 1, \ldots, d_2\}$.

1. Die meisten kleinen ($< 10^8$ für 33-stellige Faktoren) Primzahlen sollen $m_i \pm n_j$ teilen. Es genügt auch, wenn die Primzahl selbst m_i oder n_j teilt. In diesem Fall finden wir den Faktor schon bei der Punktmultiplikation $m_i \cdot Q$ bzw. $n_j \cdot Q$.

   Wenn wir sicherstellen, daß alle Primzahlen $q < d_1 d_2$ eine Summe, Differenz bzw. ein Vielfaches selbst teilen, können wir deterministisch alle Vielfachen bis $D = d_1 d_2$ abchecken und haben zusätzlich die Möglichkeit, einige größere Primzahlen zu testen.

2. Die Summen und Differenzen $d = m_i \pm n_j$ sollen viele Primteiler haben, aber kein d sollte gleich 0 sein. Außerdem sollen die m_i, n_j selbst ungleich 0 sein, wobei auch die Mengen keine doppelten Elemente $n_{j_0} = n_{j_1}$ enthalten sollen.

3. Die durchschnittliche Zeit zur Berechnung von $m_i \cdot Q$, bzw. $n_j \cdot Q$ soll 100 Multiplikationen mit Reduktion modulo N nicht überschreiten.
   Diese Schranke haben wir durch experimentelle Versuche bestätigt. Falls mehr Multiplikationen ausgeführt werden müssen, steigt der Zeitbedarf für die FFT-Continuation so stark an, daß der Vorteil durch die schnellen Berechnungen auf Polynomen verloren geht.

4. Um alle $d_1 d_2$ Möglichkeiten einer Kongruenz $m_i \equiv n_j \bmod q$ zu erhalten, ist es notwendig, daß alle n_j modulo q verschieden sind.

5. Wir versuchen, keine Paare $m_{i_1} \pm n_{j_1}$ und $m_{i_2} \pm n_{j_2}$ zu erhalten, die große Primfaktoren haben oder die sich gegenseitig teilen.

Um diese Anforderung zu erfüllen, benutzen wir die in Definition A.11 definierten Dickson Polynome. Eine ausführliche Beschreibung findet man in [Mo92/2].

Wir stellen hier nun zwei spezielle Dickson-Polynome vor, die wir auch bei der Implementierung verwendet haben. Dabei wollen wir die Mengen $M = \{m_i : i = 1, \ldots, d_1\}$ und $N = \{n_j : j = 1, \ldots, d_2\}$ bestimmen. Also geben wir nun Berechnungsvorschriften für die m_i bzw. n_j an.

4.4.2 Vielfache der Form $2^{k/2}$ und 3^k

Wir wählen für eine durch 4 teilbare Zahl k

$$\begin{array}{rcll} m_i & = & 3^{ik} & \text{für} \quad 1 \leq i \leq d_1, \\ n_j & = & 3^{(d_1/2)k} \cdot 2^{jk/2} & \text{für} \quad 1 \leq j \leq d_2. \end{array}$$

Zur Berechnung der Punktvielfachen $m_i \cdot Q$ benötigen wir dann in jedem Schritt k Punktadditionen und k Punktverdoppelungen. Damit sind insgesamt $2k(d_1 - 1)$ Operationen auf Punkten erforderlich.
Die Berechnung der n_j verbraucht in jedem Schritt nur $k/2$ Punktverdopplungen, so daß wir als Gesamtaufwand $2k(d_1 - 1) + d_2 k/2$ erhalten.

Mit unserer speziellen Wahl von $d_1 \ll d_2$ erhalten wir als mittleren Aufwand zur Berechnung eines Vielfachen den Wert $k/2$.

4.4.3 Vielfache aus arithmetischen Progressionen

Wir wollen die x-Koordinaten von Vielfachen $P(i) \cdot Q$ bestimmen. Dabei ist $P(x)$ eine Polynomfunktion vom Grad k, i durchläuft eine Folge in $\mathbb{N}$. Bevor wir einen Algorithmus zum effizienten Berechnen dieser Vielfachen angeben, wollen wir uns dies am Beispiel anschauen.

4.8. Beispiel
Wir wollen die Funktion $P(x) = x^4$ an den Stellen $M_i = 2 \cdot i + 1$ für $i \in \mathbb{N}$ auswerten. Dazu berechnen wir die ersten Werte und bestimmen damit die Differenzen. In Abbildung 4.1 stehen in der obersten Zeile die berechneten Werte $P(M_1), P(M_2), \ldots$.

Wir sehen, daß der Wert in der letzten Zeile konstant ist. Dies folgt daraus, daß der Grad des Polynoms P genau 4 ist. Dabei erkennen wir schnell die Regelmäßigkeit. Der Wert 625 ergibt sich z.B. als Summe von 81 und 544. Nutzen wir diesen Aspekt sukzessive aus, so können wir die Diagonale $[625, 1776, 2384, 1536, 384]$ aus der Diagonalen $[81, 544, 1232, 1152, 384]$ berechnen, indem wir jeweils zwei benachbarte Einträge addieren.

Wir kehren nun zum ursprünglichen Problem zurück. Wir wollen $x[m_i \cdot Q]$ bestimmen, wobei $m_i = P(M_i)$ ist. Dabei ist P ein Polynom, M_i eine Folge in $\mathbb{N}$.
Dazu setzen wir das Beispiel fort. Wir erkennen, daß wir den Punkt $625 \cdot Q$ aus den Punkten $81 \cdot Q$ und $544 \cdot Q$ durch eine Punktaddition bestimmen können. Wir erhalten also die in Abbildung 4.2 dargestellten Abhängigkeiten.

1		81		625		2401		6561		14641
	80		544		1776		4160		8080	
		464		1232		2384		3920		
			768		1152		1536			
				384		384				

Abbildung 4.1: Endliche Differenzen der Polynomfunktion $P(x) = x^4$

$$
\begin{tikzcd}
81 \cdot Q \arrow[d] & 544 \cdot Q \arrow[d] \arrow[dl] & 1232 \cdot Q \arrow[d] \arrow[dl] & 1152 \cdot Q \arrow[d] \arrow[dl] & 384 \cdot Q \arrow[d] \arrow[dl] \\
625 \cdot Q & 1776 \cdot Q & 2384 \cdot Q & 1536 \cdot Q & 384 \cdot Q
\end{tikzcd}
$$

Abbildung 4.2: Abhängigkeiten bei Progressionen

Nach einer Initialisierungsphase, in der wir die Punktvielfachen auf der ersten Diagonale berechnen, können wir also mit k Punktadditionen den nächsten gewünschten Punkt bestimmen.

Im folgenden Algorithmus beschreiben wir die Generierung einer Folge. Dabei berechnen wir sukzessive die x-Koordinaten von $P(h_i) \cdot Q$ für $i = 1, \ldots, d$. Dazu verwenden wir als Funktion $P(x) = x^k$. Die Werte h_i stellen wiederum eine Folge in $\mathbb{N}$ dar. Diese spezifizieren wir im Algorithmus nicht näher. In unserer Implementierung verwenden wir

$$
\begin{array}{rclcl}
m_i = P(M_i) & \text{mit} & M_i = 6i + 1 & \text{für} & 1 \leq i \leq d_1, \\
n_j = P(N_j) & \text{mit} & N_j = 6d_1(j + 1) & \text{für} & 1 \leq j \leq d_2.
\end{array}
$$

Die spezielle Gestalt hat aber nur einen Einfluß auf die Initialisierungsphase, wenn wir die ersten Folgenglieder berechnen müssen. Die Laufzeit des Algorithmus wird von den Operationen der Hauptschleife bestimmt.

4.9. Algorithmus

Generierung mit arithmetischen Progressionen

Wir berechnen die x-Koordinaten der Punkte $P(h_i) \cdot Q$. Dabei ist R ein Feld der Größe $k+1$, in dem wir die Punkte einer Diagonale speichern. Die berechneten x-Koordinaten speichern wir im Feld a.

EINGABE: Schranke d
AUSGABE: Feld a von x-Koordinaten

<u>INITIALISIERUNG</u>

(1) **for** $(i := 0; i \leq k; i++)$ **do**
(2) $\quad R[i] := (h_i)^k \cdot Q$
(3) **for** $(i := 1; i < k; i++)$ **do**
(4) $\quad$ **for** $(j := e; j \geq i; j--)$ **do**
(5) $\qquad R[j] := R[j] - R[j-1]$

<u>BERECHNUNG</u>

(6) **for** $(i := 0; i \leq d; i++)$ **do**
(7) $\quad a[i] := x[R[0]]$
(8) $\quad$ **for** $(j := 0; j < k; j++)$ **do**
(9) $\qquad R[j] := R[j] + R[j+1]$
(10) **od**
(11) **return** (a)

4.10. Bemerkung

Im obigen Algorithmus können wir die Schleife in den Zeilen (8) - (9) durch eine Addition von Vektoren von Punkten ersetzen. Dabei nutzen wir den ggT-Trick (siehe [Be93]). Damit ersetzen wir die k Invertierungen durch eine Invertierung und $3 \cdot (k-1)$ Multiplikationen mit Reduktion.

Kapitel 5

Die Implementierung

Nach den theoretischen Betrachtungen in den Kapiteln 1 - 4 wollen wir uns in diesem Abschnitt mit der Implementierung und praktischen Erfahrungen beschäftigen. Alle Standardfunktionen mit langen Zahlen haben wir auf folgenden Langzahlarithmetiken implementiert.

- C-Version mit libI - Arithmetik (siehe [De91]),
- C++ - Version mit GF - Arithmetik (siehe [Sh95]),
- C++ - Version mit LiDIA - Arithmetik (siehe [BiBuPa95]).

Die Darstellung von elliptischen Kurven und Punkten ist jeweils ähnlich. In den C++-Versionen verwenden wir Klassen, in der C-Version gibt es nur Strukturen.

Beim Faktorisieren mit den Algorithmen **Probedivision**, **ECM** und **Quadratisches Sieb** ([De93], [So95]) benötigen wir Primzahlen in aufsteigender Reihenfolge. Nun stellen wir ein effizientes Verfahren zur Berechnung dieser Primzahlen vor. Dieses ist genau auf die für uns wichtigen Belange ausgerichtet.

5.1 Berechnung von Primzahltafeln

Das Standardverfahren zur Berechnung von Primzahlen ist das „Sieb des Erathosthenes“. Wir wollen alle Primzahlen bis zur oberen Schranke max bestimmen. Dazu kreieren wir zu Beginn eine Liste aller natürlichen Zahlen bis zu dieser Schranke max. Aus dieser Liste streichen wir sukzessive alle echten Vielfachen von 2 (2 selbst wird nicht gestrichen), dann die echten Vielfachen von 3, usw. Wenn wir bei max angelangt sind, enthält diese Liste alle Primzahlen bis max. Wir können direkt einige kleine Verbesserungen angeben:

- Wir können von Anfang an nur die ungeraden Zahlen im Feld abspeichern, es gibt nur eine gerade Primzahl, die 2.

- Wir müssen nur mit Primzahlen sieben, alle Vielfachen von zusammengesetzten Zahlen werden auch beim Sieben mit Primzahlen gestrichen.
- Wir müssen nur bis $\sqrt{max}$ sieben, wenn max die Schranke für die Primzahlen ist.

5.1. Beispiel
Wir wollen alle ungeraden Primzahlen berechnen, die kleiner als 40 sind. Dabei wenden wir die vorne angegebenen Verbesserungen an. Die Schranke für die Primzahlen ist $max = 40$. Damit müssen wir nur mit allen ungeraden Primzahlen kleiner als $\lfloor\sqrt{max}\rfloor = 6$ sieben.

In der 1. Zeile steht die Ausgangsliste. In Zeile 2 streichen wir die Vielfachen von 3. Nach dem Streichen aller Vielfachen von 5 erhalten wir alle ungeraden Primzahlen kleiner als 40.

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

3 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 25 ~~27~~ 29 31 ~~33~~ 35 37 ~~39~~

3 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 ~~25~~ ~~27~~ 29 31 ~~33~~ ~~35~~ 37 ~~39~~

Abbildung 5.1: Generierung der Primzahlen bis 40

In der verbesserten Standard-Continuation benötigen wir die Primzahlen bis zur Schranke 77 351 879, wenn wir nach 33-stelligen Faktoren suchen (siehe Tabelle 2.2). Für jede ungerade Zahl müssen wir im Rechner 32 Bit oder 4 Byte verwenden, so daß wir mit obigen Verbesserungen einen Speicherplatzbedarf von 154 703 756 Byte = 154 MB haben, was für praktische Anwendungen deutlich zu viel ist.

In unserer Implementierung verwenden wir eine einfache Kodierung, um alle ungeraden Zahlen darzustellen. Wir stellen jede mögliche Primzahl als Eintrag in einem Bitfeld dar, also entspricht das 1.Bit der 3, das zweite Bit der 5 usw. Dadurch sinkt der Speicherplatzbedarf für die oben genannte Schranke auf knapp 5 MB.

Gleichzeitig sehen wir aber, daß wir bei ECM nicht zu jedem Zeitpunkt die komplette Primzahltafel im Speicher halten müssen. Da wir die Primzahlen nur in aufsteigender Reihenfolge benötigen, können wir nur ein kleines Intervall sieben und im Bedarfsfall das folgende.

Zur Darstellung des Bitfeldes verwenden wir ein Feld von *unsigned long's*, wobei wir als Bitlänge eines *unsigned long* 32 Bit voraussetzen, was auf den von uns benutzten Architekturen gilt. Dabei erkennen wir leicht eine Gemeinsamkeit beim Auftreten der Vielfachen der kleinen Primzahlen, die wir nun für einen Datentyp von 16 Bit illustrieren. In der folgenden Abbildung stellt jedes Kästchen 1 Bit eines 16-Bit Datentyps dar. Mit den Pfeilen haben wir alle Vielfachen von 3 markiert.

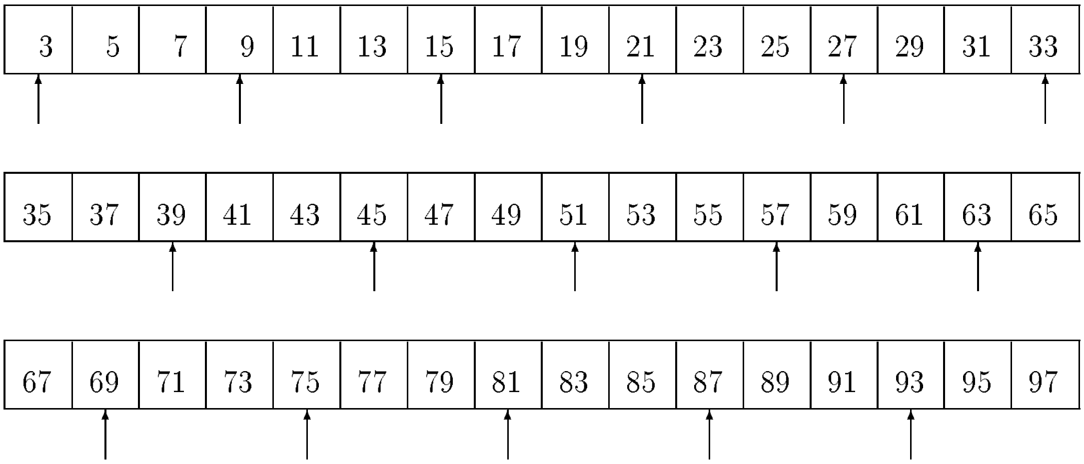


Abbildung 5.2: Vielfache von 3

Wir sehen, daß die Vielfachen der 3 gleichverteilt sind und in jedem Intervall verschiebt sich die Anordnung um ein Bit nach links. Diese Beobachtung nutzen wir aus, wir erstellen uns Masken für alle kleinen Primzahlen $(3, 5, \ldots, 31)$ — diese haben in jedem *unsigned long* mindestens ein Vorkommen — und sieben unser Bitfeld mittels dieser Masken vor. Dadurch können wir schon sehr viele ungerade zusammengesetzte Zahlen markieren, so daß sich der Aufwand beim normalen Sieben deutlich verringert. Bei Anwendung der Maske zur Zahl 3 streichen wir in jedem *unsigned long* 11 zusammengesetzte Zahlen.

Die Vielfachen der größeren Primzahlen streichen wir im Bitfeld, indem wir den ersten Eintrag p stehenlassen und dann alle Vorkommen $p + 2kp,\ k \in \mathbb{N}$ als zusammengesetzt markieren, bis $p + 2kp$ größer als die Primzahlgrenze ist.

Im ECM-Algorithmus benötigen wir die Primzahlen immer in aufsteigender Reihenfolge. Deshalb beschränken wir die Größe der Bitfelder abhängig vom vorhandenen Speicherplatz. Dabei unterscheiden wir folgende Fälle:

- Der zur Verfügung gestellte Speicherplatz reicht aus, um das komplette Bitfeld im Speicher zu halten. Dann allokieren wir nur ein Feld *table*, das diese Bitliste enthält.

- Wir haben nicht genügend Speicherplatz, um alle Primzahlen in einem Intervall zu sieben. Diesen Fall stellen wir etwas ausführlicher vor.

Bevor wir das Sieben auf mehreren Intervallen vorstellen, wollen wir noch auf die Möglichkeit zum Auslesen von Primzahlen eingehen. Mit der Initialisierung der Primzahlliste setzen wir einen Zeiger auf das Bit, welches die angegebene kleinste gewünschte Zahl darstellt. Dieser Zeiger wird beim Auslesen einer Primzahl in der Primliste verschoben, bis wir das nächste als Primzahl markierte Bit finden. Dieses decodieren wir und haben den Wert der Primzahl bestimmt.

Beim Sieben auf mehreren Intervallen tritt der Fall auf, daß beim Auslesen der Zeiger in die Bitliste am Ende steht. In diesem Fall müssen wir die Primzahlen im anschließenden Intervall sieben und setzen dort den Zeiger an den Anfang der Bitliste. Um das nächste Intervall zu sieben, benötigen wir 3 Felder, um alle notwendigen Daten zu speichern:

- *table* : Dieses Feld enthält eine Bitliste, die die Primzahlen im momentan aktuellen Intervall enthält.
- *table2* : In diesem Feld speichern wir alle Primzahlen bis $\sqrt{max}$, also alle Primzahlen, die wir zum Sieben benötigen.
- *over* : Dieses Feld nutzen wir beim Übergang von einem Intervall aufs nächste. Um die Vielfachen der Primzahlen $37, 41, \ldots, \sqrt{max}$ zu Sieben, benötigen wir das erste Auftreten eines Vielfachen der jeweiligen Primzahl im neuen Intervall. Dieses speichern wir beim Sieben auf dem Vorgängerintervall im Feld *over* und können so leicht aufsetzen.

  Beispiel: Wir haben im 1. Intervall die Primzahlen bis 100000 gesiebt. Der Offset für die Primzahl 37 ist also

  $$\left\lceil \frac{100000}{37} \right\rceil \cdot 37 - 100000 \quad = \quad 11.$$

Im folgenden Algorithmus geben wir an, wie wir die Primzahllisten initialisieren. Falls wir die komplette Primzahlliste in einem Intervall im Speicher halten können, sieben wir alle Primzahlen.

Bei der Verwendung von mehreren Intervallen sieben wir die aktuelle Liste soweit, wie wir sie im Speicher halten können. Gleichzeitig legen wir die Felder *table2* und *over* an, die in der oben angegebenen Weise belegt werden. Gegebenenfalls muß beim Auslesen der Primzahlen das nächste Intervall gesiebt werden.

Als dritte Möglichkeit kann auftreten, daß der Speicherplatz für keine der beiden Varianten ausreichend ist. In diesem Fall können wir keine Primzahltafel erstellen, die Funktion gibt eine Fehlermeldung zurück.

5.2. Algorithmus

Initialisierung der Primzahllisten

EINGABE: Schranke max, kleinste gewünschte Primzahl min,
zur Verfügung gestellter Speicherplatz

(1) **if** (genügend Speicherplatz vorhanden) **then**

SIEBE AUF EINEM INTERVALL

(2) Siebe das komplette Bitfeld mit Hilfe der Masken
(3) **for** (Primzahlen $p < \sqrt{max}$) **do**
(4) Streiche alle ungeraden Vielfachen der Primzahl p im Bitfeld
(5) **else**

SIEBE AUF MEHREREN INTERVALLEN

(6) Berechne maximale Größe des Bitfeldes
(7) Siebe das Bitfeld mit Hilfe der Masken
(8) Berechne Primzahlen $p < \sqrt{max}$ in *table2*
(9) **for** (Primzahlen $p < \sqrt{max}$) **do**
(10) Streiche alle ungeraden Vielfachen der Primzahl p im Bitfeld
(11) Speichere den Übertrag für jede Primzahl p im Feld *over*
(12) **od**
(13) **fi**
(14) Setze Primzahlzeiger auf Position von min

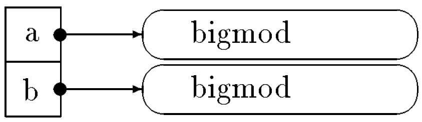
Neben den bisher erwähnten Funktionen zum Initialisieren der Primzahltafeln und Auslesen der Primzahlen kann man auch den Zeiger in der Bitliste auf jede Position setzen. Dabei wird der gewünschte Wert kodiert und der Zeiger auf dieses Bit gesetzt. Falls der Wert gerade ist, wird der Zeiger auf die Position des um 1 erhöhten Wertes gesetzt. Beim Sieben auf mehreren Intervallen kann es auftreten, daß der Zeiger auf eine Position gesetzt werden soll, die nicht im aktuell gesiebten Intervall liegt. In diesem Fall wird ein passendes Intervall neu gesiebt und der Zeiger entsprechend gesetzt.
Die Funktionsaufrufe beschreiben wir in Abschnitt B.1.

5.2 Arithmetik für elliptische Kurven

In diesem Abschnitt beschreiben wir die Darstellungen zum Rechnen mit Punkten auf elliptischen Kurven. Dabei betrachten wir beide vorgestellten Parametrisierungen, die Weierstraß- und die Montgomery-Chudnowsky-Parametrisierung. Zuerst gehen wir auf die Darstellung der elliptischen Kurven ein. Danach stellen wir die Realisierung von Punkten in beiden vorgestellten Parametrisierungen vor. Dabei verwenden wir einen abstrakten Datentyp *bigmod*. Dieser entspricht einer modulo N reduzierten Zahl.
Die Funktionen zum Rechnen mit Punkten auf elliptischen Kurven stellen eine direkte Übertragung der mathematischen Formeln bzw. der vorgestellten Algorithmen dar. In Abschnitt B.2 beschreiben wir die Aufrufe der jeweiligen Funktionen.

5.2.1 Elliptische Kurven

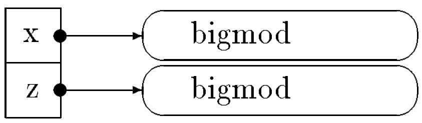
Wir wir bei der Einführung der Weierstraß-Parametrisierung und der Montgomery-Chudnowsky-Parametrisierung gesehen haben, benötigen wir die elliptische Kurve nur bei der Addition zweier Punkte auf der Kurve. Ansonsten führen wir keine Operationen auf den Kurven aus. Damit erhalten wir als Struktur für eine elliptische Kurve E.



Als Funktionen auf dieser Datenstruktur benötigen wir nur Allokierung, Zugriffsoperationen und eine Zuweisung.

5.2.2 Punkte in Montgomery-Chudnowsky-Form

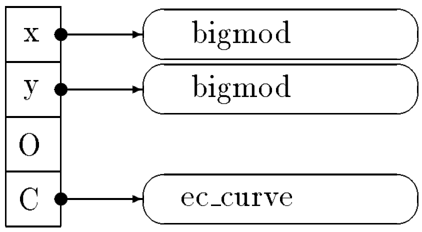
Bei den in Kapitel 2 vorgestellten Verbesserungen der Grundversion arbeiten wir mit Punkten auf elliptischen Kurven in Montgomery-Chudnowsky-Form. Der Datentyp heißt *ec_pointM* und beinhaltet die Koordinaten x, z eines Punktes.



Die Werte der Kurvenparameter a, b gehen in den Berechnungen nur bei der Punktverdoppelung ein. Dabei benötigen wir den Wert $a_2 = (a+2)/4$. Diesen berechnen wir vor und übergeben ihn jeweils den Additionsfunktionen als Parameter.

5.2.3 Punkte in Weierstraß-Form

Der Aufbau der Punkte in Weierstraß-Form ist etwas aufwendiger. Wir benötigen zu jeder Punktverdoppelung den Kurvenparameter a. Außerdem müssen wir zur Anwendung des ggT-Tricks bei der Invertierung mehrere elliptische Kurven parallel betrachten. Deshalb enthält jeder Punkt einen Zeiger auf seine dazugehörige elliptische Kurve. Zusätzlich benötigen wir ein Flag, das für endliche Punkte 0 ist, für den Nullpunkt 1. Wir haben also für den Datentyp *ec_pointW* folgenden Aufbau:



Diese Darstellung ist bei allen von uns implementierten Versionen gleich. Außerdem haben wir versucht, die Funktionsaufrufe soweit wie möglich auf allen Arithmetiken bezüglich der Reihenfolge und Anzahl der Parameter gleichzugestalten.

5.3 Faktorisieren mit ECM

Zur effizienten Ausführung der FFT-Continuation benötigen wir eine effiziente Polynomarithmetik. Zu diesem Zweck haben wir die bestehende Polynomarithmetik von Victor Shoup um einige Funktionen erweitert und mehrere Funktionen so umgeschrieben, daß der Speicherplatzbedarf nicht zu groß wird. Deshalb ist die FFT-Continuation nur auf der Langzahlarithmetik GF implementiert. Wir stellen nun zuerst eine sequentielle Version von ECM vor. Danach gehen wir auf die Möglichkeit der Verteilung von ECM ein und beschreiben unsere Lösung.

5.3.1 Installation & Bedienung

Mit unserem Programmpaket können wir mehrere verschiedene ECM-Versionen erstellen:

- Sequentielle Versionen:
  - C-Version mit verbesserter Standard-Continuation
  - C++-Version mit FFT-Continuation
- Master-Worker-Versionen:
  - C-Version mit verbesserter Standard-Continuation

– C++-Version mit FFT-Continuation

Zur Installation müssen wir das File „CONFIG“ editieren. Dabei kann man die verwendeten Compiler und Compileroptionen bestimmen. Zusätzlich müssen die Pfade für die Langzahlarithmetiken korrekt eingestellt werden. Falls wir eine verteilte Version erzeugen wollen, müssen wir auch die Pfade der LiPS-Bibliotheken angeben.

Zum Kompilieren der Programme mit FFT-Continuation kann man noch zwei Fälle unterscheiden. Standardmäßig wird eine Version erzeugt, die die speicherplatzeffizienten Polynomfunktionen aufruft. Diese Version ist langsamer als die Version mit hohem Speicherplatzbedarf. Falls man genügend Hauptspeicher zur Verfügung hat, so kann man vorm Kompilieren die Option `-DHUGE_MEMORY` zu den C++-Flags hinzufügen. Dann wird die schnellste ECM-Version mit FFT-Continuation erzeugt.

Nun haben wir folgende Möglichkeiten zum Kompilieren:

- `make single`: erzeugt die sequentiellen ECM-Programme
- `make netz`: erzeugt die verteilten Programme
- `make all`: erzeugt alle ausführbaren Programme

Die ausführbaren Programme finden wir in den Unterverzeichnissen `prog_C` bzw. `prog_C++`. Zum Faktorisieren mit den erzeugten Programmen benötigt man noch die Parameterfiles `ecm-parameters-std` bzw. `ecm-parameters-fft`, aus denen beim Programmlauf die Parameter eingelesen werden. Die Benutzung der erzeugten Programme beschreiben wir in den folgenden Abschnitten.

Die Laufzeit von ECM ist abhängig von der Größe der Faktoren, die man sucht. In Tabelle 2.2 haben wir z.B. optimale Parameter für die verbesserte Standard-Continuation angegeben. Wir steuern nun die Wahl der Parameter mit Hilfe des Files `ecm-strategy`. In diesem bestimmen wir die Strategie zum Finden von Faktoren. Dabei geben wir in jeder Zeile die Dezimallänge der Faktoren an, nach der wir suchen wollen. Abgeschlossen wird das ganze durch 0.

Beispiel: 15
25
0

Wir suchen also zuerst nach Faktoren mit 15 Dezimalstellen, danach nach 25-stelligen Faktoren. Neben dieser Steuerungsmöglichkeit haben wir noch weitere implementiert. Diese stellen wir nun getrennt vor.

5.3.2 Die sequentiellen Versionen von ECM

In allen Programmen verwenden wir die jeweils schnellsten Grundfunktionen. Deshalb berechnen wir in der 1. Phase von ECM die Multiplikation des Punktes mit einem Skalar mit Hilfe der Lukas-Ketten (siehe Algorithmus 2.6). Als Generator verwenden wir den 16-Generator (siehe Algorithmus 2.8). Zur Multiplikation mit Skalaren in der Weierstraß-Form benutzen wir schnelle Exponentiation. Dabei haben wir die Verbesserungen von Morain implementiert (siehe [MoOl90]).

ECM mit verbesserter Standard-Continuation

Wie oben angegeben liest das Programm `ecm` die Strategie zur Ausführung der Berechnung aus dem File `ecm-strategy` ein. Alle anderen Parameter können über Aufrufparameter eingegeben werden. Dabei haben wir folgende Parameter (`<num>` ist jeweils eine Zahl aus $\mathbb{N}$):

`-h`	Gibt ein Help-File aus.
`-t <num>`	Obere Schranke für Probedivision.
`-r <num>`	Restlänge der Zahl in Dezimalstellen.
`-v <num>`	Wenn num=0 ist, werden nur die Faktoren ausgegeben, bei anderen Werten ausführliche Ausgabe der Parameter, usw.
`-i`	Alle Parameter müssen beim Programmlauf eingegeben werden, auch die Strategie.
`-num <num>`	Eingabe der zu faktorisierenden Zahl.
`-time`	Das Programm legt ein Time-File an. In diesem werden die Realzeiten für alle wichtigen Schritte gespeichert und am Ende wird ein komplettes Timing ausgegeben.
`-p <file>`	Das Programm legt beim Abbruch ein Aufsetzpunktefile an. Wir können die Strategie fortsetzen, indem wir das Programm mit der Option `-c` neustarten.
`-c <file>`	Das Programm liest die Parameter aus dem File `<file>` ein und setzt die Berechnung zum angegebenen Zeitpunkt fort.

Wie wir schon bei den beiden letzten Parametern ablesen können, können wir die Berechnung nach jeder berechneten Kurve unterbrechen. Dies geschieht durch Anlegen des Files `stop-ecm` in dem Verzeichnis, in dem der momentane Prozess läuft. Dies kann man durch „`touch stop-ecm`“ tun. Falls `ecm` mit Option `-p <file>` gestartet wurde, wird der momentane Berechnungsstand im angegebenen File `<file>` abgespeichert.

ECM mit FFT-Continuation

Das ausführbare Programm mit FFT-Continuation heißt `ecmfft` und steht im Verzeichnis `prog_C++`.
In der FFT-Continuation rechnen wir mit Polynomen über dem Ring $\mathbb{Z}/N\mathbb{Z}$. Dazu müssen wir die Polynomarithmetik initialisieren. Diese Initialisierung ist abhängig vom Modul N und kann nachträglich nicht mehr geändert werden. Deshalb terminiert das Programm mit FFT-Continuation direkt nach dem Finden eines echten

Faktors. Zur Verwendung des Programms sollte sichergestellt sein, daß die zu faktorisierende Zahl keine kleinen Faktoren enthält. Für kleine Faktoren ist die verbesserte Standard-Continuation auch schneller. Wir führen keine Überprüfungen auf kleine Faktoren durch. Die Eingabe kann wiederum über die Kommandozeile ausgeführt werden. Aus den oben genannten Gründen gibt es weniger Steuerungsparameter.

`-h`	Gibt ein Help-File aus.
`-v <num>`	Wenn num=0 ist, werden nur die Faktoren ausgegeben, bei anderen Werten ausführliche Ausgabe der Parameter, usw..
`-i`	Alle Parameter müssen beim Programmlauf eingegeben werden, auch die Strategie.
`-num <num>`	Eingabe der zu faktorisierenden Zahl.
`-time`	Das Programm legt ein Time-File an. In diesem werden die Realzeiten für alle wichtigen Schritte gespeichert und am Ende wird ein komplettes Timing ausgegeben.
`-p <file>`	Das Programm legt beim Abbruch ein Aufsetzpunktefile an. Wir können die Strategie fortsetzen, indem wir das Programm mit der Option `-c` neustarten.
`-c <file>`	Das Programm liest die Parameter aus dem File `<file>` ein und setzt die Berechnung zum angegebenen Zeitpunkt fort.

5.3.3 Die verteilte Implementierung

Zur Faktorisierung der Partitionszahl p(15727) benötigten wir eine Rechenzeit von ca. 17 MIPS-Jahren. 1 MIPS-Jahr entspricht dabei der Rechenleistung eines Rechners mit 1 MIPS (Million Instructions Per Second) in einem Jahr. Die von uns genutzten „SUN SPARC station ELC" Rechner haben eine MIPS-Leistung von ungefähr 21 MIPS. Wir hätten also auf solch einem Rechner fast 300 Tage durchrechnen müssen, um den Faktor zu finden. Das legt natürlich die Verteilung der Berechnungen nahe.

Zur Verteilung von ECM sind zwei Strategien möglich:

- Wie wir bei der Parameterbestimmung gesehen haben, müssen wir z.B. 1088 Durchläufe auf verschiedenen Kurven berechnen, um einen Faktor mit 33 Dezimalstellen zu finden. Dies legt die erste Strategie nahe, einfach die Kurven auf verschiedene Rechner zu verteilen und dort die Berechnungen durchzuführen.

- Neben dieser Parallelisierung kann man natürlich auch die Berechnungen auf einer Kurve parallel ausführen. Als Beispiel können wir bei der Polynommultiplikation die Berechnungen modulo der kleinen Primzahlen (siehe Abschnitt 3.2) auf verschiedenen Prozessoren ausführen. Weitere Parallelisierungen finden wir in [DiLe92] und [Mo92/2].

In unserer Implementierung haben wir die erste Möglichkeit gewählt. Diese hat den Vorteil, daß eine geringe Kommunikation zwischen den Prozessen durchgeführt werden muß. Bei der zweiten Version muß mehr kommuniziert werden, diese bietet sich auf Parallelrechnern mit schneller Kommunikation an.

Dabei nutzen wir das am Lehrstuhl entwickelte verteilte System LiPS [RoSe92]. Dieses stellt uns Kommunikationsroutinen zwischen allen Rechnern zur Verfügung, die im LiPS-Netz eingebunden sind. Da die Berechnungen auf verschiedenen Kurven aber vollkommen unabhängig voneinander ablaufen, starten wir auf einem Rechner ein Steuerprogramm, den sogenannten „Master“. Dieser verteilt die Aufgaben an die „Worker“ und führt die Verwaltung aus. Im Worker werden die eigentlichen Berechnungen ausgeführt, die Ergebnisse werden wieder an den Master geschickt. Die Kommunikation spielt sich also wie in der folgenden Abbildung ab.

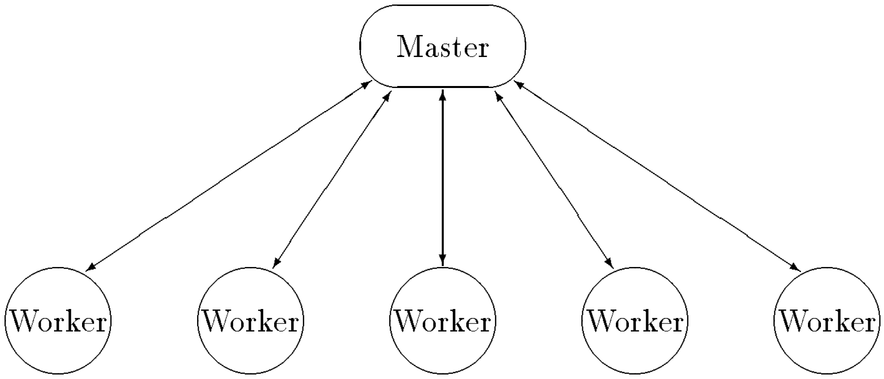


Abbildung 5.3: Master-Worker-Konfiguration

Das Master - Programm

Das Master-Programm führt nur Vorberechnungen aus und verwaltet die berechneten Jobs. Die Vorberechnungen sind

- Probedivision,
- ECM mit kleinen Parametern (ECMI),
- Berechnung der Parameter.

Dabei sichern wir die gefundenen Faktoren in einem Faktorfile. Mit Hilfe dieser Vorberechnungen stellen wir sicher, daß in der zu faktorisierenden Zahl die kleinen Faktoren gefunden werden. Damit sparen wir einige Kommunikation ein und belasten das Netzwerk weniger. Wir stellen nun zuerst die Möglichkeiten vor, das Master-Programm zu steuern. Die Eingabe ist

- Symbol für die Faktorisierung: Dieses bestimmt den Namen des Faktor- und des Checkpoint-Files. Bei den Partitionszahlen ist dieser Name z.B. P15727, das Faktorfile heißt factors_P15727.
- die zu faktorisierende Zahl N.
- Restlänge der Zahl: Falls wir einen Faktor l der Zahl N finden und die Dezimallänge der Restzahl N/l kleiner als diese Restlänge ist, so beenden wir das Programm. Dieser Parameter bestimmt also den Übergang von ECM zum Quadratischen Sieb.
- Obere Schranke für die Probedivision: Falls diese 0 ist, wird keine Probedivision ausgeführt.
- Info: Falls 1 eingegeben wird, legen die Worker ein Logfile an und protokollieren die Ausgaben im Worker mit, für 0 schreibt das Worker-Programm keine Ausgaben.
- Anzahl der Versuche mit kleinen Parametern.

Gewöhnlicherweise verwenden wir ein Eingabefile, um diese Angaben einzulesen. Wie bei den sequentiellen Programmen können wir auch die verteilte Berechnung zu jedem Zeitpunkt unterbrechen. Das Master-Programm legt dann einen Checkpoint an, der z.B. im File P15727 abgelegt wird. Durch die Option `-c <file>` kann man den Master an der im Checkpoint-File `<file>` angegebenen Stelle wieder aufsetzen.

Nach Ausführung von Probedivision und ECMI lesen wir aus dem File *ecm-strategy* die gewünschte Strategie ein und berechnen die Parameter zu dieser Strategie. Der Master verteilt nun die einzelnen Aufgaben an die Worker und verarbeitet die eingehenden Ergebnisse. Das eingehende Ergebnis baut sich auf aus

- Faktor l der Zahl N (sehr oft ist dieser Faktor 1),
- Rechenzeit in Real-Time,
- Mips-Leistung des Workers,
- Job-Nummer job_nr,
- Anzahl der berechneten Kurven $curves$.

Der Master testet den gefundenen Faktor. Dabei unterscheiden wir 2 Fälle:

- Wir haben einen nichttrivialen Faktor von N gefunden. In diesem Fall berechnen wir die neue zu faktorisierende Zahl (d.h. der Rest). Dann überprüfen wir, ob entweder der Rest prim oder die Restlängenbedingung erfüllt ist. Für diese beiden Fälle bricht der Master die Berechnungen auf dem Netz ab und terminiert. Anderenfalls sendet er an <u>jeden</u> Worker einen neuen Job mit dieser neuen Zahl. Durch diesen Schritt stellen wir sicher, daß der gefundene

Faktor nicht von weiteren Workern nochmals gefunden wird. Außerdem hat die Dezimallänge der zu faktorisierenden Zahl einen großen Einfluß auf die Laufzeit (siehe Tabelle 6.5), die Berechnungen laufen nun schneller ab.

- Wir haben einen trivialen Faktor gefunden. Wir senden an diesen Worker den nächsten freien Job.

Durch den Wert von *curves* wird die Anzahl der berechneten Kurven bestimmt. Dieser ist normalerweise 1. Bei der Vorstellung der Worker-Programme werden wir sehen, daß die Worker beim Fehlschlagen der Kommunikation zum Master einen neue Berechnung mit den gleichen Parametern ausführen. In diesem Fall erhöht sich also die Anzahl der berechneten Kurven.

Das Master-Programm sorgt dafür, daß die benötigte Anzahl von Kurven für die berechneten Parameter vollständig ausgeführt wird. Dazu sendet der Master gleiche Jobs an die verschiedenen Worker, bis die Anzahl der berechneten Kurven die nötige Anzahl überschreitet. Damit gleichen wir ausgefallene Rechner aus, weil nicht die Zahl der gesendeten Jobs entscheidend ist, sondern die Zahl der berechneten und dann zurückgeschickten.

Das Worker - Programm

Wir stellen nun das Worker-Programm vor, das relativ einfach ist. Zuerst geben wir generelle Eigenschaften der Worker an.
Falls ein echter Faktor der Zahl N gefunden wird, speichern wir diesen mit zusätzlichen Informationen ab. Dadurch stellen wir sicher, daß auch bei Übermittlungsproblemen der gefundene Faktor nicht verloren geht. Gleichzeitig senden wir den Faktor zum Master. Falls diese Übertragung nicht korrekt funktioniert, wiederholen wir diese Aktion in immer größeren Zeitabständen. Wir wollen also sicherstellen, daß der Faktor auch bei kurzzeitigen Netzstörungen etc. zum Master übertragen wird.

Falls wir keinen Faktor gefunden haben und die Kommunikation nicht durchführbar ist, rechnen wir nochmals einen Job mit den gleichen Parametern wie im letzten Job. Dies ist auch der Fall, falls nach dem Zurücksenden kein neuer Job vom Master ankommt.

Der Worker arbeitet in einer Endlosschleife. Wenn er einmal korrekte Parameter und die zu faktorisierende Zahl erhalten hat, so kann er autark weiterrechnen. Dies ist günstig für den Fall, daß z.B. der Master-Prozess gestoppt wird. Zum Beenden des Worker-Prozesses haben wir zwei Mechanismen.

Wie in der sequentiellen Version können wir durch Anlegen des Files `stop-ecm` im Verzeichnis, in dem der Master-Prozess gestartet wurde, die Berechnung abbrechen. Der Master beendet alle Worker, legt das Aufsetzpunkte-File an und terminiert.

Eine zweite Möglichkeit bietet das System LiPS, mit dessen Hilfe wir die Verteilung der Aufgaben durchführen. Mit Hilfe der Oberfläche **limon** können wir einzelne oder alle Workerprozesse beenden.

Die Arbeitsweise des Workers beschreiben wir leicht in folgendem Algorithmus:

5.3. Algorithmus

ECM - STD - Worker	
(1)	Initialisiere Netzwerkfunktionen
(2)	**do**
(3)	Warte auf Job
(4)	Berechne ECM mit erhaltenem Job
(5)	Sende Ergebnis an Master
(6)	**while** (1)

Dabei testen wir beim Berechnen von ECM in regelmäßigen Abständen, ob ein neuer Job angekommen ist. Falls dies der Fall ist, brechen wir den momentan berechneten Job ab und beginnen die Berechnung mit diesem neuen Job. Dies tritt genau dann auf, wenn ein anderer Worker einen Faktor gefunden hat. Das Master-Programm verschickt dann neue Jobs mit der neuen zu faktorisierenden Zahl.

Die FFT - Netzversion

Bei der Verwendung der FFT - Continuation müssen wir die Polynomarithmetik initialisieren. Diese Initialisierung ist abhängig von der Dezimallänge des Moduls — also der zu faktorisierenden Zahl — und der Größe der auftretenden Polynome. Eine Änderung zu einem späteren Zeitpunkt ist nicht möglich. Dies schränkt die Verwendung der FFT - Continuation etwas ein. Wir können nur Faktoren einer vorher bestimmten Größenordnung suchen.

Wie in der sequentiellen Version führen wir in der Netzversion weder Probedivision noch ein ECM mit kleinen Parametern aus. Die Eingabe beschränkt sich also auf die drei Punkte

- Symbol für die Faktorisierung,
- die zu faktorisierende Zahl N,
- Info.

Nach dem Lesen des Files `ecm-strategy` liest der Master die Parameter aus dem File `ecm-parameters-fft` ein, berechnet die Jobs und verteilt diese an die Worker. Danach arbeitet er nur noch die eingehenden Ergebnisse ab. Falls der Worker einen trivialen Faktor geschickt hat, erniedrigt der Master die Zahl der noch zu

berechnenden Kurven um die Anzahl der von diesem Worker berechneten. Bei einem echten Faktor gibt der Master die Faktoren im Faktorfile aus und beendet die Berechnungen auf dem Netz.
Neben dieser Terminierung beendet der Master alle Worker, falls die komplette Strategie abgearbeitet wurde.

Die Berechnungen im Worker laufen solange, bis entweder ein Faktor gefunden oder alle Jobs ausgeführt wurden. Weil sich die Jobs nicht mehr ändern können, muß der Worker nur zu Beginn einen Job mit allen Parametern erhalten, danach läuft nur noch Kommunikation vom Worker zum Master.

5.4. Algorithmus

ECM - FFT - Worker

(1) Initialisiere Netzwerkfunktionen
(2) Warte auf 1. Job
(3) Initialisiere die Polynomarithmetik
(4) **do**
(5) Berechne ECM mit erhaltenem Job
(6) Sende Ergebnis an Master
(7) **while** (1)

Auch im ECM-FFT-Worker führen wir eine Sicherung der Kurve und des Faktors aus, falls wir einen echten Faktor finden. Dadurch können wir Probleme, die beim Netzbetrieb auftreten, ausschließen.

Mit Hilfe der vorgestellten Programme haben wir beim Bearbeiten meiner Diplomarbeit viele große Faktoren gefunden. Dabei haben wir anfangs nur mit ECM und der verbesserten Standard-Continuation gearbeitet, später auch die FFT-Continuation auf dem Netz benutzt. Durch die Verteilung ist es erst möglich geworden, sehr viele Tests mit hohen Parametern auszuführen. Dadurch haben wir viele Faktoren gefunden. In Kapitel 7 stellen wir die insbesondere mit den verteilten Programmen errungenen Erfolge vor.

Kapitel 6

Laufzeittests

In diesem Kapitel geben wir verschiedene Tabellen mit erreichten Laufzeiten an. Dabei betrachten wir zum Beispiel, wie sich die Zahlgröße auf die Laufzeit auswirkt oder welche realen Zeiten erreicht werden. Die Reallaufzeiten, die in diesem Abschnitt angegeben werden, wurden jeweils auf einer „SUN SPARC station ELC“ (mit ca. 21 MIPS (Million Instructions per Second)) gemessen. Falls keine anderen Angaben gemacht werden, führen wir die Tests mit 100-stelligen Zahlen aus.

6.1 Die Arithmetik für elliptische Kurven

Wir haben verschiedene Algorithmen zur Multiplikation eines Punktes mit einem Skalar implementiert. Diese haben wir untereinander verglichen und dann die jeweils schnellste Variante in unsere Programme eingebaut. Im ersten kleinen Test sehen wir Vergleichswerte zur Multiplikation in Weierstraß-Darstellung. Dabei vergleichen wir die normale schnelle Exponentiation mit der durch den Morain-Trick (siehe [MoOl90]) verbesserten Version. Um aussagekräftige Ergebnisse zu erhalten, haben wir die 1. Phase von ECM mit beiden Versionen und verschiedenen Parametern durchgerechnet. Die Laufzeiten wurden jeweils in Sekunden gemessen. Das Verhältnis berechnet sich nach folgender Formel:

$$\text{Verhältnis} \;=\; \frac{\text{Zeit für schnelle Exp. mit Morain-Trick}}{\text{Zeit für schnelle Exp. ohne Morain-Trick}}.$$

Auch in der Montgomery-Chudnowsky-Parametrisierung haben wir zwei Algorithmen vorgestellt, um einen Punkt mit einem Skalar zu multiplizieren. Dies ist zum einen der binäre Algorithmus (siehe Algorithmus 2.3) und zum zweiten die Berechnung mit Hilfe von Lukas-Ketten (siehe Algorithmus 2.6). Hier haben wir die gleichen Tests ausgeführt und in Tabelle 6.2 aufgereiht. Die Laufzeit ist wiederum in Sekunden angegeben. Um die Verbesserung aufzuzeigen berechnen wir

$$\text{Verhältnis} \;=\; \frac{\text{Zeit für Algorithmus mit Lukas-Ketten}}{\text{Zeit für binären Algorithmus}}.$$

B	Schnelle Exp.	Schnelle Exp. mit Morain-Trick	Verhältnis
500	13.2 sec	12.2 sec	0.93
1 000	26.8 sec	24.4 sec	0.91
10 000	270.6 sec	248.6 sec	0.92
50 000	1 369.7 sec	1 248.6 sec	0.91

Tabelle 6.1: Auswirkung des Morain-Tricks

B	Bin. Alg.	Lukas-Ketten	Verhältnis
500	3.3 sec	2.9 sec	0.88
1 000	6.5 sec	5.8 sec	0.89
10 000	68.0 sec	59.9 sec	0.88
50 000	244.1 sec	303.0 sec	0.88

Tabelle 6.2: Vergleich: Binärer Algorithmus — Lukas-Ketten

6.2 Berechnung der 1. Phase von ECM

Beim Vergleich der Ergebnisse aus dem letzten Abschnitt sieht man, daß die Skalarmultiplikation in der Montgomery-Chudnowsky-Parametrisierung deutlich schneller als beide Varianten in Weierstraß-Darstellung ist. Dies erklärt sich daher, daß in der Weierstraß-Parametrisierung bei jeder Punktaddition und -verdoppelung eine Invertierung ausgeführt werden muß. Diese Invertierung können wir teilweise durch Multiplikationen ersetzen, wenn wir mehrere elliptische Kurven parallel betrachten und den ggT-Trick anwenden (siehe [Be93]).

Wir vergleichen in diesem Abschnitt mehrere Varianten zur Ausführung der 1. Phase von ECM. Dazu berechnen wir

$$k \cdot P \text{ mit } k = \prod_{\mathbb{P} \ni q \leq B} q^{e_q}, \quad e_q = \max\{j \in \mathbb{N} : q^j \leq B\}.$$

Zum Vergleich geben wir die Zeiten an, die wir bei der Berechnung der 1. Phase für $B = 100\,000$ mit den langsameren Algorithmen erreicht haben:

- Montgomery-Chudnowsky-Form (Binärer Alg.): 11 min 32 sec
- Weierstraß-Form (Schnelle Exp. mit Morain-Trick): 41 min 42 sec
- Weierstraß-Form (Schnelle Exp. ohne Morain-Trick): 45 min 13 sec

Folgende Möglichkeiten zum Ausführen dieser Berechnung haben wir ausführlicher verglichen:

(1) Homogene Koordinaten in Montgomery-Chudnowsky-Form unter Verwendung des Algorithmus mit Lukas-Ketten (siehe Algorithmus 2.6).

(2) Weierstraß-Koordinaten mit ggT-Trick:
Dabei berechnen wir $k \cdot P$ parallel auf 15 Kurven und müssen deshalb jeweils nur einen ggT für alle parallelen Additionen berechnen (siehe [Be93]). Bei der schnellen Exponentiation wenden wir den Morain-Trick an.

(3) Weierstraß-Koordinaten mit Vorberechnung optimaler Skalare:
Im binären Algorithmus führen wir eine Punktaddition und eine Punktverdoppelung aus, falls das momentan betrachtete Bit des Skalars 1 ist. Für eine 0 müssen wir nur eine Punktverdoppelung berechnen. Durch Vorberechnungen kann man die Primzahlpotenzen unter der Schranke B so kombinieren, daß die Skalare sehr wenige 1-Einträge haben (siehe [DiLe92]).

B	(1)		(2)	(3)
	1 Kurve	15 Kurven		
10 000	1m 02s	15m 30s	17m 33s	18m 32s
100 000	10m 10s	2h 32m 30s	2h 58m 10s	2h 39m 07s
500 000	51m 20s	12h 50m 00s	14h 56m 31s	13h 10m 41s
1 000 000	1h 42m 11s	25h 32m 45s	29h 52m 27s	26h 15m 10s

Tabelle 6.3: Vergleich der 1. Phase

Die Skalarmultiplikation in Montgomery-Chudnowsky-Form unter Verwendung von Algorithmus 2.6 ist in unserer Implementierung die schnellste Variante. Dabei stellt die Anwendung der Lukas-Ketten den entscheidenden Vorteil gegenüber der Vorberechnung der Skalare dar, der vorgestellte binäre Algorithmus 2.3 ist langsamer als Variante (3). Neben diesem Vorteil können wir in der verteilten Version durch Verwendung der Montgomery-Chudnowsky-Form relativ schnell die Berechnung einer Kurve abschließen. Dies nützt uns auf zweierlei Weise:

- Beim Finden eines Faktors in einem Worker bzw. beim Abbrechen der Berechnung auf dem Netz ist die investierte Zeit, die für die Ausführung des momentan berechneten Jobs auf den anderen Rechnern benötigt wurde, relativ gering.

- Die Gefahr sinkt, daß durch das Terminieren eines Workers durch Dritte Zwischenergebnisse gelöscht werden oder z.B. die Berechnung vor Ausführen der Continuation beendet wird. Damit müssen wir kein Checkpointing im Worker durchführen.

6.3 Die Polynomarithmetik

In diesem Abschnitt geben wir die Realzeiten an, die wir beim Rechnen mit der Polynomarithmetik erhalten. Wir nutzen dabei die Polynomarithmetik von Victor Shoup (siehe [Sh95]). Die für ECM relevanten Funktionen haben wir teilweise umgeschrieben, um den Speicherplatzbedarf zu senken. Dadurch sind die Funktionen etwas langsamer als in der Grundversion. Wir geben zu jedem Test jeweils Laufzeit (in Sekunden) und Speicherplatzbedarf (in KiloByte) an. Alle Tests haben wir jeweils mit einer Zahl mit 100 Dezimalstellen auf einer „SUN SPARC station ELC" durchgeführt.

Operation	Grad	Realzeit (in sec)	Speicherplatz (in kB)
Polynom-multiplikation	1 000	5.45 sec	540 kB
	2 000	11.60 sec	1 248 kB
	4 000	23.67 sec	2 228 kB
	8000	48.96 sec	4 184 kB
Aufbau aus den Wurzeln	1 000	16.72 sec	740 kB
	2 000	38.25 sec	1 164 kB
	4 000	87.76 sec	2 060 kB
	8 000	196.71 sec	3 712 kB
Polynom-ggT	1 000	125.27 sec	1 008 kB
	2 000	296.20 sec	1 700 kB
	4 000	702.00 sec	3 116 kB
	8 000	1 540.92 sec	5 668 kB

Tabelle 6.4: Polynomoperationen bei ECM

6.4 Die Größe der zu faktorisierenden Zahl

Die Arithmetik im ECM-Algorithmus spielt sich immer modulo der zu faktorisierenden Zahl N ab. Deshalb ist die Dezimallänge der Zahl N von großer Bedeutung für die Laufzeit. Um diese Abhängigkeit deutlich zu machen, haben wir Zahlen verschiedener Dezimallängen $(50, 75, 100, \ldots, 250$ Dezimalstellen) mit Parametern zum Finden 30-stelliger Faktoren getestet. Der Test der 100-stelligen Zahl ist dabei unser Bezugswert, wobei wir hier keine absoluten Zeiten angeben.
Dabei verwenden wir den 16-Generator und berechnen $k \cdot P$ mit Hilfe der Lukas-Ketten. In der anschließenden FFT-Continuation benutzen mit als Zufallsfunktion den Algorithmus aus Abschnitt 4.4.3 mit $k = 12$.
Wir unterscheiden in der Tabelle in 1. Phase und Continuation, weil wir verschiedene Langzahlarithmetiken verwenden. Außerdem macht sich bemerkbar, daß die Operationen in der Continuation modular ausgeführt werden.

<table>
<tr><td colspan="2">#Stellen</td><td>50</td><td>100</td><td>150</td><td>200</td><td>250</td></tr>
<tr><td rowspan="3">Faktor</td><td>1. Phase</td><td>0.37</td><td>1.00</td><td>2.03</td><td>3.40</td><td>5.16</td></tr>
<tr><td>Continuation</td><td>0.45</td><td>1.00</td><td>1.91</td><td>2.95</td><td>4.17</td></tr>
<tr><td>gesamt</td><td>0.41</td><td>1.00</td><td>2.00</td><td>3.20</td><td>4.71</td></tr>
</table>

Tabelle 6.5: Abhängigkeit der Laufzeit von der Größe der zu faktorisierenden Zahl

6.5 Erreichte Laufzeiten

Durch Verwendung der berechneten Parameter finden wir mit Wahrscheinlichkeit von ca. 50 % einen Faktor in der angegebenen Größenordnung. Dabei liegt die Wahrscheinlichkeit bei der verbesserten Standard-Continuation eher etwas höher (fast 60%). Dadurch haben wir bei den Laufzeiten eine geringe Verschiebung zugunsten der FFT-Continuation. Je höher wir die Schranken B und D bzw. B, d_1 und d_2 wählen, desto länger dauert natürlich ein Durchlauf von ECM.
Wir geben nun Realzeiten an, die wir bei der Faktorisierung einer 100-stelligen Zahl gemessen haben. Wir verwenden den 16-Generator und rechnen in der 1. Phase mit Algorithmus 2.6. Bei der FFT-Continuation generieren wir die zufälligen Vielfachen mit Hilfe von Algorithmus 4.9 mit $k = 12$. Zur Ausführung der Polynomoperationen nutzen wir Funktionen, die für sequentielle Berechnungen einen Kompromiss zwischen Laufzeit und Speicherplatzbedarf darstellen.
Weil wir bei ECM für beide Continuations eine verschiedene Anzahl von Kurven berechnen müssen, um einen Faktor einer bestimmten Größenordnung zu finden, haben wir die Laufzeit eines Versuchs auf die notwendige Anzahl von Versuchen hochgerechnet. Damit haben wir eine Vergleichswert beider Continuations.

#Stellen	B	D	Zeit pro Versuch	Gesamtzeit
10	411	8 858	5 s	30 s
15	3 071	144 949	32 s	11 m 12 s
20	16 337	1 285 635	169 s	3 h 11 m 32 s
25	71 535	6 996 393	707 s	42 h 13 m 25 s
30	280 894	33 036 146	2 802 s	476 h 20 m 24 s
35	1 031 865	131 338 190	10 247 s	4613 h 59 m 47 s

Tabelle 6.6: Zeitbedarf ECM mit verbesserter Standard-Continuation

#Stellen	B	d_1	d_2	Zeit pro Versuch	Gesamtzeit
20	18 000	160	1 000	230 s	3 h 15 m 0 s
25	88 000	540	3 700	1 145 s	38 h 10 m 0 s
27	160 000	860	6 100	1 720 s	79 h 20 m 0 s
30	390 000	1 700	12 000	5 782 s	449 h 42 m 40 s
33	880 000	3 300	24 000	13 144 s	1643 h 0 m 0 s
35	1 000 000	5 000	38 000	20 138 s	3412 h 16 m 20 s

Tabelle 6.7: Zeitbedarf ECM mit FFT-Continuation

Kapitel 7

Erfolge

Wir haben uns mit unserer Implementierung von ECM an der Faktorisierung von Partitionszahlen und Cunningham-Zahlen beteiligt. Dabei haben wir Faktoren mit bis zu 41 Dezimalstellen gefunden. So konnten wir teilweise allein, teilweise zusammen mit dem Quadratischen Sieb viele Partitionszahlen komplett faktorisieren.

7.1 RSA-Zahlen

Die Firma „RSA Data Security" veranstaltet seit 1991 einen Wettbewerb zur Faktorisierung großer Zahlen, den „RSA Factoring Challenge". Dabei existieren zwei Listen von Zahlen, die RSA-Liste und die Partition-Liste.

In der RSA-Liste sind Zahlen aufgereiht, die genau zwei Primfaktoren haben. Diese Faktoren haben etwa gleiche Größe. Mit ECM haben wir wenig Chancen, eine Zahl dieser Liste zu faktorisieren. Die momentan kleinste Zahl in der Liste hat 130 Dezimalstellen, die Faktoren haben also ungefähr 65 Dezimalstellen. Wie wir bei den Parametern gesehen haben, ist dieser Bereich mit den momentan bekannten ECM-Varianten nicht zu erreichen. Die größte bisher faktorisierte RSA-Zahl ist RSA-129. Diese hat, wie der Name schon sagt, 129 Dezimalstellen. Sie wurde mit dem Quadratischen Sieb mit Hilfe einer weltweiten Kooperation im Mai 1994 faktorisiert.

Mit ECM können wir erfolgreich bei der Faktorisierung der Partitionszahlen beitragen. Die Faktorisierung einer Partitionszahl soll etwa den gleichen Schwierigkeitsgrad wie die Faktorisierung einer zufälligen Zahl der gleichen Größenordnung haben. Dadurch kann man den Fortschritt im Bereich der Faktorisierung erkennen.

Die Partitionszahl $p(n)$ ist die Anzahl aller Möglichkeiten, n als Summe natürlicher Zahlen darzustellen. Dabei wird jede Kombination nur einmal gezählt, Vertauschungen haben keinen Einfluß. Als Beispiel führen wir die Möglichkeiten von $p(7) = 15$ an.

Wir gliedern dabei nach der Anzahl der Summanden:

1: (7)

2: $(6+1), \quad (5+2), \quad (4+3)$

3: $(5+1+1), \quad (4+2+1), \quad (3+3+1), \quad (3+2+2)$

4: $(4+1+1+1), \quad (3+2+1+1), \quad (2+2+2+1)$

5: $(2+2+1+1+1), \quad (3+1+1+1+1)$

6: $(2+1+1+1+1+1)$

7: $(1+1+1+1+1+1+1)$

Um die Anzahl der Partitionszahlen zu beschränken, werden im Wettbewerb nur Partitionszahlen $p(n)$ für Primzahlen n faktorisiert. Abhängig von der Anzahl der Dezimalstellen der Partitionszahl erhält man für eine Komplettfaktorisierung Punkte. Am Ende jeden Quartals werden 1750 $ Preisgeld prozentual unter allen Teilnehmern mit Punkten aufgeteilt.
Die Liste der faktorisierten Zahlen wird regelmäßig aktualisiert, man kann sie per Email von *challenge-news@rsa.com* beziehen. Nähere Informationen erhält man per Email von *challenge-administrator@rsa.com*. Wir geben nun die Punktverteilungen der letzten Quartale an. Die folgende Aufstellung stellt einen Teil der oben erwähnten Liste dar.

```
Partition Challenge List:
-------------------------

        (1st Quarter, 1994)
                48 points, Bruce Dodson, Arjen K. Lenstra, $1218
                19 points, Factoring Factory Buchmann
                           (Franz-Dieter Berger, Johannes Buchmann,
                            Thomas F. Denny, Andreas M"uller,
                            Thomas Sosnowski), $482
                 1 point, Paul C Leyland, $25
                 1 point, Reynald Lercier, Francois Morain, $25
                --------------
                69 points total

        (2nd Quarter, 1994)
                30 points, Factoring Factory Buchmann
                           (Franz-Dieter Berger, Johannes Buchmann,
                            Thomas F. Denny, Andreas M"uller,
                            Thomas Sosnowski), $665
                28 points, Bruce Dodson, Arjen K. Lenstra, $620
                16 points, Magnus Y Alvestad, Arjen K. Lenstra, $354
                 4 points, Paul C Leyland, $89
                 1 point, Heinz-Bernd Eggenstein,
                          Factoring Factory Buchmann
                          (Franz-Dieter Berger, Johannes Buchmann,
                          Thomas F. Denny, Andreas M"uller,
```

```
                  Thomas Sosnowski), $22
         --------------
         79 points total

(3rd Quarter, 1994)
         12 points, Factoring Factory Buchmann
                    (Franz-Dieter Berger, Johannes Buchmann,
                     Thomas F. Denny, Andreas M"uller,
                     Thomas Sosnowski), $328
          8 points, Bruce Dodson, Arjen K. Lenstra,
                    Peter L. Montgomery, $219
         28 points, Bruce Dodson, Arjen K. Lenstra, $766
          8 points, Magnus Y Alvestad, Arjen K. Lenstra, $219
          8 points, Michael Graff, Arjen K. Lenstra,
                    Paul Leyland, $219
         --------------
         64 points total

(4th Quarter, 1994)
         18 points, Factoring Factory Buchmann
                    (Franz-Dieter Berger, Johannes Buchmann,
                     Thomas F. Denny, Andreas M"uller,
                     Thomas Sosnowski), $516
         23 points, Bruce Dodson, Arjen K. Lenstra, $660
         12 points, Arjen K. Lenstra, $344
          8 points, Scott Contini, Bruce Dodson, Arjen K. Lenstra,
                    Peter Montgomery, $230
         --------------
         61 points total

(1st Quarter, 1995)
         14 points, Bruce Dodson, Arjen K. Lenstra, $
         25 points, Factoring Factory Buchmann
                    (Franz-Dieter Berger, Johannes Buchmann,
                     Thomas F. Denny, Andreas M"uller,
                     Thomas Sosnowski), $
          8 points, Arjen K. Lenstra, $
          0 points, Raymond L. Jerome, $0
         --------------
         47 points total
```

Die letzte Angabe ist der Stand des Wettbewerbs am 22.03.1995.
Bei der Teilnahme an diesem Wettbewerb haben wir fast 90 Partitionszahlen teilweise zusammen mit dem Quadratischen Sieb faktorisiert. In der folgenden Tabelle geben wir alle Faktoren mit mehr als 33 Dezimalstellen an, die wir gefunden haben. Einige dieser Faktoren haben wir mit ECM mit der verbesserten Standard-Continuation gefunden, andere mit der FFT-Continuation.

Zahl	Faktor
p(14489)	2273696030977005519570565402015273771 (37)
p(15083)	14152669930737933243950680654260527 (35)
p(15329)	2222565653301339313658936859608420497867 (40)
p(15647)	84394719860368846600630163154101993088491 (41)
p(15727)	53493526911277747216041707018188163323 (38)
p(15923)	19465999333830490275285946921528097 (35)
p(16057)	149102709955929649828981929749040490471 (39)
p(16057)	17247806607650556918057245637345946590803 (41)
p(16073)	232040180339276024080841089644853 (33)
p(17239)	2965290528161283955599430579612201 (34)
p(17321)	52682446929394963173933498687452563 (35)
p(17333)	1264771634822348528899646580163483 (34)
p(18061)	20190112383094068571086009831559789 (35)
p(18397)	682027972473415704830399616516007 (33)
p(18443)	106975049563923102950357283192958763 (36)
p(18493)	4459069373454099623296858452791291 (34)
p(18637)	41665178037922593242291830626784777 (35)
p(18793)	45844419560978896644405788888790299 (35)
p(19373)	270802452690257205094049826614169913 (36)
p(19417)	11068493351185748319301464185571721 (35)
p(19699)	20887357092034499373209737347988379 (35)
p(20551)	235505013126622797185062510075070181389 (39)
p(20873)	117436485578607998335593041934839 (33)
p(21937)	1026959030034056105885882821297709(34)
p(22031)	629487314866522051873918124475770311 (36)
p(23539)	219708361595055316773735203384941 (33)
p(23887)	20418938729686795459795674946865023 (35)
p(23917)	166203370873277623804469829798347 (33)
p(24071)	4640736470766479458356364755924029 (34)
p(24371)	23249295690776630720048570898055453 (35)

Tabelle 7.1: Große gefundene Faktoren

Bei vielen von uns faktorisierten Zahlen war die Restzahl nach dem Finden eines großen Faktors prim. In der Beschreibung der verteilten Implementierung von ECM (siehe Abschnitt 5.3.3) haben wir vorgestellt, daß ECM auch terminiert, falls die Dezimallänge der Restzahl unter einer vorgegebenen Schranke liegt. Dieses Zusammenspiel von ECM und QS illustrieren wir an folgenden Beispielen:

p(15923) =		3377014432854388576584008735198693109569103195051266108589105372195406 8003957695523772359082855996049733438411790118797424004598496393237 (136)	
	=	$3^2 \cdot 7^2 \cdot 11$	(Probedivision)
	·	19465999333830490275285946921528097 (35)	(ECM)
	·	78787867102141936431606615607675035323984250487 (47)	(QS)
	·	453905655854145899354711077878465162255885535349633 (51)	(Rest)

p(24359) =		435446508814474257630570766073337736944681466189039008529587724024232 955820548686461845035605935047772937600963770299424790030704457138735 7218325125857896982934414158025 (169)	
	=	$5^2 \cdot 11$	(Probedivision)
	·	67953567882060178399224799 (26)	(ECM)
	·	153820860481879162656369427 (27)	(ECM)
	·	35735249152277912997384901354891 (32)	(ECM)
	·	50033720723013171752636785247278639 (35)	(QS)
	·	847256684536780665710354698595898645604332733043 (48)	Rest

Mit Hilfe des am Lehrstuhl Buchmann implementierten Programmes *eco* (siehe [LMMS94]) haben wir nachträglich zu jedem Faktor die Gruppenordnung der elliptischen Kurve berechnet, mit der der Faktor gefunden wurde. Dabei traten verschiedene Überraschungen auf:

- Bei der Faktorisierung von p(23893) erhielten wir als Gruppenordnung der elliptischen Kurve

  $$3515804135058305919105608206848 =$$
  $$2^9 \cdot 3 \cdot 7 \cdot 19 \cdot 197 \cdot 2729 \cdot 3251 \cdot 118603 \cdot 1050926441.$$

  Der Faktor wurde aber schon bei der Schranke $B = 600000$ in der 1. Phase gefunden. Bei anschließenden Tests fanden wir heraus, daß der größte Teiler der Ordnung des generierten Punktes genau 118603 war und nicht 1050926441.

- Die Gruppenordnung der elliptischen Kurve, mit der wir den 32-stelligen Faktor der Zahl p(16453) fanden, war

  $$11448152764230478854128771617872 =$$
  $$2^4 \cdot 3 \cdot 7^2 \cdot 13 \cdot 149 \cdot 2089 \cdot 6271 \cdot 6569 \cdot 20123 \cdot 26141 \cdot 55511.$$

  Damit kann man den Faktor schon mit Parametern für 22-stellige Faktoren finden.

- Bei den meisten durchgeführten Tests haben wir zur Generierung von elliptischer Kurve und einem Punkt auf dieser Kurve den 16-Generator (siehe Algorithmus 2.8) benutzt. Damit ist sicher, daß die Gruppenordnung für alle Primteiler p der Zahl N durch 16 teilbar ist. In vielen Fällen kommt der Faktor 2 in deutlich höheren Potenzen vor.

p(15647):

$$\begin{aligned}
p &= 8439471986036884660063016315410199308849 1\ (41) \\
E &= (\ 5927717929823909699981732027313980988607 4, \\
&\quad\ 707439737990784940760085955925590952153 97\ ) \\
|E\,(\mathbb{Z}/p\mathbb{Z})| &= 8439471986036884660073872279543050914611 2 \\
&= 2^{12} \cdot 3 \cdot 719 \cdot 1873 \cdot 8179 \cdot 26083 \cdot 52067 \cdot 179233 \\
&\quad \cdot 207061 \cdot 12371741
\end{aligned}$$

p(21011):

$$\begin{aligned}
p &= 5809827825770619693606853603\ (28) \\
E &= (\ 3745814261221944245063812212, \\
&\quad\ 1818411909647842218625686971\ ) \\
|E\,(\mathbb{Z}/p\mathbb{Z})| &= 5809827825770555420908802048 \\
&= 2^{12} \cdot 17 \cdot 41 \cdot 59 \cdot 3533 \cdot 19763 \cdot 57271 \cdot 8625559
\end{aligned}$$

Insgesamt haben wir bei den meisten faktorisierten Zahlen festgestellt, daß die Gruppenordnung fast immer schon mit Parametern für kleinere Faktorlängen gefunden worden wäre. Aus der Anzahl der Faktoren läßt sich aber noch keine allgemeine Regel ableiten, ob z.B. die Parameter zu hoch gewählt sind.

In der Tabelle 7.2 geben wir zusätzlich die beiden größten Teiler der Gruppenordnung an. In der letzten Spalte markieren wir die Zahlen mit FFT, die unter Anwendung optimaler Parameter nur mit der FFT-Continuation gefunden wurden. Gleichzeitig markieren wir Faktoren, die schon in der 1. Phase gefunden wurden. Dabei zeigen wir noch einige unserer Meinung nach interessante Zahlen auf.

7.2 Cunningham-Zahlen

Unter einer Cunningham-Zahl versteht man eine Zahl der Gestalt

$$a^n \pm 1, \qquad a, n \in \mathbb{N}.$$

Richard Brent und Herman Te Riele (siehe [BrRi92]) haben ein Projekt gestartet. Sie versuchen im erweiterten Cunningham-Projekt, alle Cunningham-Zahlen mit der Basis $13 \leq a \leq 100$ zu faktorisieren. Die Exponenten sind nicht beschränkt, momentan bewegt sich der Exponent n im Bereich bis zu 250. Die Zahlen sind auf die Dezimallänge 255 beschränkt. Sie veröffentlichen in unregelmäßigen Abständen die Liste der Teil- und Komplettfaktorisierungen.

An diesem Projekt beteiligen wir uns seit geraumer mit Zeit mit einer Implementierung des Quadratischen Siebs auf dem Hypercube (siehe [So95]). Mit ECM haben wir nur einige wenige Zahlen getestet. Bei diesen Tests haben wir einen 34-stelligen Faktor gefunden. Damit war die Zahl $18^{127}+1$ komplett faktorisiert.

$$
\begin{aligned}
18^{127}+1 = \; & 26278958442728535850178214481871092935104596386558534091289990249891 7\\
& 26574918898251040881733544118309641030313579023127051640293588186908 7\\
& 5518387168395009196033 \; (160)\\
= \; & 19 && \text{(Probedivision)}\\
\cdot \; & 326813520888984977 \; (18) && \text{(ECM)}\\
\cdot \; & 2224113911952795497 \; (19) && \text{(ECM)}\\
\cdot \; & 4017882819509660635131381483969043 \; (34) && \text{(ECM)}\\
\cdot \; & 47358750787555873799025064603194873102301946840475150471051816112447732\\
& 103831195229901921 \; (89) && \text{(Rest)}
\end{aligned}
$$

Zahl	Dezimalstellen	zweitgrößter Teiler	größter Teiler	Bem.
p(15329)	40	371 669	2 625 541	
p(15923)	35	237 467	472 193	1. Phase
p(16057)	39	3543 409	1 058 574 929	FFT
p(16057)	41	112 199	11 425 784 561	FFT
p(16073)	33	35 977	5 418 913	
p(17321)	35	258 241	291 419	1. Phase
p(17333)	34	147 517	1 666 279	
p(18061)	38	50 417	1 612 935 209	FFT
p(18397)	33	740 849	9 045 011	FFT
p(18493)	34	148 483	244 999	1. Phase
p(19373)	36	156 733	585 073	1. Phase
p(19417)	35	27 691	81 270 001	
p(20551)	39	549 547	28 247 207	
p(21211)	32	86 813	163 973	1. Phase
p(21937)	34	20 641	3 293 503	
p(24547)	31	19 559	151 897	1. Phase

Tabelle 7.2: Untersuchung der Gruppenordnung für gefundene Faktoren

Anhang A

Definitionen und Sätze

A.1. Satz (Fundamentalsatz der Algebra)
Jede von Eins verschiedene natürliche Zahl ist als Produkt endlich vieler Primzahlen darstellbar; diese Darstellung ist eindeutig, wenn man die in ihr vorkommenden Primzahlen der Größe nach ordnet.

Beweis: siehe [Bu88]. ■

A.2. Definition
Ist $(G; \cdot)$ *eine endliche Gruppe, dann heißt*

$$|G| := \#\{a \in G\} \qquad \textit{(Elementanzahl)}$$

Ordnung der Gruppe *(siehe [La80]).*

A.3. Definition
Ist $(G; \cdot)$ *eine endliche Gruppe mit neutralem Element 1 und* $a \in G$, *so heißt*

$$\mathrm{ord}_G(a) = \min\{r \in \mathbb{N}, a^r = 1\}$$

die **Ordnung des Elementes** a *(siehe [La80]).*

A.4. Satz
Ist $(G; \cdot)$ *eine endliche Gruppe, so ist die Ordnung* $\mathrm{ord}_G(a)$ *jedes* $a \in G$ *stets ein Teiler der Gruppenordnung* $|G|$.

Beweis: siehe [La80]. ■

A.5. Satz
Die Gruppenordnung einer elliptischen Kurve über dem Körper $\mathbb{Z}/p\mathbb{Z}, p \in \mathbb{P}$ *liegt im* **Hasse-Intervall**

$$p + 1 - 2\lfloor\sqrt{p}\rfloor \quad < \quad \left|E\left(\mathbb{Z}/_{p\mathbb{Z}}\right)\right| \quad < \quad p + 1 + 2\lfloor\sqrt{p}\rfloor.$$

Beweis: siehe [Si85]. ■

A.6. Satz (kleiner Fermatscher Satz)
Sei $p \in \mathbb{P}, a \in \mathbb{Z}$*. Dann gilt:*

$$a^p \equiv a \bmod p.$$

Für $a \in \mathbb{Z},\ p \nmid a$ *gilt:*

$$a^{p-1} \equiv 1 \bmod p.$$

Beweis: Einen elementaren Beweis finden wir in [Bu88]. ■

A.7. Definition
Die Funktion

$$\begin{aligned} \varphi : \mathbb{N} \quad &\rightarrow \quad \mathbb{N} \\ \varphi(n) \quad &:= \quad \#\{l \in \{1, \ldots, n-1\} : \mathrm{ggT}(l, n) = 1\} \end{aligned}$$

heißt **Eulersche** φ**-Funktion**. $\varphi(n)$ *ist die Anzahl der natürlichen Zahlen kleiner als* n*, die zu* n *teilerfremd sind.*

A.8. Definition
Die Matrix

$$V(x_0, x_1, \ldots, x_n) \quad = \quad \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}$$

heißt **Vandermonde-Matrix** *von* $x_0, x_1, \ldots, x_n$*.*

A.9. Satz
Ein Polynom vom Grad n ist durch die Funktionswerte an $n+1$ verschiedenen Stützstellen eindeutig bestimmt.

Beweis: siehe [CoLeRi89]. ■

A.10. Satz (Chinesischer Restsatz)
Sei

$$N = \prod_{i=1}^{r} p_i^{e_i}, \quad p_i \in \mathbb{P}, e_i \in \mathbb{N} \quad \forall\, i = 1, \ldots, r$$

Dann ist die Abbildung

$$\begin{array}{rcl} \left(\mathbb{Z}/N\mathbb{Z}\right) & \longleftarrow & \left(\mathbb{Z}/p_1^{e_1}\mathbb{Z}\right) \times \left(\mathbb{Z}/p_2^{e_2}\mathbb{Z}\right) \times \cdots \times \left(\mathbb{Z}/p_r^{e_r}\mathbb{Z}\right) \\ a & \longmapsto & \left(a \bmod p_1^{e_1}, a \bmod p_2^{e_2}, \ldots, a \bmod p_r^{e_r}\right). \end{array}$$

ein Isomorphismus.

Beweis: siehe [CoLeRi89]. ■

A.11. Definition
Sei $k \in \mathbb{N}$. Für ein festes $\alpha \in \mathbb{Z}$ heißt $g_{k,\alpha}$ **Dickson Polynom**, *falls $g_{k,\alpha}(x)$ die formale Identität*

$$g_{k,\alpha}\left(X + \frac{\alpha}{X}\right) = X^k + \frac{\alpha^k}{X^k}.$$

erfüllt.

A.12. Satz
Eine rekursive Beschreibung der Dickson Polynome wird folgendermaßen gegeben:

$$\begin{array}{rcl} g_{1,\alpha}(X) & = & X, \\ g_{2,\alpha}(X) & = & X^2 - 2\alpha, \\ g_{k+2,\alpha}(X) & = & X g_{k+1,\alpha}(X) - \alpha g_{k,\alpha}(X) \qquad (k \geq 1). \end{array}$$

Beweis: siehe [Mo92/2]. ■

Anhang B

Die Funktionen

B.1 Funktionen zum Primzahlsieben

int initprimes(unsigned int speicher, unsigned int min,
unsigned int max)

Ergebnis: 0, falls genügend Speicherplatz zur Verfügung gestellt wird und korrekt gesiebt wurde.
-1, sonst.

In der Variablen *speicher* muß beim Aufruf die Größe des Speicherplatzes in Byte angegeben werden, die mit den Primzahltafeln belegt werden darf. Abhängig von dieser Größe und der oberen Primzahlschranke *max* entscheidet die Funktion, ob auf einem oder mehreren Intervallen gesiebt wird. Sie siebt dann die Primzahlen bis zum Intervallende und setzt den Zeiger in das Bitfeld auf das Bit, das die Zahl *min* repräsentiert. Falls *min* gerade ist, wird der Zeiger auf das (*min*+1) repräsentierende Bit gesetzt.

Achtung: Diese Funktion muß immer zuerst aufgerufen werden, bevor die folgenden Funktionen aufgerufen werden dürfen.

unsigned int getprimes( )

Ergebnis:	nächste Primzahl,	falls noch nicht alle Primzahlen bis *max* aus der Primzahltafel ausgelesen sind.
	1,	falls der Zeiger am Ende der Bitliste steht.

Diese Funktion liefert erst nach der Initialisierung durch die Funktion *initprimes* vernünftige Ergebnisse, die kleinste zurückgegebene Primzahl ist 3. Falls beim Sieben auf mehreren Intervallen alle Primzahlen aus einem Intervall ausgelesen wurden, siebt die Funktion das nächste Intervall und liefert aus diesem die 1. Primzahl zurück. Falls die Primzahltafel komplett ausgelesen wurde, liefert die Funktion bei jedem weiteren Aufruf die Zahl 1.

int resetprimes(unsigned int min)

Ergebnis: -1, falls *min* größer als die mit der Funktion *initprimes* gesetzte obere Primzahlschranke ist.
0, falls alles korrekt abläuft.

Bei korrektem Ablauf setzt diese Funktion den Zeiger auf das Bit in der Bitliste, welches die Zahl *min* bzw. (*min*+1) für gerades *min* repräsentiert. Beim Arbeiten auf mehreren Intervallen wird gegebenenfalls das benötigte Intervall neugesiebt.

void kill_tables( )

Diese Funktion gibt den für die Primzahltafeln allokierten Speicherplatz frei.

B.2 Funktionen für elliptische Kurven

Wir verwenden in diesem Abschnitt zwei verschiedene Typen von langen Zahlen. Dies sind

- **bigint**: Lange Zahl, z.B. gefundener Faktor.
- **bigmod**: Modulo der zu faktorisierenden Zahl N reduzierte lange Zahl.

Wie in Abschnitt 5.2 vorgestellt, benötigen wir zum Rechnen mit Punkten auf elliptischen Kurven verschiedene Datentypen. Dies sind:

- **eccurve**: Elliptische Kurve.
- **ec_pointM**: Punkt auf elliptischer Kurve in Montgomery-Chudnowsky-Parametrisierung (siehe 5.2).
- **ec_pointW**: Punkt auf elliptischer Kurve in Weierstraß-Form.

Bei der Vorstellung der Funktionen geben wir die Deklarationen in C++-Syntax an. Dabei wenden wir bei den Aufrufparametern jeweils ein *call-by-reference* an. Die Parameter, die bei der Funktionsabarbeitung nicht geändert werden, haben wir mit *const* gekennzeichnet.

B.2.1 Speicherverwaltung

Wir haben alle einfachen Anwendungen bei der Speicherverwaltung mit Makros geschrieben, um im Programmablauf Funktionsaufrufe einzusparen.

ecalloc(eccurve C)

Allokiert Speicherplatz für die elliptische Kurve C.

ecfree(eccurve C)
Gibt den mit *ecalloc()* allokierten Speicher wieder frei.

mepalloc(ec_pointM P)
Allokiert Speicherplatz für den Punkt P auf einer elliptischen Kurve in Montgomery-Chudnowsky-Parametrisierung

mepfree(ec_pointM P)
Gibt den mit *mepalloc()* allokierten Speicher wieder frei.

wepalloc(ec_pointM P)
Allokiert Speicherplatz für den Punkt P auf einer elliptischen Kurve in Weierstraß-Parametrisierung

wepfree(ec_pointM P)
Gibt den mit *wepalloc()* allokierten Speicher wieder frei.

B.2.2 Arithmetik

Viele in diesem Abschnitt vorgestellte Funktionen liefern als Resultat einen Wert vom Typ *ECM_STATUS*. Dabei kann eine Variable vom Typ *ECM_STATUS* folgende Bedeutung haben:

no_factor_found : Beim Abarbeiten der Funktion sind alle Operationen korrekt ausgeführt worden, es wurde kein Faktor gefunden. Die Berechnung kann korrekt fortgesetzt werden.

factor_found : Ein echter Faktor der Zahl N wurde gefunden. Dieser wird jeweils in der Variablen *factor* zurückgegeben.

hopeless : Die Zahl N selbst ist bei einer ggT-Berechnung bzw. Invertierung als Faktor gefunden worden. Wir können die Berechnung auf dieser elliptischen Kurve abbrechen und einen neuen Durchlauf starten.

new_message : Dieser Fall tritt nur in der verteilten Version von ECM auf, falls bei der Ausführung der 1. oder 2. Phase ein neuer Job vom Master an einen Worker gesendet wird.

Alle Funktionen, die wir in diesem Abschnitt nun vorstellen und die als Resultat einen Wert vom Typ *ECM_STATUS* liefern, erzeugen die Ausgabe nach diesem Muster. Deshalb beschreiben wir nicht für jede Funktion explizit das mögliche Resultat.

**ECM_STATUS mecgen12(eccurve & E, ec_pointM & P,
bigint & factor)**

Diese Funktion generiert mit dem in Abschnitt 2.2.1 vorgestellten Algorithmus eine elliptische Kurve E und einen Punkt P auf dieser Kurve in Montgomery-Chudnowsky-Form. Die Gruppenordnung ist dabei modulo jedes Teilers von N durch 12 teilbar.

**ECM_STATUS mecgen16(eccurve & E, ec_pointM & P,
bigint & factor)**

Diese Funktion generiert mit dem in Abschnitt 2.2.2 vorgestellten Algorithmus eine elliptische Kurve E und einen Punkt P auf dieser Kurve in Montgomery-Chudnowsky-Form. Die Gruppenordnung ist dabei modulo jedes Teilers von N durch 16 teilbar.

**ECM_STATUS mecgen16_fix(eccurve & E, ec_pointM & P,
bigint & factor, int m)**

Im Algorithmus 2.8 wählen wir eine zufällige Zahl m, die die generierte Kurve bestimmt. In dieser Funktion geben wir m als Parameter mit, wir erhalten das gleiche Ergebnis wie in der Funktion *mecgen16*.

**void mepadd(ec_pointM & erg, const ec_pointM & P,
const ec_pointM & Q, const ec_pointM & Diff)**

erg = P + Q — Addition zweier Punkte in Montgomery-Chudnowsky-Form. Dabei muß für den Punkt Diff = P - Q gelten (siehe Abschnitt 2.1.2).

**void mepadd2P(ec_pointM & erg, const ec_pointM & P,
const bigmod & a_2)**

erg = 2· P — Verdopplung eines Punktes in Montgomery-Chudnowsky-Form. a_2 hat den durch den Parameter a der elliptischen Kurve als a_2 = (a+2)/4 bestimmten Wert (siehe Abschnitt 2.1.2).

**void mepmul(ec_pointM & erg, const ec_pointM & P,
const bigmod & a_2, unsigned int k)**

erg = k · P — Multiplikation des Punktes P mit dem Skalar k in Montgomery-Chudnowsky-Parametrisierung.

Wir wenden den binären Algorithmus (siehe 2.3) zur Multiplikation an, a_2 hat die gleiche Bedeutung wie in der Funktion *mepadd2P*.

**void prac(ec_pointM & erg, const ec_pointM & P,
const bigmod & a_2, unsigned int p, unsigned int k)**

erg = k · P — Multiplikation des Punktes P mit dem Skalar k in Montgomery-Chudnowsky-Parametrisierung.

Die Berechnung führen wir mit Hilfe der Lukas-Ketten nach Algorithmus 2.6 aus. Nach den dort angegebenen Bedingungen muß p ein Teiler von k sein.

ECM_STATUS mecfindfactor(const eccurve & E, ec_pointM & P, bigint & factor, unsigned long B, unsigned long C)

ECM-Algorithmus ohne Continuation, zur Multiplikation des Punktes P mit einem Skalar verwenden wir die Funktion *mepmul*.

ECM_STATUS mecfind_prac(const eccurve & E, ec_pointM & P, bigint & factor, unsigned long B, unsigned long C)

ECM-Algorithmus ohne Continuation, zur Multiplikation des Punktes P mit einem Skalar verwenden wir die Funktion *prac*.

ECM_STATUS mwectrans(eccurve & E, const ec_pointM MP, ec_pointW & WP, bigint & factor)

Transformation von Punkt MP und elliptischer Kurve E aus der Montgomery-Chudnowsky-Form auf die Weierstraß-Form. Beim Aufruf ist E eine Kurve in Montgomery-Chudnowsky-Form, nach der Abarbeitung der Funktion enthält sie die Parameter der Kurve in Weierstraß-Form.

ECM_STATUS wepadd2P(ec_pointW & erg, const ec_pointW & P, bigint & factor)

erg $= 2 \cdot$ P — Verdopplungung eines Punktes in Weierstraß-Form.
WP muß ein endlicher Punkt sein, es wird keine Überprüfung durchgeführt.

ECM_STATUS wepaddPQ(ec_pointW & erg, const ec_pointW & P, const ec_pointW & Q, bigint & factor)

erg = P + Q — Addition zweier Punkte in Weierstraß-Form.
P und Q müssen endliche, verschiedene Punkte sein, wiederum wird diese Bedingung nicht überprüft.

ECM_STATUS wepadd(ec_pointW & erg, const ec_pointW & P, const ec_pointW & Q, bigint & factor)

erg = P + Q — Addition zweier Punkte in Weierstraß-Form.
Diese Funktion kann zur Addition von Punkten verwendet werden, wenn keine Information über diese Punkte bekannt ist. Dabei werden Tests auf infiniten Punkt oder Gleichheit der Punkte durchgeführt.

ECM_STATUS wepmul(ec_pointW & erg, const ec_pointW & P, bigint & factor, unsigned int k)

erg $= k \cdot$ P — Multiplikation des Punktes P mit dem Skalar k in Weierstraß-Parametrisierung.

Die Berechnung wird mit schneller Exponentiation ausgeführt.

ECM_STATUS wepmul2(ec_pointW & erg, const ec_pointW & P, bigint & factor, unsigned int k)

erg = k · P — Multiplikation des Punktes P mit dem Skalar k in Weierstraß-Parametrisierung.

Die Berechnung wird mit Hilfe der schnellen Exponentiation unter Anwendung des Morain-Tricks (siehe [MoOl90]) ausgeführt.

ECM_STATUS ecm_stdcont2(ec_pointW & WP, bigint & factor, unsigned long B, unsigned long C)

Verbesserte Standard-Continuation, siehe Algorithmus 1.10.

B.2.3 Hilfsfunktionen

ecsto(eccurve E1, eccurve E2)

Makro zum Kopieren der Kurve E2 auf die Kurve E1.

mepsto(ec_pointM P1, ec_pointM P2)

Makro zum Kopieren des Punktes P2 auf den Punkt P1 für Punkte P1 und P2 in Montgomery-Chudnowsky-Form.

wepsto(ec_pointW P1, ec_pointW P2)

Makro zum Kopieren des Punktes P2 auf den Punkt P1 für Punkte P1 und P2 in Weierstraß-Form.

BOOLEAN mep_on_curve(const eccurve & E, const ec_pointM & P)

Ergebnis: 0, falls der Punkt P nicht auf der Kurve E liegt.
1, falls der Punkt P wahrscheinlich auf der Kurve liegt.

Diese Funktion prüft, ob der Punkt P in Montgomery-Chudnowsky-Form auf der elliptischen Kurve E liegt. Weil die y-Koordinate des Punktes nicht bekannt ist, können wir nur mit Sicherheit sagen, daß der Punkt nicht auf einer Kurve liegt.

BOOLEAN wep_on_curve(const eccurve & E, const ec_pointW & P)

Ergebnis: 1, falls der Punkt P auf der Kurve E liegt.
0, sonst.

Diese Funktion prüft, ob der Punkt P in Weierstraß-Form auf der elliptischen Kurve E liegt.

BOOLEAN wepcomp(const ec_pointW & P1, const ec_pointW & P2)

Ergebnis: 0, falls (P1 == P2) für Punkte in Weierstraß-Form
1, sonst.

BOOLEAN wepnegcomp(const ec_pointW & P1,
const ec_pointW & P2)

Ergebnis: 0, falls (P1 == -P2) für Punkte in Weierstraß-Form
1, sonst.

B.3 Die FFT-Continuation

In diesem Abschnitt stellen wir noch die speziellen Funktionen vor, die wir in der FFT-Continuation verwenden. Da diese mit Hilfe der Langzahlarithmetik GF implementiert sind, verwenden wir hier einen Division-Handler, der beim Fehlschlagen einer Invertierung von der Arithmetik selbst aufgerufen wird. Deshalb haben die nun vorgestellten Funktionen keine Rückgabewerte, die den Berechnungszustand kennzeichnen.

In den Aufrufen verwenden wir mehrmals den Datentyp *vector(bigmod)*. Dieser enthält ein Feld von Einträgen vom Typ *bigmod* und einen Parameter, der die Anzahl der Einträge bestimmt.

ec_point_W ist ein Punkt in Weierstraß-Parametrisierung, wie wir ihn in allen C++-Versionen verwenden. Der Aufbau ist wie in Abschnitt 5.2 beschrieben.

void simult_inv(vector(bigmod) & xinv,
const vector(bigmod) & x)

Alle Einträge im Vektor x werden modulo der zu faktorisierenden Zahl N, die als globaler Modul abgespeichert ist, invertiert. Dabei verwenden wir den ggT-Trick zur Invertierung (siehe [Be93]).

void my_vec_addPQ(ec_point_W* erg, const ec_point_W* wp1,
vector(bigmod) & x)

Bei der Generierung der pseudozufälligen Vielfachen mit Hilfe arithmetischer Progressionen (siehe 4.4.3) müssen wir in einem Vektor von Punkten auf der gleichen elliptischen Kurve jeweils benachbarte Punkte addieren, d.h. wir berechnen $erg[i] = wp1[i] + wp1[i+1]$ für alle Punkte im Feld $wp1$. Die Variablen $wp1$ und erg sind also beide Zeiger auf ein Feld von Punkten, die Länge des Feldes wird durch die Länge des mitgegebenen Vektors x bestimmt. Die Einträge im Vektor $wp1$ müssen endliche Punkte sein.

Dabei wenden wir zur Invertierung den ggT-Trick an. Zur Ausführung der Invertierung nutzen wir die Funktion *simult_inv*. Durch Übergabe einer Referenz auf den Vektor x sparen wir in jedem Aufruf die Allokierung und spätere Deallokierung des zur parallelen Invertierung benötigten Vektors ein.

void ecm_fftcont(ec_point_W & P, int d1, int d2, int k)

In dieser FFT-Continuation verwenden wir zur Generierung der pseudozufälligen Vielfachen Vielfache der Form $2^{k/2}$ und 3^k. Die Parameter d1, d2, und k sind wie in Abschnitt 4.4.2 angegeben. d1 bestimmt die Größe der Menge 1, d2 die Anzahl der Elemente der zweiten Menge. Zur Berechnung verwenden wir das Dickson Polynom $g_{k,0}$, dieses hat Grad k.

void ecm_fftcont2(ec_point_W & P, int d1, int d2, int k)

Zur Berechnung der zufälligen Vielfachen wenden wir in dieser Funktion den Algorithmus zur Berechnung arithmetischer Progressionen an (siehe 4.4.3). Alle Parameter sind wie in der Funktion *ecm_fftcont*. In unseren Programmen verwenden wir für den Parameter k den Wert 12. Dieser stellt nach Betrachtungen bei der Parameterberechnung einen guten Kompromiss zwischen der Laufzeit der FFT-Continuation und den Möglichkeiten zum Finden eines Faktors dar.

Anhang C

Listings

C.1 Die Include-Files

C.1.1 Die C-Version

```
#ifndef _ecm_h
#define _ecm_h
#define TRIAL           1000000       /* Schranke fuer Probedivision */
#define SP_PRIMES        200000       /* Speicherplatz fuer Primtafel */
#define MAXDEZSTEL         1000       /* Maximale Laenge der Zahl */
#define MAXSTRATEGY          50       /* Maximale Anzahl verschiedener Jobs */
#define REST                  1       /* Restlaenge */

#define ECM_PARAMETERS_STD "ecm-parameters-std" /* Parameter verb. Std.Cont.*/
#define ECM_PARAMETERS_FFT "ecm-parameters-fft" /* Parameter fuer FFT-Cont. */
#define STRATEGYFILES      "ecm-strategy"       /* Strategie-File */
#define ECM_HELP           "ecm.help"           /* Hilfe-Files */
#define ECM_HELP_FFT       "ecm.fft.help"
#define STOP_ECM           "stop-ecm"           /* Abbruch-File */
#endif
```

```
#ifndef _icontrib_h
#define _icontrib_h
#define USE_THIS_LOT_OF_MACROS
#include <iint.h>
#include <imod.h>
#include <imacros.h>

void IasintmpowI ( register pInteger, register const int,
                   register const Integer*, register const Integer *);
BOOLEAN Iisprime( register pInteger, register int);
int Ijacobi(register pInteger, register pInteger);
BOOLEAN init_trialdiv( unsigned long, unsigned long, unsigned long);
unsigned long trialdiv( register pInteger);
#endif
```

```
#ifndef _libmisc_h
#define _libmisc_h
#include <stdio.h>
#include <ecm.h>

#define MAXFILENAMELEN   100  /* Maximale Laenge von File-Namen */

float mips();
```

```
int fexist(char*);
FILE* fopena(char*, char*);
                                   /* Einlesen der Strategie */
int read_strategy( char strat_file[MAXFILENAMELEN], int strategy[MAXSTRATEGY]);
                                   /* Einlesen der Parameter */
int read_parameters( char para_file[MAXFILENAMELEN], int, unsigned long*,
                     unsigned long*, int*);
int read_parameters_fft( char para_file[MAXFILENAMELEN], int, unsigned long*,
                         unsigned long*, unsigned long*, int*);
                                   /* Berechnung der Jobs */
int job_planing( char para_file[MAXFILENAMELEN], int strat[MAXSTRATEGY],
                 long buf[MAXSTRATEGY][5], int);
int job_planing_fft( char para_file[MAXFILENAMELEN], int strat[MAXSTRATEGY],
                     long buf[MAXSTRATEGY][5], int);

void start_time(char*);               / Timing - Funktionen */
void stop_time(char*);
void print_time(int);
int start_timing(int timer[2]);
int get_timing(int timer[2]);
int cont_timing(int timer[2]);
char* strtime(char*, unsigned long, char);
#endif



#ifndef _libint_h
#define _libint_h
#include <stdio.h>

extern double sqrt(double);

int gcd(int, int);
void init_random();
unsigned int pow_int(register unsigned int, register int);

                                      /* Funktionen zur Primzahlberechnung */
int initprimes(unsigned int, unsigned int, unsigned int);
unsigned int getprimes();
int resetprimes(unsigned int);
void kill_tables();
#endif



#ifndef _libelliptic_h
#define _libelliptic_h

#include <icontrib.h>
extern ModulusType n_mod;

/* result of several functions :
          factor_found:    great!
          no_factor_found: go on with computing
          no_job:          no current job
          new_message:     new job arrived
          hopeless:        gcd(denominator, n) = n => Curve is useless */
typedef enum{factor_found, no_factor_found, no_job, new_message, hopeless}
        ECM_STATUS;

                                /* elliptic curves */
struct ECCURVE { Minteger a, b; };
typedef struct ECCURVE *eccurve;

#define ecalloc(E)    { E = (eccurve) malloc (sizeof(struct ECCURVE));\
                        cM(&E->a, &n_mod); cM(&E->b, &n_mod); }
#define ecfree(E)     { dM(&E->a); dM(&E->b); free(E); }
#define ecsto(E1, E2) { MasM(&E1->a, &E2->a); MasM(&E1->b, &E2->b); }

  /* points in the weierstrass model */
```

```
struct EC_POINTW { Minteger x, y; BOOLEAN O; eccurve C; };
typedef struct EC_POINTW *ec_pointW;

#define weptest0(P)   ( P->O == TRUE )
#define wepset0(P)    ( P->O = TRUE  )
#define wepunset0(P)  ( P->O = FALSE )

#define wepalloc(P)  { P = (ec_pointW) malloc (sizeof(struct EC_POINTW));\
                       cM(&P->x, &n_mod); cM(&P->y, &n_mod); }
#define wepfree(P)   { dM(&P->x); dM(&P->y); free(P); }
#define wepsto(P, Q) { P->C = Q->C; \
                       if ( weptest0(Q) )    wepset0(P); \
                       else { MasM(&P->x, &Q->x); MasM(&P->y, &Q->y);\
                       wepunset0(P);}}

              /* points in the montgomery model */
struct EC_POINTM { Minteger x, z; };
typedef struct EC_POINTM *ec_pointM;

#define meptest0(P) ( Meq0(&P->x) && Meq0(&P->z) ) ;

#define mepalloc(P)  { P = (ec_pointM) malloc (sizeof(struct EC_POINTM));\
                       cM(&P->x, &n_mod); cM(&P->z, &n_mod); }
#define mepfree(P)   { dM(&P->x); dM(&P->z); free(P); }
#define mepsto(P, Q) { MasM(&P->x,&Q->x); MasM(&P->z,&Q->z); }


          /* Funktionen */
ECM_STATUS test_factor( register pInteger, Minteger *);

ECM_STATUS mecgen12( eccurve, ec_pointM, register pInteger);
ECM_STATUS mecgen16( eccurve, ec_pointM, register pInteger);
ECM_STATUS mecgen16_fix( eccurve, ec_pointM, register pInteger, int);

void mepadd( ec_pointM, ec_pointM, ec_pointM, ec_pointM);
void mepadd2P( ec_pointM, ec_pointM, Minteger);
void mepmul( ec_pointM, ec_pointM, Minteger, unsigned int);
void prac( ec_pointM, ec_pointM, Minteger, unsigned int, unsigned int);
BOOLEAN mep_on_curve( eccurve, ec_pointM);

ECM_STATUS wepadd2P( ec_pointW, ec_pointW, register pInteger);
ECM_STATUS wepaddPQ( ec_pointW, ec_pointW, ec_pointW, register pInteger);
ECM_STATUS wepadd( ec_pointW, ec_pointW, ec_pointW, register pInteger);
ECM_STATUS wepmul( ec_pointW, ec_pointW, register pInteger, unsigned int);
ECM_STATUS wepmulI( ec_pointW, ec_pointW, register pInteger, Integer);
ECM_STATUS wepmul2( ec_pointW, ec_pointW, register pInteger, unsigned int);
ECM_STATUS wepmulI2( ec_pointW, ec_pointW, register pInteger, Integer);

BOOLEAN wep_on_curve( eccurve, ec_pointW);
BOOLEAN wepcomp( ec_pointW, ec_pointW);
BOOLEAN wepnegcomp( ec_pointW, ec_pointW);

ECM_STATUS mecfindfactor( eccurve, ec_pointM, register pInteger,
                          unsigned long, unsigned long);
ECM_STATUS mecfind_prac( eccurve, ec_pointM, register pInteger,
                         unsigned long, unsigned long);
ECM_STATUS mwectrans( eccurve, ec_pointM, ec_pointW, register pInteger);
ECM_STATUS ecm_stdcont2( ec_pointW, register pInteger, unsigned long,
                         unsigned long);

/* Funktionen auf Vektoren */
eccurve *ecvalloc(long);
void ecvfree(eccurve*, long);
ec_pointW *wepvalloc(long);
void wepvfree(ec_pointW*, long);
void wepvsto(ec_pointW*, ec_pointW*, long);

ECM_STATUS wepvadd2P( ec_pointW*, ec_pointW*, long, Minteger*,
      register pInteger);
ECM_STATUS wepvaddPQ( ec_pointW*, ec_pointW*, ec_pointW*, long,
```

```
                      Minteger*, register pInteger);
ECM_STATUS wepvadd( ec_pointW*, ec_pointW*, ec_pointW*, long,
                    Minteger*, register pInteger);
ECM_STATUS wepvmul( ec_pointW*, ec_pointW*, long, Minteger*,
                    register pInteger, unsigned int);
ECM_STATUS wepvmulI( ec_pointW*, ec_pointW *, long, Minteger*,
                     register pInteger, Integer);
#endif
```

C.1.2 Die C++-Version

```
#ifndef EC_POINT_H
#define EC_POINT_H

extern "C" {
#include <libint.h>
}
#include <GF.h>

class ec_curve
{
  ZZ_p a, b;

public:
  ec_curve()  { }
  ec_curve(const ZZ_p &la, const ZZ_p &lb) { a = la; b = lb; }
  ~ec_curve()  { }

  void assign(const ZZ_p &la, const ZZ_p &lb) { a = la; b = lb; }
  ZZ_p & A() { return a; }
  ZZ_p & B() { return b; }

  friend ostream & operator << (ostream &out, const ec_curve & me)
  {
    out << "( " << me.a << ", " << me.b << ") \n";
    return out;
  }
};


class ec_point_W
{
  ZZ_p x, y;
  ec_curve  *curve;
  char is_0;

 public:

  ec_point_W() {}
  ec_point_W(const ec_curve & we) { is_0 = 1; curve = (ec_curve *) &we;}
  ec_point_W(const ec_curve &we, const ZZ_p &lx, const ZZ_p &ly)
  {
    x = lx; y = ly; is_0 = 0;
    curve =(ec_curve *) &we;
  }
  ec_point_W(const ec_point_W &wp)
  {
    if ( !(is_0 = wp.is_0) )
    {
      x = wp.x; y = wp.y;
      curve = wp.Curve();
    }
  }
  ~ec_point_W() {}

  int is_zero() const { return is_0; }
  int is_negative(const ec_point_W &) const;
  void negate();
  ec_curve * Curve() const { return curve; }

  ZZ_p & X() { return x; }
```

```
  ZZ_p & Y() { return y; }

  void assign_one()  { is_0 = 0; }
  void assign_zero() { is_0 = 1; }
  void assign(const ec_point_W &);
  void assign(const ZZ_p &, const ZZ_p &);

  friend int operator == (const ec_point_W &, const ec_point_W &);

  friend void add(ec_point_W &, const ec_point_W &, const ec_point_W &);
  friend void addPQ(ec_point_W &, const ec_point_W &, const ec_point_W &);
  friend void multiply_by_2(ec_point_W &, const ec_point_W &);
  friend void multiply(ec_point_W &, const ec_point_W &, int);

                      /* Funktionen auf Vektoren */
  friend void vec_multiply_by_2(ec_point_W*, const ec_point_W*, vector(ZZ_p)&);
  friend void vec_addPQ(ec_point_W*, const ec_point_W*, const ec_point_W*,
                      vector(ZZ_p)&);
  friend void vec_multiply(ec_point_W *, const ec_point_W *, int, int);
  friend void my_vec_addPQ(ec_point_W*, const ec_point_W*, vector(ZZ_p) &);

  friend void ecm_fftcont(ec_point_W&, int, int, int);
  friend void ecm_fftcont2(ec_point_W&, int, int, int);

  friend ostream & operator <<(ostream &, const ec_point_W &);
};
#endif
```

C.2 Die Funktionen

C.2.1 Funktionen zum Primzahlsieben

```
/*
   Diese Funktionen berechnen die Primzahlen zwischen zwei anzugebenden
   Schranken. In Abhaengigkeit vom Speicherplatz siebe ich dabei auf einem
   bzw. nacheinander auf mehreren Intervallen.

 Funktionen :

   initprimes - Initialisiert das Array fuer die Primzahlen und siebt das
                erste  Intervall
   getprimes -  liefert bei jedem Aufruf eine Primzahl und setzt den
                Primzahlzeiger weiter, siebt bei geringer Speichergroesse
                auf folgendem Intervall neue Primzahlen aus
   resetprimes - setzt Zeiger auf Primzahlen an passende Stelle, gegebenfalls
                wird nachgesiebt
   kill_tables - loescht den allokierten Speicherplatz
   sieve_array - siebt alle Vielfache der Primzahlen zwischen 3 und 31 mit
                 Masken aus
   set_maske - initialisiert die Masken

   `sieve_array' und `set_maske' sind Funktionen, die nur intern aus den
   Funktionen `initprimes', `getprimes' und `resetprimes' aufgerufen werden
   und fuer den Benutzer nicht von Belang sind .

   ACHTUNG : Die Primzahl 2 wird nicht zurueckgegeben.
             Bei Aufruf der Funktionen 'getprimes` oder 'resetprimes` ohne
             vorherige Initialisierung mit 'initprimes` stuerzt das Programm
             ab, bzw. es gibt nur Muell zurueck .

   Def: int initprimes(speicher, us, os)
        unsigned int speicher // maximale Speichergroesse in Bytes
                     us       // kleinste gew"unschte Primzahl
                     os       // obere Schranke fuer Primzahlen
   Res:    0  ,  bei korrektem Aufbau des Siebfeldes
          -1  ,  bei Fehler

   Def: unsigned int getprimes()
```

```
   Res: naechste Primzahl , falls vorhanden
         1        , sonst

   Def: int resetprimes(start)
        unsigned int start    // neue untere Schranke fuer Primzahlen
   Res:   0  ,  bei korrektem Zur"ucksetzen
          1  ,  bei Fehler

   Def: void kill_tables() - gibt den allokierten Speicherplatz frei

   Andreas Mueller, 19.07.1993
*/

#include <malloc.h>
#include <libint.h>

void sieve_array(unsigned int start);
void set_maske();

#define ulong unsigned long
#define uint unsigned int

ulong BIT = 0x80000000;

static ulong * table;          /* Siebarray fuer Primzahlen */
static ulong * table2;         /* Primzahlen von 3 bis zur Wurzel der
                                  groessten gewuenschten Primzahl*/
static uint * over;           /* Offsets beim Sieben auf mehr. Intervallen */

static uint dim_ar;           /* Groesse des Siebintervalls */
static uint feld, feld_abs, pl;     /* Zeiger auf Stelle im Siebarray */
static uint min, max, p_max, prime ;

            /* Masken zum Sieben der Primzahlen  zwischen 3 und 31 */
static ulong maske35 , maske7, maske11, maske13, maske17;
static ulong maske19, maske23, maske29, maske31 ,maske_i;
static char flag;
static uint space;

int initprimes(uint speicher, uint us, uint os)
{
  uint h, j, k, step, i;

  min = us; max = os; space = speicher;

  p_max = (uint) sqrt( (double)max ) >> 1;
  h = (uint) ( 500 + (p_max >> 3) + (p_max<<2) );

  if( speicher<((max>>4)+500) && speicher<h )    return(-1);

  if( speicher< ((max>>4)+500) )
  { dim_ar = (speicher - h) >> 2; flag = 1;  }
  else
  { dim_ar = max >> 6; flag = 0;  }

  prime = dim_ar << 6;

  if((table=(ulong*) calloc (dim_ar, sizeof(ulong))) == NULL)  return (-1);

  if (flag)
  {
    h = (p_max>>5)+1 ;
    if( (over=(uint*) calloc (h<<5, sizeof(uint))) == NULL )  return (-1);
    if( (table2=(ulong*) calloc (h, sizeof(ulong))) == NULL ) return (-1);
  }

  maske35 = 0x5b4cb699; maske7  = 0xfdfbf7ef; maske11 = 0xfdffbff7;
  maske13 = 0x7ffbffdf; maske17 = 0xffdfffef; maske19 = 0xfffeffff;
  maske23 = 0xdfffffbf; maske29 = 0xffefffff; maske31 = 0xfffdffff;

  table[0] = 0x76d32d26;
  sieve_array(1);

  for (i = 17;i < p_max; i++)
  {
```

```
    j =  2 * i + 3;
    if( table[j>>6] & (BIT>>((j & 0x0000003f)>>1)) )
    {
      step = j << 1;
      for (k = j*j; k < prime; k += step)
      {
        pl = (k & 0x0000003f) >> 1;
        feld = k >> 6;
        table[feld] = table[feld] & ~(BIT >> pl);
      }
      if( flag )        over[i] = k - prime ;
    }
  }

  if( flag )
  {
    for (i = 0; i < h ; i++)   table2[i] = table[i];
    if ( (min > prime) && flag )  set_maske();
  }

  feld = min >> 6; feld_abs = feld;
  pl = (min & 0x03f) >> 1;
  maske_i = BIT >> pl;
  return 0;
}

uint getprimes()
{
  uint j, k, step, i = 0;

 st:
  while ( (feld < dim_ar) && ((feld_abs<<6)<max) )
  {
    while ( pl < 32 )
    {
      pl++;
      if ( table[feld] & maske_i )
      {
        maske_i = maske_i >> 1;
        return( (pl<<1) + (feld_abs<<6) - 1 );
      }
      maske_i = maske_i >> 1;
    }
    pl = 0; feld_abs++; feld++;
    maske_i = 0x80000000;
  }

  if ( flag && (feld_abs<(max>>6)) )
  {
    sieve_array(0);

    for (i = 17; i < p_max; i++)
    {
      j = 2 * i + 3;
      if ( table2[j>>6] & (BIT>>((j & 0x03f)>>1)) )
      {
        step = j << 1 ;
        for (k = over[i]; k < prime; k += step )
        {
          pl = (k & 0x0000003f) >> 1 ;
          feld = k >> 6 ;
          table[feld] = table[feld] & ~(BIT >> pl) ;
        }
      }
      over[i] = k - prime ;
    }

    feld = 0; pl = 0;
    maske_i = 0x80000000;

    goto st;
  }
  return 1;
```

```
}

int resetprimes(uint us)
{
  int i, h;
  min = us;

  if ( min > max )   return(1);

  feld_abs = min >> 6; feld = feld_abs;
  pl = (us & 0x0000003f) >> 1;
  maske_i = BIT >> pl;

  if ( flag )
  {
    free(table); free(table2); free(over);
    initprimes(space, min, max);
  }
  return(0);
}

void kill_tables()
{
  free(table);
  if (flag)   { free(table2); free(over);  }
}

void sieve_array(uint start)
{
  uint i = start ;

  while ( i < dim_ar )
  {
    table[i++] = maske35 & maske7 & maske11 & maske13 & maske17 & \
                 maske19 & maske23 & maske29 & maske31 ;

    maske35 = maske35 << 2 ;
    maske35 = ((maske35 & 0xc0000000) >> 30 ) | maske35 ;

    maske7  = maske7 << 4 ;
    maske7  = ((maske7 & 0xf0000000) >> 28 ) | maske7 ;

    maske11 = maske11 << 10 ;
    maske11 = ((maske11 & 0xffc00000) >> 22) | maske11 ;

    maske13 = maske13 << 6 ;
    maske13 = ((maske13 & 0xfc000000) >> 26 ) | maske13 ;

    maske17 = maske17 << 15 ;
    maske17  = ((maske17 & 0xfffe0000) >> 17 ) | maske17 ;

    maske19 = maske19 << 13 ;
    maske19  = ((maske19 & 0xfff80000) >> 19 ) | maske19 ;

    maske23 = maske23 << 9 ;
    maske23  = ((maske23 & 0xff800000) >> 23 ) | maske23 ;

    maske29 = maske29 << 3 ;
    maske29  = ((maske29 & 0xe0000000) >> 29 ) | maske29 ;

    maske31 = maske31 << 1 ;
    maske31  = ((maske31 & 0x80000000) >> 31 ) | maske31 ;
  }
}

void set_maske()
{
  uint i, k, j, h;

  maske35 = 0x96d32da6; maske7  = 0xefdfbf7e; maske11 = 0xfbff7fef;
  maske13 = 0xfdffefff; maske17 = 0xff7fffbf; maske19 = 0xffbffff7;
  maske23 = 0xffefffff; maske29 = 0xfffdffff; maske31 = 0xfffeffff;
```

```
  k = min >> 6;
  j = k % 15;
  for (i = 0; i < j; i++)
  {
    maske35 = maske35 << 2;
    maske35 = ((maske35 & 0xc0000000) >> 30 ) | maske35;
  }
  j = k % 7;
  for (i = 0; i < j; i++)
  {
    maske7  = maske7 << 4;
    maske7  = ((maske7 & 0xf0000000) >> 28 ) | maske7;
  }
  j = k % 11;
  for (i = 0; i < j; i++)
  {
    maske11 = maske11 << 10;
    maske11 = ((maske11 & 0xffc00000) >> 22 ) | maske11;
  }
  j = k % 13;
  for (i = 0; i < j; i++)
  {
    maske13 = maske13 << 6;
    maske13 = ((maske13 & 0xfc000000) >> 26 ) | maske13;
  }
  j = k % 17;
  for (i = 0; i < j; i++)
  {
    maske17 = maske17 << 15;
    maske17  = ((maske17 & 0xfffe0000) >> 17 ) | maske17;
  }
  j = k % 19;
  for (i = 0; i < j; i++)
  {
    maske19 = maske19 << 13;
    maske19  = ((maske19 & 0xfff80000) >> 19 ) | maske19;
  }
  j = k % 23;
  for (i = 0; i < j; i++)
  {
    maske23 = maske23 << 9;
    maske23  = ((maske23 & 0xff800000) >> 23 ) | maske23;
  }
  j = k % 29;
  for (i = 0; i < j; i++)
  {
    maske29 = maske29 << 3;
    maske29  = ((maske29 & 0xe0000000) >> 29 ) | maske29;
  }
  j = k % 31;
  for (i = 0; i < j; i++)
  {
    maske31 = maske31 << 1;
    maske31  = ((maske31 & 0x80000000) >> 31 ) | maske31;
  }

  k = min & ~0x0000003f;

  for (i = 17; i < p_max; i++)
  {
    j =  2 * i + 3 ;
    if( table2[j>>6] & (BIT>>((j & 0x0000003f)>>1)) )
    {
      h = (( k / j ) + 1 ) * j - k;
      if (!(h&1))   h += j;
      over[i] = h;
    }
  }
}
#undef ulong
#undef uint
```

C.2.2 Elliptische Kurven und Punkte

```
/*
   mecgen16 -- creates an elliptic curve E in the montgomery model
               and a point P on it.

   Def: ECM_STATUS mecgen16(E, P, factor)
        ECM_STATUS mecgen16_fix(E, P, factor, m)
        eccurve E;                 // curve in the montgomery model
        ec_pointM P;               // point on curve E in montgomery model
        register pInteger factor;  // possibly found factor
        int m;                     //
   Res: no_factor_found   , if no factor of n was found
        factor_found      , if a factor was found
        hopeless          , if gcd(n, ...) == n
   Alg: A.O.L. Atkin, F. Morain: Finding Suitable Curves For The Elliptic Curve
                                 Method Of Factorization / Jan 1992
   Andreas Mueller , 25.08.1993
*/

#include <libelliptic.h>

extern ModulusType n_mod;

ECM_STATUS mecgen16(eccurve E, ec_pointM P, register pInteger factor)
{
  int m;

  init_random(); m = random();
  printf(" m = %d " , m); fflush(stdout);

  return ( mecgen16_fix(E, P, factor, m));
}


ECM_STATUS mecgen16_fix(eccurve E, ec_pointM P, register pInteger factor, int m)
{
  ECM_STATUS condition = no_factor_found;
  Minteger one, h1, h2, h3, d, alpha;
  Integer help;
  eccurve we;
  ec_pointW wp;

  ecalloc(we); wepalloc(wp);
  cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h3, &n_mod);
  cM(&d, &n_mod); cM(&alpha, &n_mod); cM(&one, &n_mod);
  cIasint(&help, 1); MasI(&one, &help);

  Iasint(&help, -8); MasI(&we->a, &help);           /* Initialisierung */
  Iasint(&help, -32); MasI(&we->b, &help);
  Iasint(&help, 12); MasI(&wp->x, &help);
  Iasint(&help, 40); MasI(&wp->y, &help);
  wp->C = we; wepunset0(wp);
  if ( (condition = wepmul(wp, wp, factor, m)) == factor_found )  goto ende;

  Iasint(&help, 9); MasI(&h1, &help);             /* computation of alpha */
  MasMmiM(&h2, &wp->x, &h1);       /* alpha = ((wp->y+25)/(wp->x-9)+1)^-1 */
  if ( !MasinvM(&h3, &h2) )
  {
    condition = test_factor(factor, &h2);
    goto ende;
  }

  Iasint(&help, 25); MasI(&h1, &help);
  MplasM(&h1, &wp->y);
  MmuasM(&h1, &h3);
  MplasM(&h1, &one);
  if ( !MasinvM(&alpha, &h1) )
  {
    condition = test_factor(factor, &h1);
    goto ende;
  }
```

```
 MasM(&h1, &alpha);                   /* d = 2alpha(4alpha+1)/(8alpha^2+1) */
 MmuasM(&h1, &h1);
 MplasM(&h1, &h1);
 MplasM(&h1, &h1);
 MplasM(&h1, &h1);
 MmiasM(&h1, &one);

 if ( !MasinvM(&h2, &h1) )
 {
   condition = test_factor(factor, &h1);
   goto ende;
 }

 MasMplM(&h1, &alpha, &alpha);
 MplasM(&h1, &h1);
 MplasM(&h1, &one);
 MmuasM(&h1, &alpha);
 MplasM(&h1, &h1);
 MasMmuM(&d, &h1, &h2);

 Iasint(&help, 4); MasI(&h1, &help);        /* computation of h2 = 4^-1 */
 MasinvM(&h2, &h1);

 MasMplM(&h1, &d, &d);                      /* computation of X = -(2d-1)/4 */
 MmiasM(&h1, &one);
 MmuasM(&h1, &h2);
 Ias0(&help); MasI(&h2, &help);
 MasMmiM(&wp->x, &h2, &h1);

 MasM(&h1, &d);                      /*  computation of the initial point */
 MmiasM(&h1, &one);
 MmuasM(&h1, &h1);
 if ( !MasinvM(&h2, &h1) )
 {
   condition = test_factor(factor, &h1);
   goto ende;
 }

 MasM(&h1, &d);
 MmuasM(&h1, &h1);
 MasMplM(&h3, &wp->x, &d);
 MmiasM(&h3, &h1);
 MasMmuM(&P->x, &h3, &h2);
 MasM(&P->z, &one);

 MasM(&h1, &d);                     /*   transformation of the curve     */
 MmiasM(&h1, &one);
 MmuasM(&h1, &d);
 MplasM(&h1, &h1);

 if ( !MasinvM(&h2, &h1) )
 {
   condition = test_factor(factor, &h1);
   goto ende;
 }

 MasMplM(&h1, &d, &d);
 MmiasM(&h1, &one);
 MmuasM(&h1, &h1);
 MmuasM(&h1, &h2);
 MmuasM(&h1, &h1);
 MmiasM(&h1, &one);
 MmiasM(&h1, &one);
 MasM(&E->a, &h1);
 MasM(&E->b, &one);

ende:
 wepfree(wp); ecfree(we);
 dM(&alpha); dM(&h1); dM(&h2); dM(&h3); dM(&one); dI(&help);

 return (condition);
```

```
}



/*C
   mecgen12 -- creates an elliptic curve E in the montgomery model
               and a point P on it.

   Def: ECM_STATUS mecgen12(E, P, factor)
        eccurve E;              // curve in the montgomery model
        ec_pointM P;            // point on curve E in montgomery model
        register pInteger factor;  // possibly found factor
   Res: no_factor_found   , if no factor of n was found
        factor_found      , if a factor was found
        hopeless          , if gcd(n, ...) == n
   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , ch. 5.1

   Andreas Mueller , 25.08.1993
C*/

#include <libelliptic.h>

extern ModulusType n_mod;

ECM_STATUS mecgen12(eccurve E, ec_pointM P, register pInteger factor)
{
   ECM_STATUS condition = no_factor_found;
   int r;
   Minteger u, h1, h2, h3;
   Integer s;

   cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h3, &n_mod);

   init_random(); r = random();

   cIasint(&s, r); MasI(&h1, &s);            /* Berechnung von u */
   MasM(&h2, &h1);
   MmuasM(&h2, &h2);
   Iasint(&s, 6); MasI(&h3, &s);
   MmuasM(&h1, &h3);
   MplasM(&h2, &h3);

   if ( !MasinvM(&h3, &h2) )
   {
     condition = test_factor(factor, &h2);
     goto end;
   }
   MasMmuM(&u, &h1, &h3);

   Iasint(&s, 4);  MasI(&h1, &s);            /* Berechnung des Punktes */
   if ( !MasinvM(&h1, &h1) )
   {
     condition = test_factor(factor, &h1);
     goto end;
   }

   MasMplM(&h2, &u, &u);
   MplasM(&h2, &u);
   MasMmuM(&P->x, &h2, &h1);

   IAS1(&s); MasI(&h1, &s);
   MasM(&E->b, &h1);
   MasM(&P->z, &h1);

   MasM(&h2, &u);              /* ggt - Test */
   MmuasM(&h2, &h2);
   MasM(&h3, &h2);
   MplasM(&h3, &h3);
   MplasM(&h3, &h3);
   MplasM(&h3, &h3);
   MplasM(&h3, &h2);
   MmiasM(&h3, &h1);
```

```
   MmiasM(&h2, &h1);
   MmuasM(&h3, &h2);
   MmuasM(&h3, &u);

   if ( !MasinvM(&h2, &h3) )
   {
     condition = test_factor(factor, &h3);
     goto end;
   }

   MasM(&h1, &u);            /* Berechnung der Kurve */
   MmuasM(&h1, &h1);
   MasM(&h2, &h1);
   MmuasM(&h1, &u);
   MplasM(&h1, &h1);
   MplasM(&h1, &h1);
   MasinvM(&h1, &h1);
   MasM(&h3, &h2);
   MmuasM(&h2, &h2);

   Iasint (&s, -3); MasI(&u, &s);
   MmuasM(&h2, &u);
   MmuasM(&h3, &u);
   MplasM(&h3, &h3);
   MplasM(&h2, &h3);
   MplasM(&h2, &E->b);
   MasMmuM(&E->a, &h2, &h1);

 end:
   dM(&h1); dM(&h2); dM(&h3); dM(&u); dI(&s);

   return (condition);
}



/*
   mepadd - computes the sum of two points P and Q in the
            montgomery model given the difference P-Q

   Def: void mepadd(Erg, P, Q, Diff)
        ec_pointM P, Q,      // points on the same curve in montgomery model
                Diff,        // Diff = P - Q
                Erg;         // result: Erg = P + Q

        extern ModulusType n_mod; // modul for using Montgomery-Multiplication
                             of libI-library

   Res: none

   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , chap. 4.11.2

   Andreas Mueller , 13.03.1995
*/

#include <libelliptic.h>

extern ModulusType n_mod;

void mepadd(ec_pointM Erg, ec_pointM P, ec_pointM Q, ec_pointM Diff)
{
   Minteger t1, t2, h3, h4;

   cM(&t1, &n_mod); cM(&t2, &n_mod); cM(&h3, &n_mod); cM(&h4, &n_mod);

   MasMmiM(&t1, &P->x, &P->z);               /* computation of t1   */
   MasMplM(&h3, &Q->x, &Q->z);
   MmuasM(&t1, &h3);

   MasMplM(&t2, &P->x, &P->z);               /* comptutation of t2 */
   MasMmiM(&h3, &Q->x, &Q->z);
   MmuasM(&t2, &h3) ;
```

```
   MasMplM(&h3, &t1, &t2);                     /* new x-coordinate  */
   MmuasM(&h3, &h3);
   MasMmuM(&h4, &h3, &Diff->z);

   MasMmiM(&h3, &t1, &t2);                     /* new z-coordinate  */
   MmuasM(&h3, &h3);
   MasMmuM(&Erg->z, &h3, &Diff->x);

   MasM(&Erg->x, &h4);
   dM(&t1); dM(&t2); dM(&h3); dM(&h4);
}



/*
   mepadd2P - computes 2*P for a point P in the montgomery model

   Def: void mepadd2P(Erg, P, a_2)
        ec_pointM P,               // point on curve E in montgomery model
                  Erg;             // result: Erg = 2 * P
        register pInteger a_2;     // constant for a fixed curve E
        extern ModulusType n_mod;  // modul for using Montgomery-Multiplication
                              of libI-library
   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , chap. 4.11.1

   Andreas Mueller , 15.03.1995
C*/

#include <libelliptic.h>

extern ModulusType n_mod;

void mepadd2P(ec_pointM Erg, ec_pointM P, Minteger a_2)
{
   Minteger h1, h2, h3;

   cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h3, &n_mod);

   MasMplM(&h1, &P->x, &P->z);               /*  (x+z)^2  */
   MmuasM(&h1, &h1);
   MasMmiM(&h2, &P->x, &P->z);               /*  (x-z)^2  */
   MmuasM(&h2, &h2);

   MasMmuM(&Erg->x, &h1, &h2);               /* new x-coordinate   */

   MmiasM(&h1, &h2);                         /*  4 * x * z   */
   MasMmuM(&h3, &h1, &a_2);                  /*  4 * x * z * a_2  */
   MplasM(&h3, &h2);
   MasMmuM(&Erg->z, &h1, &h3);               /* new z-coordinate      */

   dM(&h1); dM(&h2); dM(&h3);
}



/*
   mepmul - computes m*P for a point P on a curve E in
            the montgomery model

   Def: void mepmul(Erg, P, a_2, m)
        unsigned int m;       // odd scalar to compute m * P
        ec_pointM P,          // point on curve E in montgomery model
                Erg;          // result: Erg = m * P
        Minteger a_2;         // constant for a fixed curve E
   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , chap. 4.11.3

   Andreas Mueller , 15.03.1995
*/

#include <libelliptic.h>
```

```
extern ModulusType n_mod;

void mepmul(ec_pointM Erg, ec_pointM P, Minteger a_2, unsigned int m)
{
  ec_pointM P1, P2, Erg1;
  unsigned int i, r;
  unsigned long bit;

  mepalloc(P1); mepalloc(P2); mepalloc(Erg1);

  mepsto(P1, P);                    /*  Initialisierung : P1 =  P   */
  mepadd2P(P2, P1, a_2);            /*     P2 = 2*P                 */

  r = intlog(m);                    /*  r = erstes gesetztes Bit in m */
  bit = 0x00000001 << r;            /*  bit = 2^r  */

   for (i = r-1; i > 0; i--)
   {
      mepadd(Erg1, P1, P2, P);       /*  Erg1 = P1 + P2 mit P1-P2 = P  */
      bit >>= 1;
      if (bit & m)                   /*  i.tes Bit von m ist 1 */
      {
        mepadd2P(P2, P2, a_2);       /*    P2 = 2 * P2   */
        mepsto(P1, Erg1);            /*   P1  = P1 + P2  */
      }
      else                           /* i.tes Bit von m ist 0  */
      {
        mepadd2P(P1, P1, a_2);       /*   P1 = 2 * P1   */
        mepsto(P2, Erg1);            /*   P2  = P1 + P2  */
      }
   }
   mepadd(Erg, P1, P2, P);
   mepfree(P1); mepfree(P2); mepfree(Erg1);
}



/*
   prac - computes m*P for a point P on a curve E in the montgomery model
          using Lucas-chains

   Def: void mepmul(Erg, P, a_2, m, p)
        unsigned int m,   // scalar to compute m * P
                     p;   // p divisor of m
        ec_pointM P,      // point on curve E in montgomery model
                  Erg;    // result: Erg = m * P
        Minteger a_2;     // constant for a fixed curve E
   Alg: Evaluating Recurrences Of Form X_{m+n} = f(X_m, X_n, X_{m-n}) Via
        Lucas Chains by  Peter L. Montgomery / Jan 1992

   Andreas Mueller , 15.03.1995
*/

#include <libelliptic.h>

void prac( ec_pointM erg, ec_pointM mp, Minteger  a_2,
           unsigned int p, unsigned int m)
{
  unsigned long d, r, e, h;
  float alpha = 1.618034;
  ec_pointM A, B, C, H, T;

  mepalloc(A); mepalloc(B); mepalloc(C); mepalloc(H); mepalloc(T);
  mepsto(A, mp);

  while ( !(m&1) )
  {
    m >>= 1;
    mepadd2P(A, A, a_2);
  }

  while ( !(m%3) )
```

```
{
  m /= 3;
  mepadd2P(H, A, a_2);
  mepadd(A, A, H, A);
}
d = m;

while ( d != 1 )
{
  if ( p > d ) p = d;
  r = (unsigned long) ( (float)p / alpha + 0.5);
  r = ( d / p ) * r;
  e = 2 * r - d; d = 2 * d - 3 * r;

  mepsto(C, A);                                  /*  A = 2A, B = 3A, C = A */
  mepadd2P(A, A, a_2);
  mepadd(B, A, C, C);

  while ( d != e )
  {
    if ( d < e )                              /* Vertauschen: A = B, B = A */
    {
      h = d; d = e; e = h;
      mepsto(H, A); mepsto(A, B); mepsto(B, H);
    }

    if ( (d-e) < (e/4) )
    {
      h = (int) (d+e)/3 ;

      if ( (d + e) == (3 * h) )        /* Fall1: A = 2A+B, B = A+2B, C = % */
      {
        d -= h; e -= h;
        mepadd(T, A, B, C);
        mepadd(H, T, A, B);
        mepadd(B, T, B, A);
        mepsto(A, H);
        goto end_while;
      }
      else
      {
        if ( !((d - e) % 6) )           /* Fall2: A = 2A, B = A+B, C = % */
        {
          d = ( d - e ) / 2;
          mepadd(B, A, B, C);
          mepadd2P(A, A, a_2);
          goto end_while;
        }

        d = d - e;                      /* Fall2a: A = %, B = A+B, C = B */
        mepadd(T, A, B, C);
        mepsto(C, B);
        mepsto(B, T);
        goto end_while;
      }
    }

    if ( d <= ( 4 * e ) )                /* Fall3: A = %, B = A+B, C = B */
    {
      d = d - e;
      mepadd(H, A, B, C);
      mepsto(C, B);
      mepsto(B, H);
      goto end_while;
    }

    if ( !((d-e)&1) )                    /* Fall4: A = 2A, B = A+B, C = % */
    {
      d = ( d - e ) / 2;
      mepadd(B, A, B, C);
      mepadd2P(A, A, a_2);
      goto end_while;
```

```
      }
      if ( !(d&1) )                    /* Fall5: A = 2A, B = %, C = 2A-B */
      {
        d >>= 1;
        mepadd(C, A, C, B);
        mepadd2P(A, A, a_2);
        goto end_while;
      }
      if ( !(d%3) )                    /* Fall6: A = 3A, B = 3A+B, C = B */
      {
        d = d / 3 - e ;
        mepadd2P(H, A, a_2);
        mepadd(T, A, B, C);
        mepadd(A, H, A, A);
        mepadd(H, H, T, C);
        mepsto(C, B);
        mepsto(B, H);
        goto end_while;
      }
      if ( !((d+e) % 3 ) )             /* Fall7: A = 3A, B = 2A+B, C = % */
      {
        d = ( d + e ) / 3 - e;
        mepadd(H, A, B, C);
        mepadd(B, H, A, B);
        mepadd2P(T, A, a_2);
        mepadd(A, A, T, A);
        goto end_while;
      }
      if ( !(( d-e) % 3 ) )          /* Fall8: A = 3A, B = A+B, C = 2A-B */
      {
        d = ( d - e ) / 3;
        mepadd(H, A, B, C);
        mepadd(C, A, C, B);
        mepsto(B, H);
        mepadd2P(T, A, a_2);         /*  X3(A)  */
        mepadd(A, A, T, A);
        goto end_while;
      }
      if ( !(e&1) )                       /* Fall9: A = %, B = 2B, C = A */
      {
        e >>= 1 ;
        mepadd(C, C, B, A);
        mepadd2P(B, B, a_2);
        goto end_while;
      }
    end_while: ;
    }
    mepadd(A, A, B, C);                  /* Ende:  A = A+B, B = %, C = % */
  }

  mepsto(erg, A);
  mepfree(A); mepfree(B); mepfree(C); mepfree(T); mepfree(H);
}


/*
   mep_on_curve - tests, whether a Point P in the montgomery model is
                  on a curve E

   Def:  BOOLEAN mep_on_curve(E, P)
         eccurve E;            // curve in the montgomery model
         ec_pointM P;          // point on curve E in montgomery model
   Res:  FALSE    , Point P is not on curve E
         TRUE     , Point P is possibly on curve E
```

```
   Andreas Mueller , 25.08.1993
*/

#include <libelliptic.h>

extern ModulusType n_mod;

BOOLEAN mep_on_curve(eccurve E, ec_pointM P)
{
   int sum;
   Minteger h1, h2, h3;
   Integer h, n;

   cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h3, &n_mod);
   cI(&n); cIasint(&h, 1);

   MasinvM(&h2, &E->b);
   MasM(&h1, &P->x);
   MmuasM(&h2, &h1);
   MmuasM(&h1, &h1);
   MasMmuM(&h3, &P->x, &E->a);
   MplasM(&h3, &h1);
   MasI(&h1, &h);
   MplasM(&h3, &h1);
   MmuasM(&h3, &h2);

   IasM(&h, &h3);
   IasMod(&n, &n_mod);

   sum = Ijacobi(&h, &n);

   dM(&h1); dM(&h2); dM(&h3);
   dI(&h); dI(&n);
   if ( sum==1 )    return ( TRUE );
   else             return ( FALSE );
}



/*
   mecfindfactor - first phase of elliptic curve method
   mecfind_prac  - first phase of elliptic curve method using Lucas-chains
                   for computing k * P

   Def:  ECM_STATUS mecfindfactor(E, P, factor, B, C)
         ECM_STATUS mecfind_prac(E, P, factor, B, C)
         unsigned long B,          // upper limit for primes
                       C;          // upper limit for powers of primes
         register pInteger factor;   // possibly found factor

         ec_curve E;              // curve in the montgomery model
         ec_point_M P;            // point on curve E in montgomery model

         extern ModulusType n_mod;  // modul for using Montgomery-Multiplication
                                       of libI-library
   Res:  no_factor_found , if no factor of n was found
         factor_found    , if a factor was found
         hopeless        , no hope
   Warning: This function can't be used before sieving primes with the
         function "initprimes", but there is no check.
   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , chap. 5

   Andreas Mueller , 15.03.1995
*/

#include <libelliptic.h>
#include <libint.h>

extern ModulusType n_mod;

ECM_STATUS mecfindfactor ( eccurve E, ec_pointM P, register pInteger factor,\
                           unsigned long B, unsigned long C)
{
```

```
  ECM_STATUS condition = no_factor_found;
  Minteger a_2, m1, m2;
  unsigned long k, prim;
  register pInteger h;
  double log_C = log2( (double) C);
  int sqrt_C = (int) sqrt( (double)C ), count = 0;

  h = ncIasint(2); cMasI(&m1, h, &n_mod);
  cM(&m2, &n_mod); cM(&a_2, &n_mod);
                                /*  computation of a_2 = (E->a+2)/4 :  */
                                /*  a_2 is used for doubling a point   */
  MasM(&a_2, &E->a);            /*  and is fixed for a fixed curve E   */
  MplasM(&a_2, &m1);
  Iasint(h, 4);
  MasI(&m1, h);
  MasinvM(&m2, &m1);
  MmuasM(&a_2, &m2);
  IAS1(h);

  for (k = 0; k < 31; k++)
    mepadd2P(P, P, a_2);

                     /*  successive computation of    P = prim^k * P     */
                     /*  for all primes prim = 3, ..., B1   and          */
                     /*  k integer with prim^k <= C && prim^(k+1) > C    */

  while ( (prim = getprimes()) <= sqrt_C )    /*   loop: exponent k > 1   */
  {
    k = (unsigned long)( log_C/(double)log2((double)prim) );
    mepmul(P, P, a_2, pow_int(prim, k));
  }

  do                                           /*   loop: exponent k = 1   */
  {
    mepmul(P, P, a_2, prim);

    if (count++ > 1000)
      if ( MasinvM(&m1, &P->z) )
      {
        count = 0;
        MasI(&P->z, h);
        MmuasM(&P->x,&m1);
      }
      else
      {
        condition = test_factor(factor, &P->z);
        goto end;
      }
  }
  while ( (prim = getprimes()) < B );

   if ( MasinvM(&m1, &P->z) )
   {
     MasI(&P->z, h);
     MmuasM(&P->x, &m1);
   }
   else
     condition = test_factor(factor, &P->z);

 end:
  dM(&a_2); dM(&m1); dM(&m2); ddI(h);
  return(condition);
}


ECM_STATUS mecfind_prac ( eccurve E, ec_pointM P, register pInteger factor,\
                          unsigned long B, unsigned long C)
{
  ECM_STATUS condition = no_factor_found;
  Minteger a_2, m1, m2;
  unsigned long k, prim;
  register pInteger h;
  double log_C = log2((double) C);
```

```
  int sqrt_C = (int) sqrt( (double)C ), count = 0;

  h = ncIasint(2); cMasI(&m1, h, &n_mod);
  cM(&m2, &n_mod); cM(&a_2, &n_mod);

  MasM(&a_2, &E->a);                /*  computation of a_2 = (E->a+2)/4 :  */
  MplasM(&a_2, &m1);                /*  a_2 is used for doubling a point   */
  Iasint(h,4); MasI(&m1, h);        /*  and is fixed for a fixed curve E   */
  MasinvM(&m2, &m1);
  MmuasM(&a_2, &m2);
  IAS1(h);

  for (k = 0; k < 31; k++)
    mepadd2P(P, P, a_2);
                          /*  successive computation of    P = prim^k * P   */
                          /*  for all primes prim = 3, ..., B1   and        */
                          /*  k integer with prim^k <= C && prim^(k+1) > C  */
  while ( (prim = getprimes()) <= sqrt_C )        /*   loop: exponent k > 1 */
  {
    k = (unsigned long)( log_C/(double)log2((double)prim) );
    prac(P, P, a_2, prim, pow_int(prim, k));
  }

  do                                          /*   loop: exponent k = 1   */
  {
    prac(P, P, a_2, prim, prim);

    if (count++ > 1000)
      if ( MasinvM(&m1, &P->z) )
      {
        count = 0;
        MasI(&P->z, h);
        MmuasM(&P->x,&m1);
      }
      else
      {
        condition = test_factor(factor, &P->z);
        goto end;
      }
  }
  while ( (prim = getprimes()) < B );

  if ( MasinvM(&m1, &P->z) )
  {
    MasI(&P->z, h);
    MmuasM(&P->x, &m1);
  }
  else
    condition = test_factor(factor, &P->z);

 end:
  dM(&a_2); dM(&m1); dM(&m2); ddI(h);

  return(condition) ;
}



/*
   mwectrans - transforms elliptic curves and points in the montgomery
               model to the weierstrass model

   Def:  ECM_STATUS mwectrans(e, mp, wp, factor)
         eccurve e;             // elliptic curve
         ec_pointM mp;          // point on curve e in montgomery model
         ec_pointW wp;          // point on curve e in the weierstrass model
         register pInteger factor;    // possibly found factor
   Res: factor_found      , if a factor of n was found
        no_factor_found   , the transformation is ok
        hopeless          , otherwise
   Alg: 'An Implementation of the Elliptic Curve Integer Factorization Method'
         by Wieb Bosma, Arjen K. Lenstra , chap. 4.2
```

```
   Andreas Mueller , 05.11.1993
*/

#include <libelliptic.h>

extern ModulusType n_mod;

ECM_STATUS mwectrans ( eccurve e, ec_pointM mp, ec_pointW wp,\
                       register pInteger factor)
{
  ECM_STATUS condition = no_factor_found;
  Minteger h1, h2, h3, t, inv_3, inv_t;
  Integer help;

  cIasint(&help, 3);
  cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h3, &n_mod); cM(&t, &n_mod);
  cM(&inv_3, &n_mod); cM(&inv_t, &n_mod);

  MasI(&h3, &help);
  MasinvM(&inv_3, &h3);

  IAS1(&help); MasI(&h3, &help);
  MasM(&h2, &mp->x);                  /* t = (mp.x^2+mp.x*e.a+1)*mp.x) */
  MmuasM(&h2, &h2);
  MasMmuM(&h1, &mp->x, &e->a);
  MasMplM(&t, &h2, &h1);

  MplasM(&t, &h3);
  MmuasM(&t, &mp->x);

  if ( !MasinvM(&inv_t, &t) )                         /* inv_t = t^-1 */
  {
    condition = test_factor(factor, &t);
    goto end;
  }

  MasM(&wp->y, &inv_t);                               /* wp.x = inv_t */
  MasMmuM(&h3, &inv_3, &e->a);
  MplasM(&h3, &mp->x);
  MasMmuM(&wp->x, &h3, &inv_t);           /* wp.x = (e.a/3 + mp.x)/t */

  MasM(&h2, &e->a);
  MmuasM(&h2, &h2);
  MasMmuM(&h1, &h2, &inv_3);

  MasI(&h3, &help);
  MmiasM(&h3, &h1);

  MasM(&h1, &inv_t);
  MmuasM(&h1, &h1);
  MasMmuM(&t, &h3, &h1);                   /* e.a = (1-(e.a^2/3)/t^2 */

  MmuasM(&h1, &inv_t);
  MmuasM(&h1, &inv_3);
  MasM(&h3, &inv_3);
  MmuasM(&h3, &h3);
  MmuasM(&h2, &h3);
  MasMplM(&h2, &h2, &h2);

  MasI(&h3, &help);
  MmiasM(&h2, &h3);
  MmuasM(&h2, &e->a);

  MasM(&e->a, &t);
  MasMmuM(&e->b, &h2, &h1);       /* e.b = (2(e.a^3 / 9) - e.a)/t^3  */
  wp->C = e;                                        /* wp->C = e */

 end:
  dM(&h1) ; dM(&h2) ; dM(&h3) ; dM(&t) ;
  dM(&inv_3); dM(&inv_t); dI(&help);

  return (condition) ;
}
```

```
/*
   wepadd2P - computes 2*P for a point P on a curve E in the weierstrass model

   Def: ECM_STATUS wepadd2P(Erg, P, factor)
        WEPOINT P;                 // point in weierstrass model
        WEPoint Erg;               // result: Erg = 2 * P
        register pInteger factor;  // possibly found factor
   Res: no_factor_found , if no factor of n was found
        factor_found    , if a factor was found
        hopeless        , no hope

   Andreas Mueller ,  01.07.1993
*/

#include <libelliptic.h>

extern ModulusType n_mod;

ECM_STATUS wepadd2P(ec_pointW Erg, ec_pointW P, register pInteger factor)
{
  Minteger h, h1, h2;
  ECM_STATUS condition = no_factor_found;

  cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h, &n_mod);

  MasMplM(&h, &P->y, &P->y);
  if ( !MasinvM(&h2, &h) )                        /*  h2 = (2y1)^(-1) */
  {
    condition = test_factor(factor, &h);
    goto ENDE;
  }

  MasM(&h1, &P->x);
  MmuasM(&h1, &h1);
  MasMplM(&h, &h1, &h1);
  MplasM(&h1, &h);
  MplasM(&h1, &P->C->a);
  MmuasM(&h1, &h2);                            /* h = (3x1^2+a)/(2y1) */
  MasM(&h, &h1);
  MmuasM(&h1, &h1);
  MasMplM(&h2, &P->x, &P->x);
  MasMmiM(&h2, &h1, &h2);                  /* Erg.x = h2 = 2P.x - h^2 */
  MasMmiM(&h1, &P->x, &h2);
  MmuasM(&h1, &h);
  MasMmiM(&Erg->y, &h1, &P->y);      /* Erg.y = -P.y+ h * (P.x-erg.x) */
  MasM(&Erg->x, &h2);

 ENDE:
  dM(&h); dM(&h1); dM(&h2);

  return(condition);
}
```

```
/*
   wepaddPQ - computes P+Q for points P, Q on a curve E in the weierstrass model

   Def: ECM_STATUS wepaddPQ(Erg, P, Q, factor)
        ec_pointW P, Q;             // points in weierstrass model
        ec_pointW Erg;              // result: Erg = P + Q
        register pInteger factor;   // possibly found factor
   Res: no_factor_found , if no factor of n was found
        factor_found    , if a factor was found
        hopeless        , gcd(denominator, n) = n => Curve is useless

   Andreas Mueller ,  02.07.1993
*/

#include <libelliptic.h>

extern ModulusType n_mod;
```

```
ECM_STATUS wepaddPQ ( ec_pointW Erg, ec_pointW P, ec_pointW Q, \
                      register pInteger factor)
{
  Minteger h, h1, h2;
  ECM_STATUS condition = no_factor_found;

  cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h, &n_mod);

  MasMmiM(&h, &Q->x, &P->x);
  if ( !MasinvM(&h2, &h) )                    /* h2 = (Q.x-P.x)^(-1) */
  {
    condition = test_factor(factor, &h);
    goto ENDE;
  }

  MasMmiM(&h1, &Q->y, &P->y);                   /* computation of Erg.x */
  MasMmuM(&h, &h2, &h1);
  MasM(&h1, &h);
  MmuasM(&h1, &h1);
  MmiasM(&h1, &P->x);
  MasMmiM(&h2, &h1, &Q->x);

  MasMmiM(&h1, &P->x, &h2);                    /* computation of Erg.y */
  MmuasM(&h1, &h);
  MasMmiM(&Erg->y, &h1, &P->y);
  MasM(&Erg->x, &h2);

 ENDE:
  dM(&h); dM(&h1); dM(&h2);

  return(condition);
}



/*
   wepadd  - addition of points on elliptic curves

   Def: ECM_STATUS wepadd(Erg, P, Q, factor)
        ec_pointW P, Q,               // points in weierstrass model
                  Erg;                // result: Erg = P + Q
        register pInteger factor;     // possibly found factor
   Res: no_factor_found , if no factor of n was found
        factor_found    , if a factor was found
        hopeless        , gcd(denominator, n) = n ==> Curve is useless

   possible : P == Q == Erg

   Andreas Mueller , 01.07.1993
*/

#include <libelliptic.h>

ECM_STATUS wepadd ( ec_pointW Erg, ec_pointW P, ec_pointW Q,\
                    register pInteger factor)
{
   ECM_STATUS condition = no_factor_found;
   Minteger h, h1, h2;

   if ( weptest0(P) )                                     /* P == 0  */
   {
     if ( Q != Erg )  wepsto(Erg, Q);
     return condition;
   }

   if ( weptest0(Q) )                                     /* Q == 0  */
   {
     if ( P != Erg ) wepsto(Erg, P);
     return condition;
   }

   if ( !wepnegcomp(P, Q) )                               /* P == -Q */
   {
     wepset0(Erg);
```

```
    return condition;
  }

  cM(&h1, &n_mod); cM(&h2, &n_mod); cM(&h, &n_mod);

  if ( !wepcomp(P, Q) )                                   /* P == Q  */
  {
    MasMplM(&h, &P->y, &P->y);
    if ( !MasinvM(&h2, &h) )
    {
      condition = test_factor(factor, &h);
      goto ENDE;
    }

    MasM(&h1, &P->x);              /* Formeln siehe Funktion wepadd2P */
    MmuasM(&h1, &h1);
    MasMplM(&h, &h1, &h1);
    MplasM(&h1, &h);
    MplasM(&h1, &P->C->a);
    MmuasM(&h1, &h2);
    MasM(&h, &h1);
    MmuasM(&h1, &h1);
    MasMplM(&h2, &P->x, &P->x);
    MasMmiM(&h2, &h1, &h2);
    MasMmiM(&h1, &P->x, &h2);
    MmuasM(&h1, &h);
    MasMmiM(&Erg->y, &h1, &P->y);
    MasM(&Erg->x, &h2);
  }
  else                                                    /* P != Q  */
  {
    MasMmiM(&h, &Q->x, &P->x);
    if ( !MasinvM(&h2, &h) )
    {
      condition = test_factor(factor, &h);
      goto ENDE;
    }

    MasMmiM(&h1, &Q->y, &P->y);    /* Formeln siehe Funktion wepaddPQ */
    MasMmuM(&h, &h2, &h1);
    MasM(&h1, &h);
    MmuasM(&h1, &h1);
    MmiasM(&h1, &P->x);
    MasMmiM(&h2, &h1, &Q->x);
    MasMmiM(&h1, &P->x, &h2);
    MmuasM(&h1, &h);
    MasMmiM(&Erg->y, &h1, &P->y);
    MasM(&Erg->x, &h2);
  }
ENDE:
 dM(&h); dM(&h1); dM(&h2);

 return(condition);
}



/*
   wepmul - computes m*P for a point P on a curve E in the weierstrass model

   Def: ECM_STATUS wepmul(Erg, P, factor, m)
        unsigned int m;          // scalar to compute m * P
        ec_pointW P,             // point in weierstrass model
                  Erg;           // result: Erg = m * P
        register pInteger factor;  // possibly found factor
   Res: no_factor_found , if no factor of n was found
        factor_found    , if a factor was found
        hopeless        , curve useless
   Alg: fast exponentiation

   Andreas Mueller  , 01.07.1993
*/
```

```
#include <libelliptic.h>

ECM_STATUS wepmul ( ec_pointW Erg, ec_pointW P, register pInteger factor,\
                    unsigned int m)
{
  ECM_STATUS cond = no_factor_found;
  ec_pointW X, Z;

  wepalloc(X); wepalloc(Z);

  if (m <= 5)
  {
    wepsto(Z, X);
    for (; m > 1; m--)
      if ( (cond=wepadd(Z, P, Z, factor)) != no_factor_found ) goto ENDE;
  }
  else
  {
    wepsto(X, P);
    while ( !(m & 1) )
    {
      wepadd2P(X, X, factor);
      m >>= 1;
    }

    wepsto(Z, X);
    m--;

    while( m > 0 )
    {
      while ( !(m & 1) )
      {
        if ( (cond = wepadd2P(X, X, factor)) != no_factor_found )   goto ENDE;
        m >>= 1;
      }

      if ( (cond = wepaddPQ(Z, X, Z, factor)) != no_factor_found )  goto ENDE;
      else  m-- ;
    }
  }

 ENDE:
  wepsto(Erg, Z);
  wepfree(X); wepfree(Z);

  return (cond);
}


void Mneg(Minteger a)
{
  Minteger h; Integer h1;
  cMasM(&h, &a); cIasint(&h1, 0); MasI(&h, &h1);
  MasMmiM(&a, &h, &a);
  dM(&h); dI(&h1);
}


/*
   wepmul2 - computes m*P for a point P on a curve E in the weierstrass model

   Def: ECM_STATUS wepmul2(Erg, P, factor, m)
        unsigned int m;          // scalar to compute m * P
        ec_pointW P,             // point in weierstrass model
                  Erg;           // result: Erg = m * P
        register pInteger factor;  // possibly found factor
   Res: no_factor_found , if no factor of n was found
        factor_found    , if a factor was found
        hopeless        , curve useless
   Alg: Fast exponentiation using the Morain - Trick

   Andreas Mueller  , 01.07.1993
*/
```

```
ECM_STATUS wepmul2 ( ec_pointW Erg, ec_pointW P, register pInteger factor,\
                     unsigned int m)
{
  ECM_STATUS cond = no_factor_found;
  long b, ident;
  ec_pointW Q, H, T;
  wepalloc(Q); wepalloc(H);

  if (m <= 5)
  {
    wepsto(Q, P);
    for (; m > 1; m--)
      if ( (cond = wepadd(Q, P, Q, factor)) != no_factor_found ) goto ENDE;
    wepsto(Erg, Q);
  }
  else
  {
    if ( Erg == P )     {  wepalloc(T); ident = 1; }
    else                {  T  = Erg; ident = 0; }
    wepset0(T); wepsto(Q, P);

    if ( m&1 )
    {
      b = 1;
      wepsto(H, P); Mneg(H->y);
    }
    else      b = 0;
    m >>= 1;

    while ( m > 1 )
    {
      if ( m&1 )
      {
        if ( (cond = wepadd2P(Q, Q, factor)) != no_factor_found )  goto ENDE;
        if ( b == 0 )
        {
          wepsto(H, Q); Mneg(H->y);
          b = 1;
        }
        else  b = 2;
      }
      else
      {
        if (b == 1)
        {
          if((cond = wepadd(T, Q, T, factor)) != no_factor_found ) goto ENDE;
          if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )  goto ENDE;
          b = 0;
        }
        else if ( b == 2 )
        {
          if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )  goto ENDE;
          if((cond = wepadd(T, H, T, factor)) != no_factor_found ) goto ENDE;
          if((cond = wepadd(T, Q, T, factor)) != no_factor_found ) goto ENDE;
          b = 0;
        }
        else
          if((cond = wepadd2P(Q, Q, factor)) != no_factor_found ) goto ENDE;
      }
      m >>= 1;
    }

    if ( b == 0 )
    {
      if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )   goto ENDE;
      if((cond = wepadd(T, Q, T, factor)) != no_factor_found )  goto ENDE;
    }
    else if ( b == 1 )
    {
      if((cond = wepadd(T, Q, T, factor )) != no_factor_found ) goto ENDE;
      if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )   goto ENDE;
      if((cond = wepadd(T, Q, T, factor )) != no_factor_found ) goto ENDE;
    }
```

```
    else
    {
      if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )   goto ENDE;
      if((cond = wepadd2P(Q, Q, factor)) != no_factor_found )   goto ENDE;
      if((cond = wepadd(T, H, T, factor )) != no_factor_found ) goto ENDE;
      if((cond = wepadd(T, Q, T, factor )) != no_factor_found ) goto ENDE;
    }
    if ( ident )      wepsto(Erg, T);
  }

 ENDE:
  if ( ident ) wepfree(T);
  wepfree(Q); wepfree(H);

  return(cond);
}



/*
  wepcomp  -  elliptic point compare

  Def: BOOLEAN wepcomp(P, Q)
       ec_pointW P, Q;   // points on curve E in weierstrass model
  Res: TRUE   , if  P != Q
       FALSE  , if  P == Q

  Warning: This test works only for finite points.

  Andreas Mueller , 01.07.1993
*/

#include <libelliptic.h>

BOOLEAN wepcomp(ec_pointW P, ec_pointW Q)
{
  Minteger h;
  BOOLEAN erg = FALSE;

  cMasM(&h, &P->y);
  MmiasM(&h, &Q->y);
  if ( !Meq0(&h) )    erg = TRUE;

  MasMmiM(&h, &P->x, &Q->x);
  if ( !Meq0(&h) )    erg = TRUE;

  dM(&h);
  return(erg);
}



/*
   wepnegcomp - elliptic point negation compare

   Def: BOOLEAN wepnegcomp(P, Q)
        ec_pointW P, Q;    // points on curve E in montgomery model
   Res: TRUE   , if   P != -Q
        FALSE  , if   P == -Q

   Warning: This test works only for finite points.

   Andreas Mueller , 01.07.1993
*/

#include <libelliptic.h>

BOOLEAN wepnegcomp(ec_pointW P, ec_pointW Q)
{
  Minteger h;
  BOOLEAN erg = FALSE;

  cMasM(&h, &P->y);
  MplasM(&h, &Q->y);
```

```
  if ( !MeqO(&h) )    erg = TRUE;

  MasMmiM(&h, &P->x, &Q->x);
  if ( !MeqO(&h) )    erg = TRUE;

  dM(&h);
  return (erg);
}



/*
   ecm_stdcont2 - verbesserte Standard-Continuation fuer ECM nach Montgomery

   Def : ECM_STATUS ecm_stdcont2(P, factor, B, D)
         ec_pointW P;          // point in the weierstrass model
         unsigned long B,      // upper limit of primes for the first phase
                       D;      // upper limit of primes for the continuation
         register pInteger factor;   // possibly found factor
   Res : no_factor_found , if no factor of n was found
         factor_found    , if a factor was found
         hopeless        , no hope

   Warning: This function can't be used before sieving primes with the
         function "initprimes", but there is no check.

   Alg : Diplomarbeit Franz-Dieter Berger, Kap. 3.2

   Andreas Mueller , 25.08.1993
*/

#include <libelliptic.h>
#include <libint.h>

#define DIV  30
extern ModulusType n_mod;

ECM_STATUS ecm_stdcont2 ( ec_pointW P, register pInteger factor, \
                          unsigned long B, unsigned long D)
{
  ECM_STATUS cond = no_factor_found;
  int w, count, v, q, u;
  ec_pointW Q, Q2;
  Minteger c, lh, *x = 0;
  Integer h;

  cM(&lh, &n_mod);
  cIasint(&h, 1); cMasI(&c, &h, &n_mod); dI(&h);
  wepalloc(Q); wepalloc(Q2);
  wepsto(Q, P); Q2->C = P->C;

  w = (int) ceil( sqrt( (double)D ) );
  w = (w / DIV + 1) * DIV;
  D = w * w;  v = B / w;

  x = (Minteger *) calloc( w+2, sizeof(Minteger) );

  if ( (cond = wepadd2P(Q2, P, factor)) != no_factor_found)    goto ENDE;

                    /* Tafel berechnen */
  for (count=1; count<=w; count+=2)
  {
    if (gcd(w, count) == 1)      cMasM(&x[count], &Q->x);
    if ( (cond = wepaddPQ(Q, Q2, Q, factor)) != no_factor_found )   goto ENDE;
  }

  Mneg(P->y);
  if ( (cond = wepaddPQ(Q, Q, P, factor)) != no_factor_found )   goto ENDE;

                      /* es gilt:  Q = w*P  */
                      /* Startwert P = v*Q = v*w*P */
  if ( (cond = wepmul(P, Q, factor, v)) != no_factor_found )     goto ENDE;
```

```
  q = getprimes();
  count = 1;

  while( q < D && q > 1 )
  {
    if ( (u = v*w - q) < 1 )
    {
      v++;
      if ( (cond = wepadd(P, Q, P, factor)) != no_factor_found )  goto ENDE;
      u = v*w - q;
      if ( u < 0 )      u += w;
    }

    MasMmiM(&lh, &P->x, &x[u]);
    MmuasM(&c, &lh);

    if ( count++ >= 500 )
    {
      count = 1;
      if ( !MasinvM(&lh, &c) )
      {
        cond = test_factor(factor, &c);
        goto ENDE;
      }
    }
    q = getprimes();
  }

  if ( !MasinvM(&lh, &c) )      cond = test_factor(factor, &c);

 ENDE:
  for (count=0; count<=w; count++)
    if (gcd(w, count) == 1)
      dM(&x[count]);
  free(x);

 SCHLUSS:
  dM(&c); dM(&lh);
  wepfree(Q); wepfree(Q2);

  return(cond);
}
```

C.2.3 Die FFT-Continuation

```
/*
   simult_inv --- ggT-Trick f"ur ein Feld von Zahlen

   Def: void simult_inv (xinv, x)
        vector(ZZ_p) &xinv      // Ergebnis: xinv[i] = x[i] ^ -1
        const vector(ZZ_p) &x   // Vektor von Zahlen

   Andreas Mueller, 20.03.1995
*/

#include <ec_point.h>

void simult_inv ( vector(ZZ_p) &xinv, const vector(ZZ_p) &x)
{
  int i, len = x.length();
  ZZ_p tmp;

  vector(ZZ_p) z;
  z.SetMaxLength(len);

  ZZ_p *zz = z.elts();
  ZZ_p *xi = xinv.elts();
  const ZZ_p *xx = x.elts();

  zz[0] = xx[0];
  for ( i = 1 ; i < len ; i++ )    mul(zz[i], xx[i], zz[i-1]);
```

```
  inv( zz[len-1], zz[len-1] );

  for ( i = len-1 ; i > 0 ; i-- )
  {
    mul(tmp, zz[i] , zz[i-1]);
    mul(zz[i-1], xx[i] , zz[i]);
    xi[i] = tmp;
  }
  xi[0] = z[0];
}

/*
   Def: void my_vec_addPQ(erg, wp1, x)
        ec_point_W* erg       // Ergebnis: erg[i] = wp1[i] + wp1[i+1]
        const ec_point_W* wp1 // Vektor von Punkten
        vector(ZZ_p) &x       // Hilfsvektor f"ur ggT-Trick

  Andreas Mueller, 20.03.1995
*/

void my_vec_addPQ(ec_point_W* erg, const ec_point_W* wp1, vector(ZZ_p) &x)
{
  ZZ_p h, h1, h2;
  int i , len = x.length();
  ZZ_p* xp = x.elts();

  for(i = 0; i < len; i++)    sub(x[i], wp1[i+1].x, wp1[i].x);
  simult_inv(x, x);

  for(i = 0; i< len; i++)
  {
    sub(h1, wp1[i+1].y, wp1[i].y);
    mul(h, h1, xp[i]);
    h1 = h;
    sqr(h1, h1);
    sub(h1, h1, wp1[i].x);
    sub(h1, h1, wp1[i+1].x);

    sub(h2, wp1[i].x, h1);
    mul(h2, h2, h);
    sub(h2, h2, wp1[i].y);

    erg[i].assign(h1, h2);
  }
}


/*
   ecm_fftcont - FFT-Continuation mit Vielfachen der Form 2^(k/2) und 3^k

   Def: void ecm_fftcont(ec_point_W &P, int d1, int d2, int k)
        ec_point_W & P      // Ergebnis der 1. Phase
        int d1, d2,         // Groesse der Mengen
            k;

   Andreas Mueller, 20.03.1995
*/

void ecm_fftcont(ec_point_W &P, int d1, int d2, int k)
{
  ec_point_W l_P, DIFF(P);
  register int i, j, l, d12 = d1 >> 1;

  vector(ZZ_p) x(d1+1);

  for (j=0; j <= d12; j++)     // Berechnung der 1. Menge in Vektor x
  {
    for (i = 0; i < k; i++)
    {
      multiply_by_2(DIFF, P);
      addPQ(P, DIFF, P);
    }
    x[j] = P.X();
```

```
  }
  l_P.assign(P);

  for (j = d12+1; j <= d1; j++)
  {
    for (i = 0; i < k; i++)
    {
      multiply_by_2(DIFF, P);
      addPQ(P, DIFF, P);
    }
    x[j] = P.X();
  }

  k >>= 1;  d12 = d2/d1;

  ZZ_pX f, g, h;

  BuildFromRoots(f, x);   // f = (x - x[1]) * (x - x[2]) * ... * (x - x[d1])
#ifdef HUGE_MEMORY
  PolyModulus F;
  BuildPolyModulus(F, f);
#endif

  x.kill();

  vector(ZZ_p) y(d1);
                        // Berechnung der 2. Menge in Stuecken der Laenge d1
  for (j = 0; j < d1; j++)
  {
    for (i=0; i < k; i++)
      multiply_by_2(l_P, l_P);

    y[j] = l_P.X();
  }
  BuildFromRoots(g, y);      // 1. Teil berechnet und Polynom g aufgebaut

  for (i=0; i<d12; i++)
  {
    for (j=0; j<d1; j++)
    {
      for (l=0; l<k; l++)
        multiply_by_2(l_P, l_P);
      y[j] = l_P.X();
    }

    BuildFromRoots(h, y);   // naechster Teil des Polynoms in h berechnet

#ifdef HUGE_MEMORY
    MulMod(g, g, h, F);
#else
    FFTMul_single(g, g, h); FFTRem_single(g, g, f);
#endif
  }

  clear(h); y.kill();

  my_GCD(h, f, g);                // ggT- Berechnung
}


/*
   ecm_fftcont2 - FFT-Continuation mit Vielfachen aus
                  arithmetischen Progressionen

   Def: void ecm_fftcont2(ec_point_W &P, B, int d1, int d2)
        ec_point_W & P      // Ergebnis der 1. Phase
        unsigned long B;    // Primzahlschranke der 1. Phase
        int d1, d2;         // Groesse der Mengen

   Andreas Mueller, 20.03.1995
*/

void ecm_fftcont2(ec_point_W &P, int d1, int d2, int k)
{
  register int i = 0, j, e = 1;
```

```
  ec_point_W *ra;
  ra = new ec_point_W[k+1];

  for( i = 0; i <= k; i++)   ra[i].assign(P);

  for( i = 0; i <= k; i++)       /* Berechnung von ( (6 * i + 1)^k ) * P ) */
  {
    for( j = 1; j <= k; j++)
      multiply(ra[i], ra[i], e);
    e += 6;
  }

  for( i = 1; i <= k; i++)
  {
    for(j = k; j >= i; j--)
    {
      negate(ra[j-1].y, ra[j-1].y);
      add(ra[j], ra[j], ra[j-1]);
      negate(ra[j-1].y, ra[j-1].y);
    }
  }
                        /* Speicherplatz fuer x reservieren */
  vector(ZZ_p) x(d1+1);
  vector(ZZ_p) help(k);
  ZZ_p* xp = x.elts();

                        /* Berechnung des Vektors x */
  for ( i = 0; i < d1; i++)
  {
    x[i] = ra[0].x;
    my_vec_addPQ(ra, ra, help);
  }

  ZZ_pX f;
  BuildFromRoots(f, x);  // f = (x - x[1]) * (x - x[2]) * ... * (x - x[d1])

#ifdef HUGE_MEMORY
  PolyModulus F;
  BuildPolyModulus(F, f);
#endif

  x.kill();

  e = 6 * d1;           /* Initialisierung der 2. Folge */
  multiply(P, P, e);
  for(i = 0; i <= k; i++)   ra[i].assign(P);

  e = 1;
  for( i = 0; i <= k; i++)       /* Berechnung von ( 6d1(j + 1)^e ) * P */
  {
    for( j = 1; j <= k; j++)
      multiply(ra[i], ra[i], e);
    e++;
  }

  for( i = 1; i <= k; i++)
  {
    for(j = k; j >= i; j--)
    {
      negate(ra[j-1].y, ra[j-1].y);
      add(ra[j], ra[j], ra[j-1]);
      negate(ra[j-1].y, ra[j-1].y);
    }
  }

  int l, d12 = d2/d1 - 1;
  vector(ZZ_p) y(d1);
  ZZ_p* yp = y.elts();
                       // Berechnung der 2. Menge in Stuecken der Laenge d1
  for (j = 0; j < d1; j++)
```

```
  {
    yp[j] = ra[0].x;
    my_vec_addPQ(ra, ra, help);
  }
  BuildFromRoots(g, y);      // 1. Teil berechnet und Polynom g aufgebaut

  for (i=0; i<d12; i++)
  {
    for (j=0; j<d1; j++)
    {
      yp[j] = ra[0].x;
      my_vec_addPQ(ra, ra, help);
    }

    BuildFromRoots(h, y);   // naechster Teil des Polynoms in h berechnet

#ifdef HUGE_MEMORY                              // Zusammenbau des Polynoms g
    MulMod(g, g, h, F);
#else
    FFTMul_single(g, g, h); FFTRem_single(g, g, f);
#endif

  }

  y.kill(); h.kill();

  my_GCD(h, f, g);
}
```

C.3 Die Hauptprogramme

C.3.1 Eine C-Version mit verb. Std.Continuation

```
############################################################################
#
#     Makefile fuer Programme
#     10.03.1995
#
#     Andreas Mueller
#
############################################################################


include ../CONFIG

.c.o:
$(CC) -c $(CFLAGS) -I$(FACT_INC) $<

single: ecm

netz: ecm-master ecm-worker

all: ecm ecm-master ecm-worker

ecm: ecm.o
$(CC) ecm.o -o ecm $(LNFLAGS) $(CFLAGS) -I$(FACT_INC) \
-L$(FACT_LIB) -lfact $(LIBS_SEQ_C)

ecm-master:     ecm-master.o ecmi.o
$(CC) ecm-master.o ecmi.o -o ecm-master $(LNFLAGS) $(CFLAGS) \
-I$(FACT_INC) -L$(FACT_LIB) -lfact $(LIBS_SEQ_C) $(LIPS_PAR)

ecm-worker:     ecm-worker.o ecmw.o
$(CC) ecm-worker.o ecmw.o -o ecm-worker $(LNFLAGS) $(CFLAGS) \
-I$(FACT_INC) -L$(FACT_LIB) -llips2 -lfact $(LIBS_SEQ_C) $(LIPS_PAR)

clean:
-rm *.o

veryclean:
make clean
-rm ecm ecm-master ecm-worker
```

```
#include <stdio.h>
#include <libint.h>
#include <libelliptic.h>
#include <libmisc.h>
#include <ecm.h>

ModulusType n_mod;

#define PARA_NUM  9
char *param[PARA_NUM] = { "-h", "-t", "-r", "-v", "-i", "-num", \
                          "-time", "-p", "-c"};

void main(argc, argv)
int argc;
char *argv[3];
{
  register pInteger  n, nuebrig, factor, rest;
  unsigned long B, C, D, tfactor, prime_limit, trialdiv_limit = TRIAL;
  int timer1[2], timer2[2], strat[MAXSTRATEGY], restlength = REST,
      m = 0, i, k, j, Idim, job_grp = 0, jobs_avail, job_nr = 1;
      ecm_time=0, stdc_time=0, total_time, logfiling=0;
  long j_buf[MAXJOBS][5];
  BOOLEAN found = FALSE, prim = FALSE, info = TRUE, timing = FALSE;
  char n_string[MAXDEZSTEL], symbol[MAXFILENAMELEN], input[200];
  ECM_STATUS condition;
  eccurve E; ec_pointW WP; ec_pointM MP;
  FILE *fp;

  system("date");

  strat[0] = 0;
  unlink(STOP_ECM);

  if ( argc == 1 )
  {
    fprintf(stderr, "\n zu faktorisierende Zahl "); scanf("%s", n_string);
    k = read_strat(STRATEGYFILES, strat);
  }
  else
  {
    for (i=0; i < PARA_NUM; i++)
    {
      found = FALSE; j = 0;

      while ( j < argc-1 && !found )
      {
        j++ ;
        if ( !strcmp (param[i], argv[j]) )        found = 1;
      }

      if ( found )
        switch ( i )
        {
        case 0 :
          if ( fexist(ECM_HELP) )
          {
            sprintf(n_string,"more %s", ECM_HELP);
            system(n_string);
          }
          else fprintf(stderr, "kein Help-File vorhanden !! \n");
          exit(1);
        case 1 :
          trialdiv_limit = atoi(argv[j+1]); break;
        case 2 :
          if ( !(restlength = atoi(argv[j+1])) ) restlength = 1; break;
        case 3 :
          if ( !atoi(argv[j+1]) )  info = FALSE; break;
        case 4 :
          fprintf(stderr,"\nzu faktorisierende Zahl  "); scanf("%s", n_string);
          fprintf(stderr, "\nProbediv.       "); scanf("%lu", &trialdiv_limit);
          fprintf(stderr, "\nRestlaenge      "); scanf("%d", &restlength);
          fprintf(stderr, "\nInfo (1 ausfuehrlich)"); scanf("%d", &info);

          k = -1;
```

```
        fprintf(stderr, "\nStrategie (max. %d): #Stellen des Faktors\n", \
                        MAXSTRATEGY);
        fprintf(stderr, "\nAbschliessen mit 0\n");
        do
        {
          scanf("%d", &strat[++k]);
        }
        while( (k < MAXSTRATEGY-1 ) &&  ( strat[k] != 0 ) );
        break;
      case 5 :
        sprintf(n_string, "%s", argv[j+1]); break;
      case 6 :
        timing = TRUE; break;
      case 7 :
        sprintf(symbol, "%s", argv[j+1]); logfiling = 1; break;
      case 8 :
        sprintf(symbol, "%s", argv[j+1]);

        if ( (fp = fopena(symbol, "r")) == NULL )
        {
          fprintf(stderr, "\nAufsetzpunktefile %s fehlt!!\n", symbol);
          exit(1);
        }
        logfiling = 2; trialdiv_limit = 0;

        fgets(input, 190, fp); sscanf(input, "%d %d", &restlength, &info);
        fgets(n_string, MAXDEZSTEL, fp);
        fgets(input, 190, fp);
        sscanf(input, "%d %d %d", &job_nr, &job_grp, &jobs_avail);

        for (k=0; k <= jobs_avail; k++)
        {
          fgets(input, 190, fp);
          sscanf(input, "%d %lu %lu %lu %lu", &strat[k], &j_buf[k][0],\
                       &j_buf[k][1], &j_buf[k][2], &j_buf[k][3]);
        }
        if ( info )
        {
          fgets(input, 190, fp);
          sscanf(input, "%d %d %d", &ecm_time, &stdc_time, &total_time);
          printf("\n\nECM neu aufgesetzt: "); fflush(stdout);
        }
        fclose(fp); break;
      }
    }
  }

  if ( !strat[0] )      k = read_strat(STRATEGYFILES, strat);
  if ( !strlen(n_string) )
  { fprintf(stderr, "\n zu faktorisierende Zahl "); scanf("%s", n_string); }

  if ( !m )    m = random();

  if ( strlen(n_string) <= restlength )
  {
    printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
    printf("uebrig: (%d) %s\n", strlen(n_string), n_string);
    exit(0);
  }

  Idim = (int) (2.302585093*strlen(n_string)/32)+2;
  n = ncImaxlength(Idim); rest = ncImaxlength(Idim);
  factor = ncImaxlength(Idim); nuebrig = ncImaxlength(Idim);

  atoI(n_string, n);
  if ( Iisprime(n, 8) )
  {
    printf("\nPrimzahl : (%d) %s\n", strlen(n_string), n_string);
    exit(0);
  }

  if ( read_parameters(ECM_PARAMETERS_STD, strat[k-1], &B, &D, &i) !=
```

```
    (strat[k-1]) )
{
  fprintf(stderr, "Fehler beim Einlesen der Parameter zu MAXSTRATEGY!!!!\n");
  fprintf(stderr, "%d\n", strat[k-1]);
  fflush(stderr);
  exit(1);
}

prime_limit = D + 100;
if ( !trialdiv_limit )
{
  if ( timing )   start_time(NULL);
  if ( initprimes(SP_PRIMES, 1, prime_limit) )
  {
    fprintf(stderr, "\n Zu wenig Speicherplatz zum Primzahlsieben !!!\n");
    exit(1);
  }
  if ( timing )   stop_time("Primzahlen gesiebt");
}
else
{
  if ( prime_limit < trialdiv_limit ) prime_limit = trialdiv_limit + 100;
  if ( timing )   start_time(NULL);
  if ( init_trialdiv(prime_limit, trialdiv_limit, SP_PRIMES) )  exit(1);
  if ( timing )   stop_time("Primzahlen gesiebt");

  found = FALSE;
  if (info) printf("Probedivision bis %lu\n", trialdiv_limit);

  while ( !IEQ1(n) && (tfactor = trialdiv(n))>1 )
  {
    while ( !IasIdiD(nuebrig, n, tfactor) )
    {
      IasI(n, nuebrig);
      printf("Faktor: %lu\n", tfactor); fflush(stdout);
    }
    found = TRUE;
  }
  if ( timing )   stop_time("Probedivision");

  if ( found )
  {
    found = FALSE; Itoa(n, n_string);

    if ( Iisprime(n, 5) )
    {
      printf("\n Primzahl : (%d) %s\n", strlen(n_string), n_string);
      exit(0);
    }
    if ( strlen(n_string) <= restlength )
    {
      printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
      printf("uebrig: (%d) %s\n", strlen(n_string), n_string);
      exit(0);
    }
  }
}

if ( info )
{
  printf("\n\n                ECM mit verb. Standard-Continuation\n");
  printf("                 ==========================================\n");
  printf("\nzu faktorisieren bis auf %d Dezimalstellen:\n", restlength);
  printf("(%d) %s\n", strlen(n_string), n_string);
  printf("\nStrategie:\n");
}

if (logfiling < 2)
  jobs_avail = job_planing(ECM_PARAMETERS_STD, strat, j_buf, info);

cMod(&n_mod, n);
ecalloc(E); mepalloc(MP); wepalloc(WP);
```

```
resetprimes(1);

B = j_buf[job_grp][0];        /* first job */
C = j_buf[job_grp][1];
D = j_buf[job_grp][2];

while ( jobs_avail>0 && !IEQ1(n) )
{
  if ( info )
  {
    printf("\n"); fflush(stdout);
    system("date");
    printf("%d. Job in Arbeit ", job_nr); fflush(stdout);
  }

  if ( timing )   stop_time("ECM-Schleife (Verwaltung)");

  if ( mecgen16_fix(E, MP, factor, m) == no_factor_found )
  {
    if ( timing )   stop_time("Generierung von Kurve und Punkt");
    printf(" m = %d ", m); fflush(stdout);

    m++;
    while (j_buf[job_grp][3] == 0)
    {
      job_grp++;
      B = j_buf[job_grp][0];        /* next job */
      C = j_buf[job_grp][1];
      D = j_buf[job_grp][2];
    }

    start_timing(timer1);
    condition = mecfind_prac(E, MP, factor, B, C);
    if ( timing )   stop_time("Phase 1 ");
    get_timing(timer1);

    if ( info )
    {
      ecm_time += timer1[0]/100;
      printf(" -- Ecm-Time: %s ", strtime("hms", timer1[0]/100, 's'));
      fflush(stdout);
    }

    if ( condition == no_factor_found )
    {
      start_timing(timer2);

      if ( mwectrans(E, MP, WP, factor) == factor_found )
      {
        printf("\n Faktor beim Transformieren : ");
        fprintI(stdout, factor);
        printf("\n"); fflush(stdout); break;
      }
      if ( timing )   stop_time("Transformation");
      condition = ecm_stdcont2(WP, factor, B, D);
      if ( timing )   stop_time("Phase 2");
      get_timing(timer2);

      if ( info )
      {
        stdc_time += timer2[0]/100;
        printf("-- Cont-Time: %s\n", strtime("hms", timer2[0]/100, 's'));
        fflush(stdout);
      }
      if ( condition != factor_found )      IAS1(factor);
    }
  }

  if ( !(IEQ0(factor) || IEQ1(factor) || IeqI(n, factor)) )
  {
    Idiv(nuebrig, rest, n, factor);
    while( IEQ0(rest) )
```

```
  {
    IasI(n, nuebrig);
    if ( info )
    {
      Itoa(factor, n_string);
      printf("\nFaktor: (%d) %s\n", strlen(n_string), n_string);
    }
    else    fprintI(stdout, factor);

    Idiv(nuebrig, rest, n, factor);
  }

  Itoa(n, n_string);
  printf("uebrig: (%d) %s\n", strlen(n_string), n_string);

  if ( Iisprime(n, 5) )
  {
    printf("\nPrimzahl : (%d) %s\n", strlen(n_string), n_string);
    total_time = ecm_time + stdc_time;
    if ( info )
    {
      printf("\nPhase1-Ecm-Zeit : %s\n", strtime("dms", ecm_time, 's'));
      printf("Phase2-Cont-Zeit: %s\n", strtime("dms", stdc_time, 's'));
      printf("Gesamt-Zeit: %s\n", strtime("dms", total_time, 's'));
    }
    if (timing)   print_time(1);

    exit(0);
  }

  if ( strlen(n_string)<=restlength )
  {
    printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
    printf("uebrig: (%d) %s \n", strlen(n_string), n_string);
    if ( info )
    {
      total_time = ecm_time + stdc_time;
      printf("\nPhase1-Ecm-Zeit : %s\n", strtime("dms", ecm_time, 's'));
      printf("Phase2-Cont-Zeit: %s\n", strtime("dms", stdc_time, 's'));
      printf("Gesamt-Zeit: %s\n", strtime("dms", total_time, 's'));
    }
    if ( timing )   print_time(1);
    exit(0);
  }
  dMod(&n_mod); cMod(&n_mod, n);
}

j_buf[job_grp][3]--;
if ( !j_buf[job_grp][3] )   jobs_avail--;
++job_nr;

if ( fexist(STOP_ECM) )
{
  unlink(STOP_ECM);
  Itoa(n, n_string);

  printf("\nECM MUTWILLIG GESTOPPT!!!!!!!!!\n");
  printf("uebrig: (%d) %s\n", strlen(n_string), n_string);

  if ( info )
  {
    total_time = ecm_time + stdc_time;
    printf("\nPhase1-Ecm-Zeit : %s\n", strtime("dms", ecm_time, 's'));
    printf("Phase2-Cont-Zeit: %s\n", strtime("dms", stdc_time, 's'));
    printf("Gesamt-Zeit: %s\n", strtime("dms", total_time, 's'));
  }
  fflush(stdout);
  if ( timing ) print_time(1);
  if ( logfiling )
  {
    if ( (fp=fopena(symbol, "w")) == NULL )
      fprintf(stderr, "Kann Aufsetzpunktefile nicht schreiben\n");
```

```
        else
        {
          fprintf(fp, "%d %d\n", restlength, info);
          fprintf(fp, "%s\n", n_string);
          fprintf(fp, "%d %d %d\n", job_nr, job_grp, jobs_avail);
          for (i=0; i <= jobs_avail; i++)
            fprintf(fp, "%d %lu %lu %lu %lu\n", strat[i], j_buf[i][0], \
                        j_buf[i][1], j_buf[i][2], j_buf[i][3]);
          if (info) fprintf(fp, "%d %d %d\n", ecm_time, stdc_time, total_time);
        }
      }
      exit(0);
    }

    resetprimes(1);
    IAS1(factor);
  }

  if ( info )
  {
    printf("\nuebrig, %s : ", prim? "weil Primzahl": "aber keine Primzahl");
    printf("(%d) %s\n", strlen(n_string), n_string);
  }
  else
  {
    fprintI(stdout, n);
    if ( !prim ) printf("uebrig, aber keine Primzahl\n");
  }

  if ( info )
  {
    total_time = ecm_time + stdc_time;
    printf("\nPhase1-Ecm-Zeit : %s\n", strtime("dms", ecm_time, 's'));
    printf("Phase2-Cont-Zeit: %s\n", strtime("dms", stdc_time, 's'));
    printf("Gesamt-Zeit: %s\n", strtime("dms", total_time, 's'));
  }

  if ( timing )    print_time(1);
}
```

C.3.2 ECM mit FFT-Continuation für einen Rechner

```
############################################################################
#
#     Makefile fuer Programme
#     10.03.1995
#
#     Andreas Mueller
#
############################################################################

include ../CONFIG

single: ecmfft

netz: ecm-master-fft ecm-worker-fft

all: ecmfft ecm-master-fft ecm-worker-fft

.C.o:
        $(CPP) -c $(CPPFLAGS) -I$(FACT_INC_C++) $<

.c.o:
        $(CC) -c $(CFLAGS) -I$(FACT_INC) $<

ecm-master-fft: ecm-master-fft.o
        $(CC) ecm-master-fft.o -o ecm-master-fft $(LNFLAGS) $(CFLAGS)\
         -I$(FACT_INC) -L$(FACT_LIB) -lfact $(LIBS_SEQ_C) $(LIPS_PAR)

ecm-worker-fft: ecm-worker-fft.o ecmw-fft.o ec_point.o
        $(CPP) ecmw-fft.o ecm-worker-fft.o ec_point.o \
        -o ecm-worker-fft $(LNFLAGS) $(CPPFLAGS) -I$(FACT_INC_C++) \
```

```
        -L$(FACT_LIB) -lfact $(LIBS_SEQ_C++) $(LIPS_PAR)

ecmfft:         ec_point.o ecmfft.o
        $(CPP) ecmfft.o ec_point.o -o ecmfft $(LNFLAGS) $(CPPFLAGS)\
        -I$(FACT_INC_C++) -L$(FACT_LIB) -lfact $(LIBS_SEQ_C++)

clean:
        -rm *.o

veryclean:
        -rm *.o
        -rm ecmfft ecm-master-fft ecm-worker-fft
```

```
#include <GF.h>
#include <ec_point.h>

extern "C" {
#include <libelliptic.h>
#include <libint.h>
#include <libmisc.h>
  int unlink(char*);
  unsigned int strlen(char*);
  int strcmp(char*, char*);
}


ModulusType n_mod;

#define PARA_NUM  7
char *param[PARA_NUM] = {"-h", "-v", "-i", "-num", "-time", "-p", "-c"};

int timer1[2], timer2[2], ecm_time = 0, stdc_time = 0;
BOOLEAN info = TRUE, timing = FALSE;

void ItoZZ(const pInteger a, ZZ& b);
void ItoZZ_p(const pInteger a, ZZ_p & b);
void MtoZZ_p(Minteger *a, ZZ_p & b);

void divhand(ZZ_p &a)
{
  ZZ q = rep(a);
  ZZ g, r, n = ZZ_p::modulus();
  int count = 0;

  GCD(g, q, n);

  if ( g == n )
  {
    if (info)
      cout << "zu faktorisierende Zahl selbst als Faktor gefunden!!\n\n";
    cout << "Faktor: " << g << "\n\n";
    exit(0);
  }
  DivRem(q, r, n, g);

  while ( IsZero(r) )
  {
    count ++;
    n = q;
    DivRem(q, r, n, g);
  }

  if (count)
    if (info)
    {
      get_timing(timer2);
      stdc_time += timer2[0]/100;
      cout << "-- Cont-Time: " << strtime("hms", timer2[0]/100, 's');
      cout << "\n\nFaktor " << g;
      if ( count > 1 )  cout << " ^ " << count;
      cout << "\nRest: " << n << "\n\n"; cout.flush();

      cout << "\nPhase1-Ecm-Zeit : " << strtime("dms", ecm_time, 's');
```

```
      cout << "\nPhase2-Cont-Zeit: " << strtime("dms", stdc_time, 's');
      cout << "\nGesamt-Zeit:      " << strtime("dms", ecm_time +
                                                stdc_time, 's');
    }
    else
    {
      cout << "Faktor: " << g;
      if ( count > 1 )  cout << " ^ " << count;
      cout << "\nFaktor: " << n << "\n\n"; cout.flush();
    }

  if (timing) print_time(1);
  exit(1);
}


main(int argc, char *argv[3])
{
  ECM_STATUS condition = no_factor_found;
  eccurve C_E;
  ec_pointW WP;
  ec_pointM MP;
  char n_string[MAXDEZSTEL], symbol[MAXFILENAMELEN], input[200];
  register pInteger  n, factor;
  unsigned long B, d1, d2;
  long job_buf[MAXSTRATEGY][5];
  int l, jobs_avail, job_grp = 0, job_nr = 1, k, m, i, j,
      strategy[MAXSTRATEGY], logfiling = 0;

  FILE *fp;

  system("date");
  strategy[0] = 0;
  unlink(STOP_ECM);

  if ( argc == 1 )
  {
    fprintf(stderr, "\n zu faktorisierende Zahl ");
    scanf("%s", n_string);
    k = read_strategy(STRATEGYFILES, strategy);
  }
  else
  {
    for (i = 0; i < PARA_NUM; i++)
    {
      m = 0; j = 0;
      while ( j < argc-1 && !m )
      {
        j++;
        if ( !strcmp (param[i], argv[j]) )         m = 1 ;
      }

      if ( m )
        switch ( i )
        {
        case 0 :
          if ( fexist(ECM_HELP_FFT) )
          {
            sprintf(n_string,"more %s", ECM_HELP_FFT);
            system(n_string);
          }
          else
            fprintf(stderr, "kein Help-File vorhanden !! \n");
          exit(1);

        case 1 :
          if ( !atoi(argv[j+1]) )  info = FALSE; break;

        case 2 :
          fprintf(stderr, "\n zu faktorisierende Zahl  ");
          scanf("%s", n_string);
          fprintf(stderr, "\n m = "); scanf("%ld", &m);
          fprintf(stderr, "\n Info   (1 ausfuehrlich ) "); scanf("%d", &info);
```

```
        k = -1;
        fprintf(stderr, "\n Strategie (max. %d): #Stellen des Faktors\n",
                            MAXSTRATEGY);
        fprintf(stderr, "\n Abschliessen mit 0\n");

        do
        {
          k++;
          scanf("%d", &strategy[k]);
        }
        while( (k < MAXSTRATEGY-1) &&  (strategy[k] != 0) );
        break;

      case 3 :
        sprintf(n_string, "%s", argv[j+1]); break;

      case 4 :
        timing = TRUE; break;

      case 5 :
        sprintf(symbol, "%s", argv[j+1]); logfiling = 1; break;

      case 6 :
        sprintf(symbol, "%s", argv[j+1]);

        if ( (fp = fopena(symbol, "r")) == NULL )
        {
          fprintf(stderr, "Aufsetzpunktefile %s fehlt !! \n", symbol);
          exit(1);
        }
        logfiling = 2;

        fgets(input, 190, fp);
        sscanf(input, "%d ", &info);
        fgets(n_string, MAXDEZSTEL, fp);
        fgets(input, 190, fp);
        sscanf(input, "%d %d %d", &job_nr, &job_grp, &jobs_avail);

        for (k = 0; k < jobs_avail; k++)
        {
          fgets(input, 190, fp);
          sscanf(input, "%d %lu %lu %lu %lu", &strategy[k], &job_buf[k][0],
                        &job_buf[k][1], &job_buf[k][2], &job_buf[k][3]);
        }

        if ( info )
        {
          fgets(input, 190, fp);
          sscanf(input, "%d %d", &ecm_time, &stdc_time);
          cout << "\n\nECM neu aufgesetzt: "; cout.flush();
        }
        fclose(fp); break;
      }
  }
}

if ( !strategy[0] )      k = read_strategy(STRATEGYFILES, strategy);

if ( !strlen(n_string) )
{
  fprintf(stderr, "\n zu faktorisierende Zahl ");
  scanf("%s", n_string);
}

if ( !m )      m = random();
if ( timing ) start_time(NULL);

l = (int) (2.302585093*strlen(n_string)/32)+2;
n = ncImaxlength(l);
factor = ncImaxlength(l);

atoI(n_string, n);
cMod(&n_mod, n);
```

```
ZZ zz_n;
ItoZZ(n, zz_n);
GF_init(zz_n);              // initialize the modulus for ZZ_p

ZZ_p::DivHandler = (ZZ_p::PFC) divhand;

ecalloc(C_E);
mepalloc(MP);
wepalloc(WP);

if ( Iisprime(n, 8) )
{
  cout <<"\nPrimzahl : (" << strlen(n_string) << ") " << n_string << "\n\n";
  exit(0);
}

if ( read_parameters_fft(ECM_PARAMETERS_FFT, strategy[k-1], &B, &d1, &d2, &j)
     != strategy[k-1] )
{
  fprintf(stderr, "Fehler beim Einlesen der Parameter zu Maxstrat ");
  fprintf(stderr, "%d !!!\n", strategy[l-1]); fflush(stderr);
  exit(1);
}

if ( initprimes(SP_PRIMES, 1, B + 200) )
{
  fprintf(stderr, "\n Zu wenig Speicherplatz zum Primzahlsieben !!!\n");
  exit(1);
}

if ( timing )  stop_time("Primzahlen gesiebt");

if ( info )
{
  cout << "              ECM mit FFT \n";
  cout << "             ============\n\n\n";
  cout << "\n\nStrategy:\n-----------\n\n"; cout.flush();
}

if (logfiling < 2)
  jobs_avail = job_planing_fft(ECM_PARAMETERS_FFT, strategy, job_buf, info);

B = job_buf[job_grp][0];               // 1. Job
d1 = (int) job_buf[job_grp][1];
d2 = (int) job_buf[job_grp][2];
k = 12;

while ( jobs_avail > 0 )
{
  if ( info )
  {
    cout << "\n"; cout.flush();
    system("date");
    cout << job_nr++ << ". Job -- ";
  }

  if ( mecgen16_fix(C_E, MP, factor, m) == no_factor_found )
  {
    cout << " m = " << m++; cout.flush();

    if ( timing )  stop_time("Generierung von Kurve und Punkt");

    while (job_buf[job_grp][3] == 0)
    {
      job_grp++;
      B = job_buf[job_grp][0];        /* next job */
      d1 = job_buf[job_grp][1];
      d2 = job_buf[job_grp][2];
    }

    if (info) start_timing(timer1);
    condition = mecfind_prac(C_E, MP, factor, B, B);
```

```
    if ( timing )   stop_time("Phase 1 ");

    if (info )
    {
      get_timing(timer1); ecm_time += timer1[0]/100;
      cout << " -- Ecm-Time: " << strtime("hms", timer1[0]/100, 's');
      cout.flush();
    }

    if ( condition == no_factor_found )
    {
      if ( mwectrans(C_E, MP, WP, factor) == no_factor_found )
      {
        if (info) start_timing(timer2);

        ec_curve E;

        MtoZZ_p(&C_E->a, E.A());
        MtoZZ_p(&C_E->b, E.B());
        ec_point_W P(E);

        P.assign_one();
        MtoZZ_p(&WP->x, P.X());
        MtoZZ_p(&WP->y, P.Y());

        if ( timing )    stop_time("Transformation");

        ecm_fftcont2(P, d1, d2, k);
        if (timing)     stop_time("Phase 2");

        if (info)
        {
          get_timing(timer2); stdc_time += timer2[0]/100;
          cout << "-- Cont-Time: " << strtime("hms", timer2[0]/100, 's');
        }
      }
    }
    else
    {
      if (info)
      {
        cout << "Faktor in 1. Phase: "; cout.flush();
        start_timing(timer2);
        ZZ_p g;

        ItoZZ_p(factor, g); divhand(g);
      }
    }
  }

  job_buf[job_grp][3]--;
  if ( !job_buf[job_grp][3] )           jobs_avail--;

  if ( fexist(STOP_ECM) )
  {
    unlink(STOP_ECM);

    fprintf(stderr, "\nECM MUTWILLIG GESTOPPT!!!!!!!!!\n");
    fprintf(stderr, "uebrig: (%d) %s\n", strlen(n_string), n_string);

    if ( info )
    {
      cout << "\nPhase1-Ecm-Zeit : " << strtime("dms", ecm_time, 's');
      cout << "\nPhase2-Cont-Zeit: " << strtime("dms", stdc_time, 's');
      cout << "\nGesamt-Zeit:      " << strtime("dms", ecm_time +
                                                stdc_time, 's');
    }

    if ( timing ) print_time(1);

    if ( logfiling )
    {
      if ( (fp=fopena(symbol, "w")) == NULL )
        fprintf(stderr, "Kann Aufsetzpunktefile nicht schreiben\n");
      else
      {
```

```
          fprintf(fp, "%d\n", info);
          fprintf(fp, "%s\n", n_string);
          fprintf(fp, "%d %d %d\n", job_nr, job_grp, jobs_avail);

          for (i=0; i <= jobs_avail; i++)
          {
            fprintf(fp, "%d %lu %lu %lu %lu\n", strategy[i], job_buf[i][0],
                        job_buf[i][1], job_buf[i][2], job_buf[i][3]);
          }
          if ( info ) fprintf(fp, "%d %d\n", ecm_time, stdc_time);
        }
      }
      exit(1);
    }

    resetprimes(1);
  }

  if (info)
  {
    cout << "\n\nECM BEENDET\n";
    cout << "\nuebrig: (" << strlen(n_string) << ") " << n_string << "\n";

    cout << "\nPhase1-Ecm-Zeit : " << strtime("dms", ecm_time, 's');
    cout << "\nPhase2-Cont-Zeit: " << strtime("dms", stdc_time, 's');
    cout << "\nGesamt-Zeit:      " << strtime("dms", ecm_time +
                                               stdc_time, 's');
  }

  if (timing)  print_time(1);

  if (logfiling)  unlink(symbol);
}
```

C.3.3 ECM in einer Netzversion

```
#include <GF.h>
#include <ec_point.h>
#include <stdarg.h>
#define __malloc_h
#include <lips.h>

extern "C" {
#include <icontrib.h>
#include <libelliptic.h>
#include <libmisc.h>
#include <libint.h>

  int sleep (unsigned int);
  int out(char *, char *, char*, va_list);
  void  exitlips( void );
}

extern ModulusType n_mod;

extern int timers[2], zeit, curves;
extern float my_mips;
extern char symbol[MAXFILENAMELEN], my_logname[MAXFILENAMELEN];

void ItoZZ(const pInteger a, ZZ& b);
void ItoZZ_p(const pInteger a, ZZ_p & b);
void MtoZZ_p(Minteger *a, ZZ_p & b);
void ZZtostr(const ZZ & a, char s[100]);

void lips_divhand(ZZ_p &a)
{
  ZZ g, r, q = rep(a), zzn = ZZ_p::modulus();
  int wait = 300;
  char zahl[MAXDEZSTEL], help[1000];

  GCD(g, q, zzn);
```

```
  DivRem(q, r, zzn, g);

  if ( IsZero(r) )
  {
    get_timing(timers); zeit += timers[0];
    cout << "\nRechenzeit: " << strtime("hms", timers[0]/100, 's');
    cout.flush();

    cout << "\nFaktor: " << g << "\n\n";

    system("touch sicher");
    sprintf(zahl, "cat %s >> sicher", my_logname);
    system(zahl);

    ZZtostr(g, zahl);

    sprintf(help, "%s %s %d %d %f", zahl, symbol, zeit, curves, my_mips);

    while ( out(machine[0].name, "result", "s", help) )
    {
      sleep(wait);
      if ( wait < 2400 )  wait = (int) (wait * 1.5);
    }

    exitlips(); exit(1);
  }
}


void ecmw_fft(pInteger factor, int B, int d1, int d2, int k)
{
  eccurve C_E;
  ec_pointM MP;
  ec_pointW WP;

  ecalloc(C_E);
  mepalloc(MP);
  wepalloc(WP);

  if ( mecgen16(C_E, MP, factor) != no_factor_found )
  {
    ZZ_p a;
    ItoZZ_p(factor, a); lips_divhand(a);
  }

  cout << " 1. Phase "; cout.flush();

  if ( mecfind_prac(C_E, MP, factor, B, B) != no_factor_found )
  {
    ZZ_p a;
    ItoZZ_p(factor, a); lips_divhand(a);
  }

  if ( mwectrans(C_E, MP, WP, factor) == factor_found )
  {
    ZZ_p a;
    ItoZZ_p(factor, a); lips_divhand(a);
  }

  mepfree(MP);

  ec_curve E;
  ec_point_W P(E);

  MtoZZ_p(&C_E->a, E.A());
  MtoZZ_p(&C_E->b, E.B());
  P.assign_one();
  MtoZZ_p(&WP->x, P.X());
  MtoZZ_p(&WP->y, P.Y());

  ecfree(C_E); wepfree(WP);

  ecm_fftcont2(P, d1, d2, k);
}
```

```
#include <stdio.h>
#include <string.h>

#define __malloc_h
#include <lips.h>

#include <libint.h>
#include <icontrib.h>
#include <libelliptic.h>
#include <libmisc.h>

ModulusType n_mod;

char help[1000];

int start_machine(int i, int logfiling)
{
  char logfile[MAXFILENAMELEN], istring[MAXFILENAMELEN];

  if ( logfiling )        /* sollen die worker ein Logfile anlegen? */
    sprintf(logfile, "LOGFILE.%s", machine[i].name);
  else
    sprintf(logfile, "/dev/null");

  if ( rstart("ecm-worker-fft", machine[i].name, logfile) )
  {
    fprintf(stderr, "FEHLER bei rstart! ");
    return(-1);
  }
  else
  {
    sprintf(istring, "%d", i);
    sprintf(help, "%s %s", logfile, istring);

    if ( out(machine[i].name, "start", "s", help) )
    {
      fprintf(stderr, "FEHLER bei Symbol-out! ");
      return(-2);
    }
    else  return(0);
  }
}


void main(int argc, char *argv[3])
{
  long job_buf[MAXJOBS][5], k, totaltime = 0;

  register pInteger n, factor, nuebrig, rest;

  int factorfound = 0, Idim, i = 0, job_grp = 0, jobs_grp_avail,
      job_nr = 1, curves, zeit, rmachine_status[MAXMACHINENUMBER],
      timer[2], strategy[MAXSTRATEGY],
      statistik, prim = 0, logfiling;

  char n_string[MAXDEZSTEL], n_string2[MAXDEZSTEL],
       master_log[MAXFILENAMELEN], symbol[MAXFILENAMELEN],
       factor_file[MAXFILENAMELEN];
  float my_mips;
  FILE * fp;

  unlink (STOP_ECM);

  if ( argc==1 )
  {
    printf("%%%%\n"); fflush(stdout);
    system("date");

    printf("\nAufsetzpunkte (Laenge <=15)  "); scanf("%s", master_log);
    sprintf(factor_file, "factors_%s", master_log);

    scanf("%s", n_string);
    Idim = (int) (2.302585093*strlen(n_string)/32)+2;
```

```
    n = ncImaxlength(Idim);
    nuebrig = ncImaxlength(Idim);
    factor = ncImaxlength(Idim);
    rest = ncImaxlength(Idim);

    Idim = atoI(n_string, n);

    scanf("%ld", &k);
    scanf("%d", &logfiling);
    scanf("%d", &statistik);

    if ( prim = Iisprime(n, 8) )
    {
      printf("\n\nkein Netzstart, weil Zahl schon Primzahl");
      printf("\n Primzahl : %s (%d)\n", n_string, strlen(n_string));
      fflush(stdout);
      exit(0);
    }

    if ( (fp = fopen(factor_file, "w")) != NULL)
    {
      fprintf(fp, "%s\n", n_string);
      fclose(fp);
    }

               /* ---------- einlesen der Strategie ---------- */
    printf("Einlesen der Strategy\n"); fflush(stdout);
    read_strategy("ecm-strategy", strategy);

    if ( statistik )
    {
      printf("=========================================================\n\n");
      printf("zu faktorisieren durch ECM\n");
      printf("(%d) %s\n", Idim, n_string);
      printf("Strategie:\n");
    }

    jobs_grp_avail = job_planing_fft("ecm-parameters-fft", strategy,
                                      job_buf, statistik);
  }
  else
  {
    if ( (argc==3) && !strcmp(argv[1], "-c") )
    {
      sprintf(master_log, "%s", argv[2]);
      sprintf(factor_file, "factors_%s", argv[2]);

      if ( (fp = fopena(master_log, "r")) == NULL )
      {
        printf("\nKann Aufsetzpunktefile %s nicht oeffnen!!\n", master_log);
        exit(1);
      }
      else
      {
        fgets(help, 190, fp);
        sscanf(help, "%ld %d %d", &k, &logfiling, &statistik);
        fgets(n_string, MAXDEZSTEL, fp);

        Idim = (int) (2.302585093*strlen(n_string)/32)+2;

        n = ncImaxlength(Idim);
        nuebrig = ncImaxlength(Idim);
        factor = ncImaxlength(Idim);
        rest = ncImaxlength(Idim);
        Idim = atoI(n_string,n);

        fgets(help, 190, fp);
        sscanf(help, "%d %d", &job_nr, &jobs_grp_avail);

        for (i=job_grp; i<jobs_grp_avail; i++)
        {
          fgets(help, 190, fp);
```

```
        sscanf(help, "%d %lu %lu %lu %lu", &strategy[i], &job_buf[i][0],
                    &job_buf[i][1], &job_buf[i][2], &job_buf[i][3]);
      }

      fgets(help, 190, fp);
      sscanf(help,"%d", &totaltime);

      fclose(fp);
    }

    printf("\n\nECM neu aufgesetzt: "); fflush(stdout);
    system("date");
  }
}
if ( (fp = fopena(master_log, "w")) == NULL )
{
  printf("\nKann Aufsetzpunktefile %s nicht oeffnen!!\n", master_log);
}
else
{
  fprintf(fp, "%ld %d %d\n", k, logfiling, statistik);
  fclose(fp);
}

printf("initlips: %d\n", initlips("CLIENT_CONF", NULL, NULL));
printf("Anzahl Rechner:   %d\n", rmachine_number);

     /* -------------- starte remote-Maschinen -------------------- */

if ( rmachine_number==0 )
{
  printf("Keine Remote-Maschinen zum Rechnen da!!!!!\n"); fflush(stdout);
  exit(1);
}

printf("rstart von:\n");

for (i=0; i<rmachine_number; i++)
{
  printf("%20s: ", machine[i].name); fflush(stdout);
  rmachine_status[i] = start_machine(i, logfiling);
  printf("%d\n", rmachine_status[i]); fflush(stdout);
}

for (i = 0; i < rmachine_number; i++)
{
  if ( !rmachine_status[i] )
  {
    sprintf(symbol, "%d", i);
    printf("%3d. Job an %s  ", job_nr, machine[i].name);
    printf(" uebrig in Gruppe %lu : %lu jobs \n",job_grp,
             job_buf[job_grp][3]);
    sprintf(help, "%s %d %d %d %d", n_string, job_buf[job_grp][0],
                  job_buf[job_grp][1], job_buf[job_grp][2], k);

    if ( out(machine[i].name, symbol, "s", help) != 0 )
      l_error("Fehler bei OUT von START : ");
    else job_nr++;
  }
}

fflush(stdout);


do
{    /* -------- warte auf Ergebnisse und lese sie ein ---------- */
  do
  {
    if ( fexist(STOP_ECM) )
    {
      unlink (STOP_ECM);
      printf("\nECM MUTWILLIG GESTOPPT!!!!!!!!!\n"); fflush(stdout);
```

```
      goto ENDE;
    }

    i = inp("result", "S", help);
    if ( i )     sleep(10);
  }
  while ( i );

  sscanf(help, "%s %s %d %d %f", n_string2, symbol, &zeit, &curves, &my_mips);

  totaltime += (int)(zeit*my_mips/100);

  if ( job_buf[job_grp][3] > 0 )
  {
    job_buf[job_grp][3] -= curves;

    if ( job_buf[job_grp][3] <= 0)
    {
      job_buf[job_grp][3] = 0;
      jobs_grp_avail -- ;
    }
  }

  printf("\n"); fflush(stdout);
  system("date");
  printf("%s (%2.1f Mips): Job ist fertig! Kurven: %d Erg: %s\n",
          machine[atoi(symbol)].name, my_mips, curves, n_string2);

  printf("Real-Zeit: %s - ", strtime("hms", (int) (zeit/100), 's'));
  printf("Mips-Zeit: %s - ", strtime("hms", (int)(zeit*my_mips/100), 's'));
  printf("Gesamtzeit in Mips: %s\n", strtime("ydhms", totaltime, 's'));
  fflush(stdout);

  if ( IEQ0(factor) || IEQ1(factor) || IeqI(factor, n) )   factorfound = 0;
  else
  {
    Idiv(nuebrig, rest, n, factor);

    if ( (fp = fopen(factor_file, "a")) != NULL )
    {
      while( IEQ0(rest) )
      {
        IasI(n, nuebrig);
        fprintI(fp, factor); fprintf(fp,"\n");
        Itoa(factor, n_string2);
        printf("Faktor: (%d) %s\n", strlen(n_string2), n_string2);
        Itoa(nuebrig, n_string);
        printf("uebrig: (%d) %s\n", strlen(n_string2), n_string2);
        Idiv(nuebrig, rest, n, factor);
        factorfound = 1;
      }
      fclose(fp);
    }
    else
    {
      while( IEQ0(rest) )
      {
        IasI(n, nuebrig);
        fprintI(fp, factor); fprintf(fp,"\n");
        Itoa(factor, n_string);
        printf("Faktor: (%d) %s\n", strlen(n_string2), n_string2);
        Itoa(nuebrig, n_string);
        printf("uebrig: (%d) %s\n", strlen(n_string2), n_string2);
        Idiv(nuebrig, rest, n, factor);
        factorfound = 1;
      }
    }
  }

  printf(" uebrig in Gruppe %lu: %lu jobs \n",job_grp, job_buf[job_grp][3]);
  fflush(stdout);
  job_nr++;
```

```
  }
  while( !factorfound && jobs_grp_avail > 0 );

  printf("\n--------------------------------------------\n\n"); fflush(stdout);

  prim = Iisprime(n, 8);

  printf("\nECM BEENDET !!!!!!!!!\n"); fflush(stdout);

 ENDE:

  Itoa(n, n_string2);

  if ( (fp = fopena(master_log, "a")) == NULL )
    printf("\nKann Aufsetzpunktefile %s nicht oeffnen!!\n", master_log);
  else
  {
    fprintf(fp, "%s\n", n_string2);
    fprintf(fp, "%d %d\n", job_nr-rmachine_number+1, jobs_grp_avail);

    for (i=job_grp; i< jobs_grp_avail; i++)
      fprintf(fp, "%d %lu %lu %lu %lu\n", strategy[i], job_buf[i][0],
                     job_buf[i][1], job_buf[i][2], job_buf[i][3]);

    fprintf(fp, "%d\n", totaltime);
    fclose(fp);
  }

  if ( prim )
    if ( (fp = fopena(factor_file, "a")) != NULL )
    {
      fprintf(fp, "%s\n", n_string2);
      fclose(fp);
    }

  printf("\n=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n\n");
  printf("Gerechnete Jobs : %d\n\n", job_nr - rmachine_number);
  printf("    #Stellen           Jobs uebrig\n");
  for (i = job_grp; i< jobs_grp_avail; i++)
    printf("     %5d %20lu \n", strategy[i], job_buf[i][3]);

  printf("Gesamtzeit in Mips: %s\n", strtime("ydhms", totaltime, 's'));
  printf("\nuebrig, %s : ", prim? "weil Primzahl": "aber keine Primzahl");
  printf("(%d) %s\n", strlen(n_string2), n_string2);
  fflush(stdout);

  system("date");

  kill_all_clients();
  exitlips();

  printf("ENDE\n"); fflush(stdout);
}



#include <stdio.h>
#include <stdarg.h>
#include <GF.h>

#define __malloc_h
#include <lips.h>

extern "C" {
#include <libint.h>
#include <icontrib.h>
#include <libmisc.h>

  int initlips( char*, void*, void* );
  int in(char *, char *, va_list);
  int out(char *, char *,char*, va_list );
  void  exitlips( void );
  int sleep (unsigned int);
}
```

```
void ecmw_fft(pInteger, int, int, int, int);
void ItoZZ(const pInteger a, ZZ & b);
void ItoZZ_p(const pInteger a, ZZ_p & b);
void lips_divhand(ZZ_p & a);

                                /* globale Variablen */

int timers[2], zeit, curves;
char symbol[MAXFILENAMELEN], my_logname[MAXFILENAMELEN];
float my_mips;

ModulusType n_mod;

int send_time()
{
  cout << "\nKILL: Master killt mich!\n";
  get_timing(timers); zeit += timers[0];
  cout <<"Rechenzeit bis KILL: " << strtime("hms", timers[0]/100, 's') << "\n";
  cout.flush();

  system("date");

  exitlips();
  return 1;
}


int main()
{
  register pInteger l;

  int B, d1, d2, k, i = 0;
  char n_string[MAXDEZSTEL], help[1000];

  cout << "initlips FFT-Worker" << initlips(NULL, send_time, NULL);
  cout.flush();

  if ( in("start", "S", help) )
  {
    cout << "FEHLER bei Symbol einlesen! "; cout.flush();
    exitlips(); exit(1);
  }

  sscanf(help, "%s %s", my_logname, symbol);

  my_mips = mips();

  cout << "\nLogname : " << my_logname << "\nSymbolischer Name: " << symbol;
  cout << "\nMips im Worker: " << my_mips;

  while( in(symbol, "S", help) )
  {
    cout << "\nFEHLER bei 1. IN der Daten"; cout.flush();

    sleep(120);

    if ( i++ > 5 )
    {  exitlips(); exit(1); }
  }

  sscanf(help, "%s %d %d %d %d", n_string, &B, &d1, &d2, &k);

  system("date");

  cout << "\nErster Job angekommen:\nn: " << n_string;
  cout << "\nB=" << B << " d1=" << d1 << " d2=" << d2 << " k=" << k << "\n\n";
  cout.flush();

  if ( initprimes(200000, 1, B + 1000) == -1 )
  {
    cout << "Zu wenig Speicher zum Primzahlsieben\n"; cout.flush();
    exitlips(); exit(1);
  }
```

```
  l = ncI();
  atoI(n_string, l);
  cMod(&n_mod, l);

  ZZ zz_n;
  ItoZZ(l, zz_n);
  GF_init(zz_n);
  ZZ_p::DivHandler = (ZZ_p::PFC) lips_divhand;

  IAS1(l);
  curves = 1;

  while( TRUE )
  {
    resetprimes(1);
    cout << curves << ". Kurve "; cout.flush();

    start_timing(timers);
    ecmw_fft(l, B, d1, d2, k);
    get_timing(timers);

    zeit = timers[0];

    cout << "\nRechenzeit: "<< strtime("hms", timers[0]/100, 's');
    cout << "Job fertig \n\n"; cout.flush();
    system("date");

    sprintf(help, "%s %s %d %d %f", "1", symbol, zeit, curves, my_mips);

    if ( out(machine[0].name, "result", "s", help) != 0 )
    {
      curves++;
      cout << "\n Kein 'out': Kurven " << curves << "\n"; cout.flush();
    }
    else
      curves = 1;
  }
}
```

Literaturverzeichnis

[AtMo92] A.O.L. Atkin, F. Morain: *Finding Suitable Curves For The Elliptic Curve Method Of Factorization*, Preprint, Jan. 1992

[Be93] F.-D. Berger : *ECM — Faktorisieren mit elliptischen Kurven*, Diplomarbeit, Universität des Saarlandes, 1993

[BiBuPa95] I. Biehl, J. Buchmann, T. Papanikolaou: *LiDIA - A Library For Computational Number Theory*, Preprint, 1995

[BoLe93] W. Bosma, A.K. Lenstra: *An Implementation Of The Elliptic Curve Integer Factorization Method*, Preprint, Dez. 1993

[Br86] R.P. Brent: *Some Integer Factorization Algorithms using Elliptic Curves*, Australian Computer Science Communications 8, S. 149-163, 1986

[BrRi92] R.P. Brent, H.J.J. te Riele: *Factorizations of* $a^n \pm 1, 13 \leq a \leq 100$, CWI Report, Department of Numerical Mathematics NM-R9212, Juni 1992

[Bu88] P. Bundschuh: *Einführung in die Zahlentheorie*, Springer Verlag, 1988

[CoLeRi89] T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction To Algorithms*, MIT Press, 1989

[De93] T.F. Denny: *Faktorisieren mit dem quadratischen Sieb*, Diplomarbeit, Universität des Saarlandes, 1993

[De91] R. Dentzer: *libI — eine lange ganzzahlige Arithmetik*, Manuskript, Universität Heidelberg, 1991

[DiLe92] B. Dixon, A.K. Lenstra: *Massively parallel elliptic curve factoring*, Preprint, 1992

[Kn81] D.E. Knuth: *The Art of Computer Programming Vol.2*, Addison Wesley, 1981

[La80] E. Lamprecht: *Einführung in die Algebra*, UTB Birkhäuser, 1980

[LMMS94] F. Lehmann, M. Maurer, V. Müller, V. Shoup: *Counting the Number of Points on Elliptic Curves over Finite Fields of Characteristic Greater than Three,* Proceedings of ANTS I, Lecture Notes in Computer Science 877, Seite 60 - 70, 1994

[Le87] H.W. Lenstra, Jr.: *Factoring integers with elliptic curves*, Annals of Mathematics 126, S. 649 - 673, 1987

[Mo87] P.L. Montgomery: *Speeding the Pollard and Elliptic Curve Methods of Factorization*, Mathematics of Computation, Vol. 48 (177), S. 243 - 264, 1987

[Mo92/1] P.L. Montgomery: *Evaluating Recurrences Of Form* $X_{m+n} = f(X_m, X_n, X_{m-n})$ *Via Lucas Chains*, Preprint, Jan. 1992

[Mo92/2] P.L. Montgomery: *An FFT Extension of the Elliptic Curve Method of Factorization*, Dissertation, University of California, Los Angeles, 1992

[MoOl90] F. Morain, J. Olivos: *Speeding up the Computations on an Elliptic Curve using Addition-Subtraction Chains*, in Thèse, L'Université Claude Bernard-Lyon I, 1990

[Ri85] H. Riesel: *Prime Numbers and Computer Methods for Factorization*, Birkhäuser Verlag (Progress in mathematics 57), 1985

[RoSe92] R. Roth, T. Setz: *LIPS - A System For Distributed Processing On Workstations*, Manuskript, 1992

[Sh95] V. Shoup: *Practical computations with large polynomials over finite fields*, in Vorbereitung

[Si85] J. Silverman: *The Arithmetic of Elliptic Curves*, Springer-Verlag, 1985

[So95] T. Sosnowski: *Faktorisieren mit dem quadratischen Sieb*, Diplomarbeit, Universität des Saarlandes, 1995