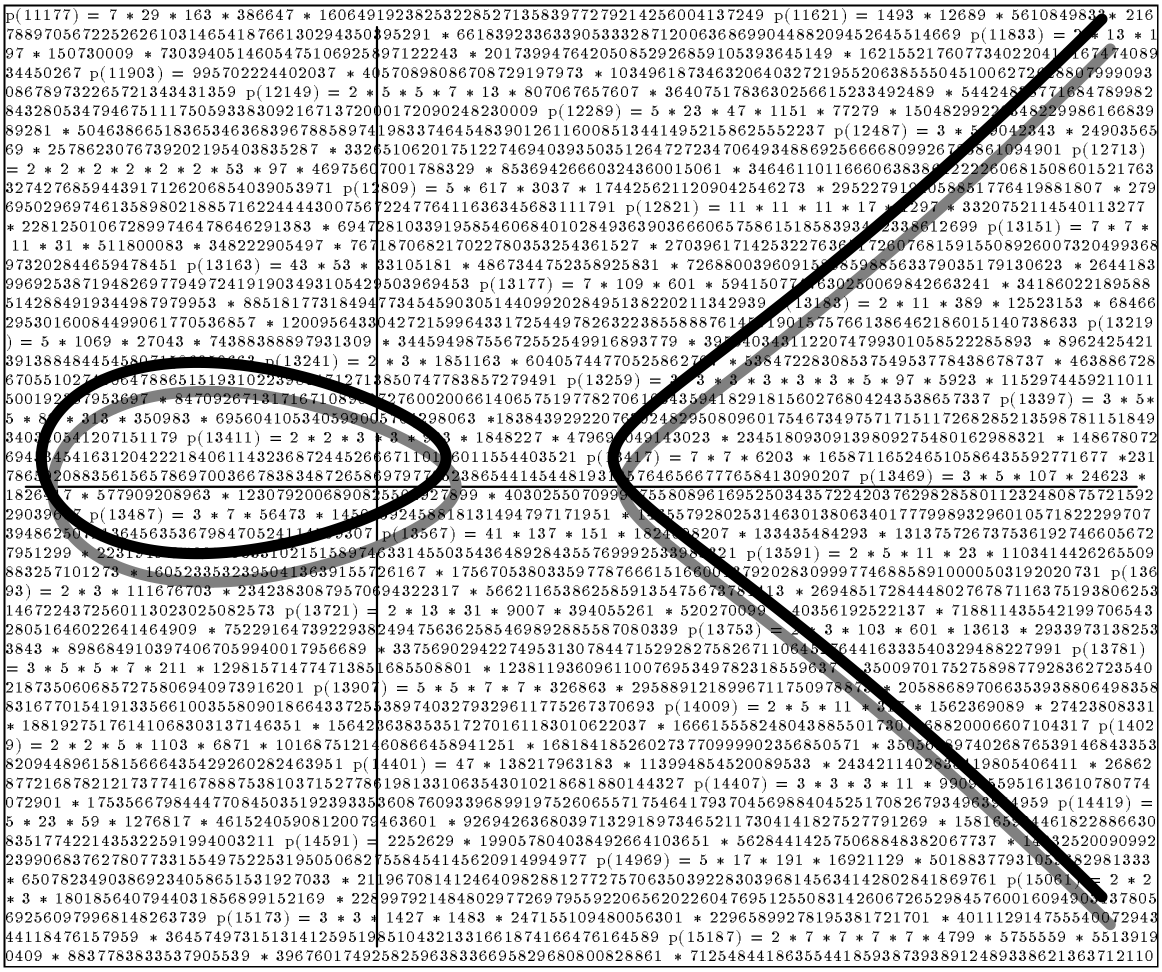
ECM
Faktorisieren mit elliptischen Kurven

Diplomarbeit

von

Franz-Dieter Berger



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

April 1993

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.

Des weiteren bedanke ich mich bei Christine Hollinger, Volker Müller und Jörg Zayer für das Korrekturlesen der Arbeit.

Außerdem danke ich Thomas Setz, der manch' lange Nacht damit verbrachte, LiPS meinen Wünschen entsprechend umzuprogrammieren.

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

Vielen Dank auch an Thomas Denny für gute Tips beim Programmieren, die hervorragende Zusammenarbeit beim Faktorisieren und lustige Nächte vor dem Terminal.

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

Abschließend danke ich besonders herzlich meiner Freundin Jutta Bartholomes für viele Verbesserungsvorschläge und tatkräftige Unterstützung bei hartnäckigen Problemen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Notation

$\mathbb{N}$	Menge der natürlichen Zahlen ohne 0
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{P}$	Menge der Primzahlen
$[x]_m$	$:= \{y \in \mathbb{Z} \mid y \equiv x \bmod m\}$ Restklasse modulo m
$\mathbb{Z}/m\mathbb{Z}$	Menge der Restklassen von $\mathbb{Z}$ modulo m
$(\mathbb{Z}/m\mathbb{Z})^*$	$:= (\mathbb{Z}/m\mathbb{Z}) - \{0\}$
$\mathrm{ord}(a)$	Ordnung des Elements a in seiner Gruppe (s. Def.(A.3))
$\mathrm{ord}_G(a)$	Ordnung des Elements a in der Gruppe G
$\lvert G\rvert$	Ordnung der Gruppe G (s. Def.(A.2))
$\log(b)$	Logarithmus von b zur Basis 2
$\log_a(b)$	Logarithmus von b zur Basis a
$\ln(b)$	Logarithmus von b zur Basis e, natürlicher Logarithmus
$\pi(x)$	Anzahl der Primzahlen $\leq x$ (s. Def. A.7)
$\varphi(x)$	Eulersche φ-Funktion (s. Def. A.9)
$O(f(x))$	groß O-Notation: größer als $Cf(x)$ für eine Konstante $C > 0$, $x \to \infty$
$o(f(x))$	klein o-Notation: kleiner als $\epsilon f(x)$, $\epsilon \to 0$, $x \to \infty$
$\mathrm{ggT}(a,b)$	größter gemeinsamer Teiler von a und b
$a \mid b$	a teilt b
$a \nmid b$	a teilt nicht b

Einleitung

Der Fundamentalsatz der Arithmetik besagt, daß jede von Eins verschiedene natürliche Zahl als Produkt endlich vieler Primzahlen darstellbar ist. Die Berechnung dieser Darstellung heißt Faktorisierung einer Zahl. Sie spielt in vielen Bereichen der Mathematik — besonders der Zahlentheorie — eine wichtige Rolle.
Leider läßt sich aus dem Beweis dieses Satzes kein effizientes Verfahren zur Faktorisierung herleiten. Deshalb beschäftigten sich schon Generationen von Mathematikern mit diesem Problem.
1987 entwickelte H. W. Lenstra, Jr. ein Verfahren, das gewisse Eigenschaften von elliptischen Kurven ausnutzt — ECM (Elliptic Curve Method). Dieses Verfahren erwies sich als sehr erfolgreich bei der Faktorisierung vieler großer Zahlen.
Generell kann man Faktorisierungsverfahren in zwei Klassen einteilen:

- allgemeine Faktorisierungsverfahren, deren Laufzeit im wesentlichen von der Größe der zu faktorisierenden Zahl abhängt,
- spezielle Faktorisierungsverfahren, deren Laufzeit im wesentlichen von besonderen Eigenschaften der Zahl abhängt.

Die Laufzeit von ECM hängt hauptsächlich von der Größe der Faktoren ab; somit ist ECM ein spezielles Faktorisierungsverfahren. In der Regel wird zuerst der kleinste Faktor einer Zahl gefunden, dann der nächstgrößere usw. Hat man den zweitgrößten Faktor einer Zahl gefunden, so kann der größte Faktor — durch Division der Zahl durch alle kleineren Faktoren — leicht berechnet werden. Eine Zahl kann also komplett faktorisiert werden, wenn der zweitgrößte Faktor eine bestimmte Größe (abhängig von der Rechenzeit, die man einsetzen will) nicht übersteigt.
Da die Größe der Faktoren i.a. vorher nicht bekannt ist, kann die Laufzeit von ECM für eine bestimmte Zahl nicht vorhergesagt werden. Man behilft sich, in dem man ECM auf eine Faktorgröße (z.B. 30 Dezimalstellen – abhängig von der einzusetzenden Rechenzeit) einstellt. Nach einem Programmdurchlauf kann man nun mit hoher Sicherheit sagen, daß alle Faktoren mit ≤ 30 Stellen gefunden wurden.
Das im Zuge dieser Arbeit entwickelte Programm erlaubt es 30-stellige Faktoren innerhalb 6 MIPS-Jahren zu finden (1 MIPS-Jahr entspricht der Rechenleistung eines 1 MIPS- (Million Instruction Per Second) Rechners über ein Jahr hinweg.). Kleinere Faktoren werden sehr viel schneller gefunden.

In diesem Beispiel könnte man somit 60 stellige Dezimalzahlen mit hoher Wahrscheinlichkeit faktorisieren.
Typischerweise hat der größte Primfaktor einer Zahl 63% ihrer Länge, der zweitgrößte Primfaktor 63% der Länge des Restes ($\approx$ 23% der Länge der Zahl) usw.
Wie schon erwähnt, hat man eine Zahl bereits faktorisiert, wenn man den zweitgrößten und alle kleineren Faktoren gefunden hat. Somit liegt die Wahrscheinlichkeit über 50%, mit ECM und optimalen Parametern für 30 stellige Faktoren Zahlen mit bis zu 130 Stellen zu faktorisieren. Diese Größenordnung ist bisher für allgemeine Faktorisierungsverfahren unerreichbar.
Der weltweit größte von ECM gefundene Faktor (März 1993) hatte 43 Dezimalstellen und wurde innerhalb von 19 Stunden von der verteilten Version meines Programmes berechnet (siehe auch Kapitel 5).
Sehr erfolgreich erwies sich die Kombination von ECM und dem "quadratischen Sieb" (QS) (s. [Denny]) — einem allgemeinen Faktorisierungsverfahren. Mit ECM werden die kleinen Faktoren einer Zahl schnell gefunden und dadurch der nicht faktorisierte Rest der Zahl in Reichweite des QS gebracht.

Meine Arbeit gliedert sich wie folgt:

Kapitel 1: Das Prinzip von ECM stammt von Pollards p-1 – Methode zur Faktorisierung. Da sie nur auf elementare Mittel der Zahlentheorie zurück greift, ist sie leicht verständlich. Deshalb erkläre ich sie zuerst und leite daraus ECM ab.

Kapitel 2: Der prinzipielle ECM-Algorithmus aus Kapitel 1 ist nicht praktikabel, da die benötigten Rechenoperationen auf elliptischen Kurven über Ringen zu komplex sind. Deshalb wird der Algorithmus so modifiziert, daß er die schnellen Rechenoperationen auf elliptischen Kurven über Körpern benutzen kann. Außerdem beschreibe ich zwei wichtige Beschleunigungen des Algorithmus und gehe auf die Berechnung der optimalen Parameter ein.

Kapitel 3: ECM benötigt i.a. viele Versuche mit verschiedenen elliptischen Kurven um einen Faktor zu finden. War ein Versuch erfolglos, so wird eine neue Kurve zufällig gewählt und der Algorithmus erneut gestartet. Leider wird das Ergebnis (bzw. die Arbeit) des vorherigen Versuchs nicht genutzt. Die in diesem Kapitel beschriebenen Algorithmen setzen auf dem Ergebnis eines erfolglosen Versuchs auf und versuchen mit schnellen Verfahren doch noch einen Faktor zu finden. Das erhöht die Erfolgswahrscheinlichkeit eines einzelnen Versuchs erheblich und beschleunigt ECM enorm.

Kapitel 4: Hier beginnt der praktische Teil meiner Arbeit.
Die wichtigsten Datenstrukturen und Funktionen, die Parallelisierung von ECM und die Installation und Bedienung der Programme werden beschrieben.

Kapitel 5: Der Unterschied zwischen Theorie und Praxis wird dokumentiert – Laufzeittests.

Kapitel 6: Um die Forschung auf dem Gebiet der Faktorisierung zu fördern und den Fortschritt zu dokumentieren, führt das Unternehmen "RSA Data Security" einen Faktorisierungswettbewerb durch, an dem wir (Thomas Denny und ich) erfolgreich teilnahmen.

Kapitel 7: Hier gebe ich noch einige Verweise auf andere wichtige Verbesserungen von ECM an, die den Rahmen dieser Arbeit gesprengt hätten.

Kapitel 1

Das Prinzip

1.1 Der Pollardsche p-1 – Algorithmus

Die elliptische Kurven Methode basiert auf der p-1 – Methode von Pollard. Hier läßt sich die zugrunde liegende Idee besonders einfach erkennen.

Betrachten wir dazu folgendes Szenario:

Seien $k,\ N \in \mathbb{N}$, N wollen wir faktorisieren.

Sei $p \in \mathbb{P}$ ein Primfaktor von N.

Sei $a \in \mathbb{Z}$ teilerfremd zu N, d.h. $\mathrm{ggT}(a, N) = 1$.

Setze $e := k(p-1)$.

Mit Hilfe des kleinen Fermatschen Satzes (Satz A.6) folgt:

$$\begin{aligned} a^{p-1} \equiv 1 \bmod p \quad \Rightarrow \quad a^{k(p-1)} &\equiv 1 \bmod p \\ \Rightarrow \quad a^e &\equiv 1 \bmod p \\ \Rightarrow \quad a^e - 1 &\equiv 0 \bmod p \\ \Rightarrow \quad p \mid a^e - 1. & \end{aligned} \tag{1.1}$$

Andererseits gilt: $p \mid N$, da p ein Primfaktor von N ist.
Es folgt:

$$\mathrm{ggT}(a^e - 1, N) > 1.$$

Ist die Ordnung von $a \bmod N$ kein Teiler von e, so gilt außerdem:

$$\begin{aligned} & a^e \not\equiv 1 \bmod N \\ & \Rightarrow 1 < \mathrm{ggT}(a^e - 1, N) < N, \end{aligned} \tag{1.2}$$

d.h.: Die ggT-Berechnung liefert einen echten Teiler von N.

Daraus ergibt sich folgender informaler

Algorithmus 1.1 :

<u>Eingabe:</u> N

<u>gesucht:</u> p

1. **Wähle** $a > 1$.
   Ist der $\mathrm{ggT}(a, N) > 1$, dann haben wir einen Teiler von N gefunden, sonst muß man jetzt

   $$e = k(p-1)$$

   berechnen. Das Problem dabei ist: Wir kennen p nicht, sondern suchen es. Nun kann aber jede positive ganze Zahl — insbesondere auch e — auf eindeutige Weise (bis auf Reihenfolge) als Produkt von Primzahlpotenzen dargestellt werden. Deshalb

2. **Wähle** $B_1, B_2 \in \mathbb{N}$ und berechne

   $$e = \prod_{\mathbb{P} \ni p \leq B_1} p^{e_p}, \quad e_p = \max\{l \in \mathbb{N} : p^l \leq B_2\}.$$

   Die Schranken B_1 und B_2 sollen die Rechenzeit und den Speicherplatz begrenzen. Sie müssen jedoch so groß sein, daß e von $p-1$ geteilt wird, genauer gesagt, muß B_1 größer gleich dem größten Primfaktor von $p-1$ sein und B_2 größer gleich der größten Primzahlpotenz in $p-1$. Hier hofft man, daß $p-1$ in kleine Primfaktoren zerfällt. Ist dies nicht der Fall, so kann man N mit dieser Methode nicht faktorisieren.
   Schließlich:

3. **Berechne**

   $$d = \mathrm{ggT}(a^e - 1, N).$$

   Ist $1 < d < N$, dann haben wir einen Teiler von N gefunden,
   sonst GOTO Schritt 1 oder gib auf.

Ist in Schritt 3) $d = 1$, so kann man einen neuen Versuch mit größeren Schranken B_1 und B_2 wagen oder ein anderes a wählen und damit auf einen Zufallstreffer hoffen, der allerdings sehr unwahrscheinlich ist.

Laufzeit:
In Schritt 2) benötigt man $O(\log e)$ Multiplikationen modulo N, wenn man den binären Algorithmus (oft auch "repeated squaring and multiplying" genannt) (s. [Horster]) benutzt. Schritt (1) und Schritt (3) fallen dem gegenüber nicht ins Gewicht.

Beispiel 1.2 :
zu faktorisieren: $N = 551$.

1. wähle $a = 10$
2. wähle $B_1 = 3,\ B_2 = 9$

   $\Rightarrow e = 2^3 \cdot 3^2 = 72$
3. berechne $a^e \bmod N$:

   $a^e = 10^{72} \equiv (10^9)^8 \equiv 18^8 \equiv (18^4)^2 \equiv 286^2 \equiv 248 \bmod 551$

   $\Rightarrow a^e - 1 \equiv 247 \bmod N$

   $\mathrm{ggT}(247, 551) = 19$

   $\Rightarrow 19|551,\ 551 = 19 \cdot 29$

Betrachtet man mit dem jetzigen Wissen ($551 = 19 \cdot 29,\ p = 19,\ \Rightarrow p - 1 = 18 = 2 \cdot 3 \cdot 3$) noch einmal Schritt 2), so sieht man, daß sich e in der gewünschten Form $e = k(p-1)$ darstellen läßt:

$$e = 2^3 \cdot 3^2 = 2^2 \cdot (p-1) = k(p-1),$$

da $p - 1$ in kleine Primfaktoren zerfällt. Hätten wir in Schritt 2) $B_2 = 8$ gewählt, so wäre $e = 2^3 \cdot 3 = 24 \neq k(p-1)$ und wir hätten $p = 19$ nicht gefunden.

Beispiel 1.3 :
zu faktorisieren: $N = 529$

1. wähle $a = 2$
2. $B_1 = 3,\ B_2 = 9$ wie in Beispiel 1.2

   $\Rightarrow e = 72$

3. $a^e = 2^{72} \equiv (2^{18})^4 \equiv 289^4 \equiv 18 \bmod 529$
   $\mathrm{ggT}(17, 529) = 1$

Hier zerfällt $p-1$ nicht in kleine Primfaktoren, denn

$$529 = 23 \cdot 23 \Rightarrow p = 23 \Rightarrow p - 1 = 22 = 2 \cdot 11$$

$$\Rightarrow e \neq k(p-1).$$

Um 529 zu faktorisieren, hätte man $B_1,\ B_2 \geq 11$ wählen müssen. Mit $B_1 = B_2 = 11$ ist aber

$$e = 2^3 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 = 27720.$$

Hier sieht man schon, daß sehr schnell die Grenzen eines vertretbaren Rechenaufwandes erreicht werden, wenn $p-1$ nicht in kleine Faktoren zerfällt. Das bedeutet aber, daß sich die p-1 – Methode nur für spezielle Zahlen eignet.
Ein sehr viel universelleres Verfahren ist die elliptische Kurven Methode — das Thema dieser Arbeit.

In der Praxis ist es nicht nötig, e explizit auszurechnen, denn wenn $e = e_1 \cdot e_2 \cdot e_3 \cdots$, dann ist $a^e \equiv a^{e_1 \cdot e_2 \cdot e_3 \cdots} \equiv (...(((a^{e_1} \bmod N)^{e_2} \bmod N)^{e_3} \bmod N)...) \bmod N$.

1.2 Das Prinzip

Betrachten wir nun die p-1 – Methode etwas allgemeiner. Sei $G_p := (\mathbb{Z}/p\mathbb{Z})^*$. Wir rechnen in $G_N := (\mathbb{Z}/N\mathbb{Z})^*$. Definieren wir nun den folgenden Homomorphismus (Reduktion modulo p):

$$\begin{array}{rcl} \phi : G_N & \to & G_p \\ {[x]_N} & \mapsto & [x]_p \end{array}$$

Theoretisch finden wir einen Teiler von N, wenn a^e ungleich dem neutralen Element in G_N ist ($a^e \not\equiv 1 \bmod N$, s. Gleichung (1.2)) und $\phi(a)^e$ gleich dem neutralen Element in G_p ist ($a^e \equiv 1 \bmod p$, s. Gleichung (1.1)). Es müssen also folgende Bedingungen gelten:

1. $e \neq c_1 \cdot \mathrm{ord}_{G_N}(a),\ \forall c_1 \in \mathbb{N}$, denn sonst wäre $a^e \equiv 1 \bmod N$,
2. $e = c_2 \cdot \mathrm{ord}_{G_p}(\phi(a)),\ c_2 \in \mathbb{N}$.

Mit der Berechnung des ggT$(a^e - 1, N)$ in Schritt (3) des Algorithmus 1.1 testen wir, ob diese Bedingungen erfüllt sind.

Praktisch finden wir nur dann einen Teiler von N, wenn zusätzlich $\mathrm{ord}_{G_p}(\phi(a))$ in kleine Primfaktoren zerfällt, da e aus kleinen Primzahlen zusammengesetzt ist (s. Schritt 3) des Algorithmus 1.1). Das ist unter anderem der Fall, wenn $|G_p| = p-1$ in kleine Primfaktoren zerfällt. Die Glücksfälle, in denen $\mathrm{ord}_{G_p}(\phi(a))$ in kleine Faktoren zerfällt nicht aber $|G_p|$, scheinen sehr rar zu sein und werden deshalb auch nicht weiter beachtet. Ist die erste Bedingung einmal nicht erfüllt, so muß man den Algorithmus mit einem anderen a wiederholen oder e verkleinern.

1.3 Lenstras elliptische Kurven Methode

Lenstras Idee war es, die beiden Gruppen G_N und G_p durch Gruppen von Punkten auf elliptischen Kurven E modulo N ($E(\mathbb{Z}/N\mathbb{Z})$) bzw. modulo p ($E(\mathbb{Z}/p\mathbb{Z})$) zu ersetzen, was den Namen "Elliptic Curve Method: ECM" erklärt.
Die Arithmetik mit Punkten auf elliptischen Kurven erkläre ich im nächsten Kapitel. Im Moment genügt es zu wissen, daß ein Punkt aus einer x-, y- und z-Koordinate besteht (also $P = (x : y : z)$), daß die Menge der Punkte auf einer Kurve eine Gruppe bildet und daß man die Verknüpfung auf der Gruppe gewöhnlich als Addition bezeichnet. Man kann also zwei Punkte auf einer Kurve addieren und erhält als Ergebnis wieder einen Punkt auf der Kurve.
Der Homomorphismus sieht dann folgendermaßen aus:

$$\phi : E(\mathbb{Z}/N\mathbb{Z}) \to E(\mathbb{Z}/p\mathbb{Z})$$

(Reduktion der Koordinaten von P modulo p).

Hier finden wir theoretisch einen Teiler von N, wenn $k \cdot P$ $(:= \underbrace{P + P + \cdots + P}_{k\text{-mal}})$ (nicht P^k, da die Gruppenoperation eine Addition und keine Multiplikation ist) ungleich dem neutralen Element in $E(\mathbb{Z}/N\mathbb{Z})$ ist und $k \cdot \phi(P)$ gleich dem neutralen Element in $E(\mathbb{Z}/p\mathbb{Z})$ ist. Somit müssen wieder die beiden Bedingungen

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

gelten. Ist E durch die homogene Weierstrass-Gleichung

$$y^2 z = x^3 + axz^2 + bz^3$$

mit neutralem Element $\mathcal{O} = (0 : 1 : 0)$ gegeben, so ist das genau dann der Fall, wenn p die z-Koordinate von $k \cdot P$ teilt, N aber nicht. Das läßt sich so einsehen: Sei $\mathcal{O}_N$ das neutrale Element in $E(\mathbb{Z}/N\mathbb{Z})$, $\mathcal{O}_p$ neutrales Element in $E(\mathbb{Z}/p\mathbb{Z})$ und z_{kP} die z-Koordinate von $k \cdot P$.

$$\begin{aligned}
k \neq c_1 \cdot \mathrm{ord}_{E(\mathbb{Z}/N\mathbb{Z})}(P) \quad &\Leftrightarrow \quad kP \neq \mathcal{O}_N \\
&\Leftrightarrow \quad z_{kP} \not\equiv 0 \bmod N \\
& \quad \text{(es gibt nur einen Punkt mit z=0 nämlich } \mathcal{O}) \\
&\Leftrightarrow \quad N \nmid z_{kP} \\
k = c_2 \cdot \mathrm{ord}_{E(\mathbb{Z}/p\mathbb{Z})}(\phi(P)) \quad &\Leftrightarrow \quad k\phi(P) = \mathcal{O}_p \\
&\Leftrightarrow \quad z_{kP} \equiv 0 \bmod p \\
&\Leftrightarrow \quad p \mid z_{kP}
\end{aligned}$$

Im Prinzip funktioniert jetzt ECM wie folgt:

Algorithmus 1.4 :

1. **Wähle** zufällig eine elliptische Kurve E und einen Punkt $P \neq \mathcal{O}$ darauf.
   Die Wahl des Punktes P entspricht der Wahl von a im p-1 – Algorithmus. Da es viele elliptische Kurven gibt, müssen wir uns hier zusätzlich auf eine Kurve festlegen.

2. **Wähle** $B_1, B_2 \in \mathbb{N}$ und berechne

   $$k = \prod_{\mathbb{P} \ni p \leq B_1} p^{k_p}, \quad k_p = \max\{l \in \mathbb{N} : p^l \leq B_2\}.$$

3. **Berechne** $d = \mathrm{ggT}(\text{z-Koordinate von } k \cdot P, N)$

Wenn man nur eine einzige Kurve benutzt, hat ECM genau die gleichen Eigenschaften wie die p-1 – Methode, d.h.: In der Praxis findet man einen Teiler, wenn $|E(\mathbb{Z}/p\mathbb{Z})|$ in kleine Primfaktoren zerfällt. Ein Satz von Hasse (s. [Koblitz]) besagt, daß

$$p + 1 - 2\sqrt{p} \quad \leq \quad |E(\mathbb{Z}/p\mathbb{Z})| \quad \leq \quad p + 1 + 2\sqrt{p} \tag{1.3}$$

ist, wobei $|E(\mathbb{Z}/p\mathbb{Z})|$ von der Kurve E und p abhängt. Somit kann man nach einem Fehlversuch einfach eine neue Kurve wählen, erhält dadurch eine andere Gruppenordnung und hat damit eine neue Chance N zu faktorisieren. Diese Option ist in der p-1 – Methode nicht verfügbar.
Leider ist dieser Algorithmus nicht effizient, weil die Berechnung von $k \cdot P$ sehr aufwendig ist. Im nächsten Kapitel beschreibe ich den praktikablen Algorithmus, doch zuvor gebe ich ein kurze Einführung über das Rechnen mit Punkten auf elliptischen Kurven.

Kapitel 2

ECM

2.1 Grundlagen

In diesem Kapitel werden einige Fakten über elliptische Kurven, die zum weiteren Verständnis benötigt werden, genannt.

Sei $\mathbb{K}$ ein Körper, wobei unser besonderes Interesse dem Fall $\mathbb{K} = \mathbb{F}_p$ für eine Primzahl p gilt. Der Einfachheit halber setzen wir voraus, daß die Charakteristik von $\mathbb{K}$ ungleich 2 und ungleich 3 ist, denn in diesen Fällen werden die elliptischen Kurven anders definiert (s. [Koblitz]).

Definition 2.1 (elliptische Kurve)
Eine elliptische Kurve $E_{a,b}$ *oder einfach* E über $\mathbb{K}$ *ist ein Paar* $(a,b) \in \mathbb{K} \times \mathbb{K}$*, für das* $4a^3 + 27b^2 \neq 0$ *ist;* a *und* b *sind die Koeffizienten der Weierstrass-Gleichung*

$$y^2 = x^3 + ax + b. \tag{2.1}$$

Die Menge aller Punkte *auf einer solchen elliptischen Kurve über* $\mathbb{K}$ *ist*

$$E(\mathbb{K}) := \{(x:y:z) \;\in\; \mathbb{P}^2(\mathbb{K}) \mid y^2z = x^3 + axz^2 + bz^3\}.$$

$\mathbb{P}^2(\mathbb{K})$ *bezeichnet die projektive Ebene über* $\mathbb{K}$*. Sie besteht aus Äquivalenzklassen der Tripel* $(x,y,z) \in \mathbb{K}^3$*,* $(x,y,z) \neq (0,0,0)$*, wobei* (x,y,z) *und* (x',y',z') *äquivalent sind, wenn ein* $c \in \mathbb{K}^*$ *existiert, sodaß gilt:* $cx = x'$*,* $cy = y'$*,* $cz = z'$*.*
$(x:y:z)$ *bezeichnet die Äquivalenzklasse, die* (x,y,z) *enthält, und heißt* Punkt *auf* E*.*
$E(\mathbb{K})$ *enthält genau einen Punkt* $(x:y:z)$ *mit* $z = 0$*, nämlich den Punkt* $(0:1:0)$*; dieser Punkt heißt* Nullpunkt *der Kurve* E *und wird mit* $\mathcal{O}$ *bezeichnet.*

Hat ein Punkt $(x : y : z)$ eine z-Koordinate $\neq 0$, dann gibt es in dieser Äquivalenzklasse genau ein Tripel $(x' : y' : 1)$ (einfach $x' = \frac{x}{z}$ und $y' = \frac{y}{z}$ setzen, was immer möglich ist, da wir in einem Körper rechnen). Folglich kann jeder Punkt $(x : y : z)$ in der projektiven Ebene mit einem Punkt (x', y') aus der affinen Ebene identifiziert werden; der Nullpunkt in der affinen Ebene wird mit (∞, ∞) dem unendlich fernen Punkt identifiziert.

Beispiel 2.2 :
Sei $\mathbb{K} = \mathbb{Z}/5\mathbb{Z}$, $E = E_{1,4}$.
Dann ist:

$$\begin{aligned}
E(\mathbb{Z}/5\mathbb{Z}) &= \{(x : y : z) \in \mathbb{P}^2(\mathbb{Z}/5\mathbb{Z}) \mid y^2 z = x^3 + 1xz^2 + 4z^3\} \\
(0 : 1 : 0) &= \{(0,1,0),\ (0,2,0),\ (0,3,0),\ (0,4,0)\} \\
(1 : 4 : 3) &= \{(1,4,3),\ (2,3,1),\ (3,2,4),\ (4,1,2)\} \\
(0 : 2 : 1) &= \{(0,2,1),\ (0,4,2),\ (0,1,3),\ (0,3,4)\}
\end{aligned}$$

Den Punkten $(1 : 4 : 3)$ und $(0 : 2 : 1)$ aus der projektiven Ebene ordnet man die Punkte $(2,3)$ und $(0,2)$ in der affinen Ebene zu.

Beispiel 2.3 :
Für $\mathbb{K} = \mathbb{R}$ ist es einfach, sich eine elliptische Kurve in der affinen Ebene vorzustellen. Man setzt $z = 1$ und betrachtet Gleichung 2.1. Es gibt drei charakteristische Kurvenformen, von denen die in Abb. 2.3 gezeigte Kurve, die bekannteste ist. Daß elliptische Kurven aber auch ganz anders aussehen können, zeigen Abb. 2.1 und 2.2.

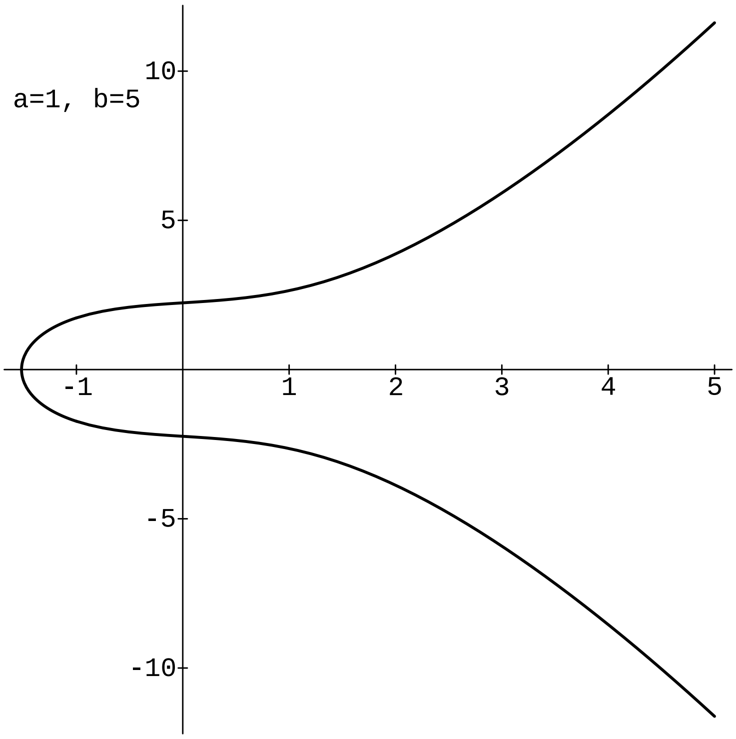


Abbildung2.1: elliptische Kurve

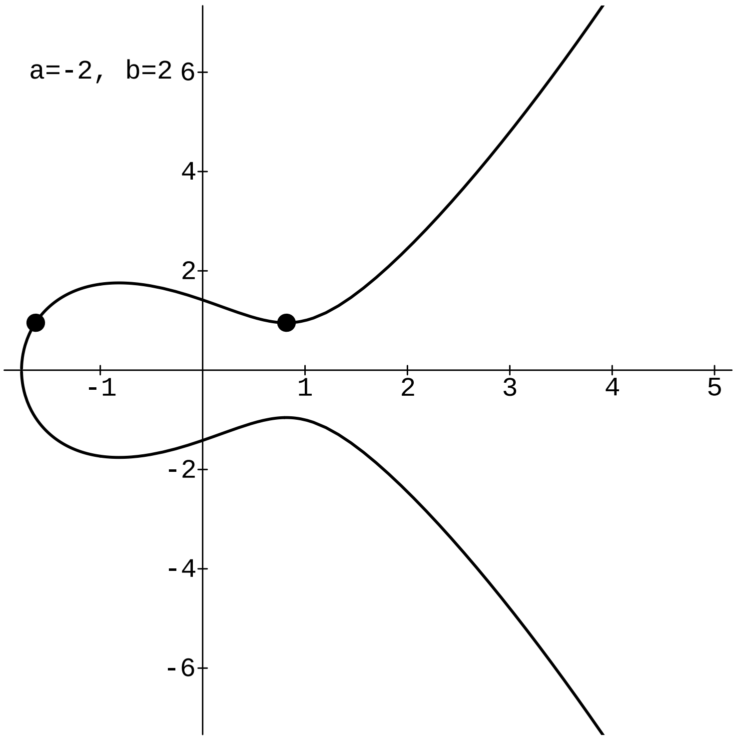


Abbildung2.2: elliptische Kurve

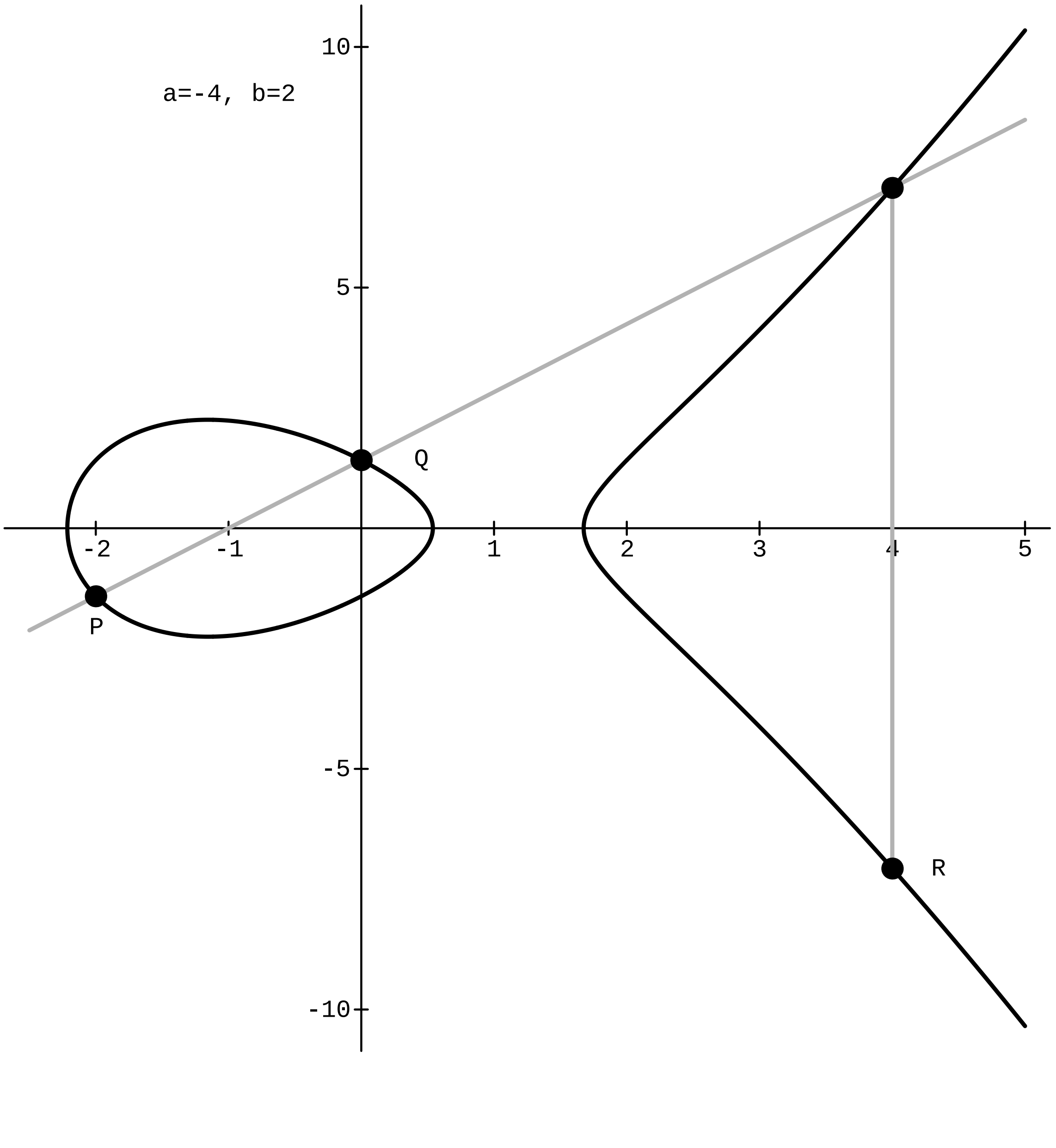


Abbildung2.3: elliptische Kurve (Addition zweier Punkte)

Die Menge $E(\mathbb{K})$ hat die Struktur einer abelschen Gruppe (s. [Koblitz]). Die Verknüpfung auf dieser Gruppe wird meistens als Addition bezeichnet und wie folgt definiert:

Definition 2.4 *(Summe zweier Punkte, Inverses eines Punktes)*
Seien E eine elliptische Kurve über $\mathbb{K}$, P und Q aus $E(\mathbb{K})$. Wir definieren das Inverse *von P und die* Summe *von P und Q mit Hilfe der folgenden Regeln:*

1. $P = \mathcal{O} \Rightarrow -P = \mathcal{O}; \quad P = (x : y : 1) \Rightarrow -P = (x : -y : 1)$
2. $\mathcal{O} + P = P + \mathcal{O} = P \quad \forall\, P \in E(\mathbb{K})$

Sind $P, Q \in E(\mathbb{K})$ ungleich dem Nullpunkt, d.h. $P = (x_P : y_P : 1)$, $Q = (x_Q : y_Q : 1)$ dann gilt:

3. $Q = -P \Leftrightarrow P + Q = \mathcal{O}$
4. *andernfalls ist $P + Q = R = (x_R : y_R : 1) \in E(\mathbb{K})$ mit*

$$x_R := m^2 - x_P - x_Q, \quad y_R := -y_P + m(x_P - x_R); \tag{2.2}$$

wenn $x_P \neq x_Q$ (d.h. $P \neq Q$ und $P \neq -Q$), dann ist

$$m := (y_Q - y_P) \cdot (x_Q - x_P)^{-1}, \tag{2.3}$$

wenn $P = Q$, dann ist

$$m := (3x_P^2 + a) \cdot (2y_P)^{-1}. \tag{2.4}$$

$\mathcal{O}$ ist das neutrale Element *der Verknüpfung.*

Zur Veranschaulichung betrachten wir Abb.2.3.
Um P und Q zu addieren, legt man eine Gerade durch P und Q. Diese Gerade hat die Steigung $m = (y_Q - y_P) \cdot (x_Q - x_P)^{-1}$ (s. 2.3) und schneidet den Graph der elliptischen Kurve in genau einem weiteren Punkt (s. [Koblitz]). $P + Q$ ist der, an der x-Achse gespiegelte, Schnittpunkt. Wäre $P = Q$, so würden wir die Tangente durch den Punkt P mit Steigung $m = (3x_P^2 + a) \cdot (2y_P)^{-1}$ benutzen. In dem Fall $P = -Q$ hätte die Linie durch P und Q eine unendliche Steigung; deshalb definiert man das neutrale Element der Addition anschaulich als den unendlich weit entfernten Punkt.

Beispiel 2.5 :
Seien $\mathbb{K} = \mathbb{R}$, $E = E_{-36,0}$, $P = (-3:9:1)$, $Q = (-2:8:1)$.
Wir berechnen $R = P + Q$, in dem wir die Koordinaten in Gleichung (2.2) einsetzen und für m Gleichung (2.3) benutzen.

$$\begin{aligned} m &= (8-9)/(-2-(-3)) = -1 \\ x_R &= 1-(-3)-(-2) = 6 \\ y_R &= -9+(-1)\cdot((-3)-6) = 0 \\ &\Rightarrow P+Q = (6:0:1) \end{aligned}$$

Um $R = 2P$ zu berechnen, benutzt man für m Gleichung (2.4) und geht wie oben vor.

Beispiel 2.6 :
Sei $K = \mathbb{Z}/5\mathbb{Z}$, $E = E_{1,4}$, $P = (0:2:1)$. Berechnung von $2 \cdot P$:

$$m = (3\cdot 0+1)/(2\cdot 2) \equiv 4^{-1} \bmod 5.$$

Um das multiplikativ Inverse von 4 zu berechnen, muß man den erweiterten euklidischen Algorithmus (ggT mit Darstellung) (s. Anhang B.1) anwenden.

$$\begin{aligned} & 5 = 1\cdot 4+1 \\ &\Rightarrow 1 = 5-1\cdot 4 \\ &\Rightarrow 4^{-1} \equiv -1 \equiv 4 \bmod 5 \\ &\Rightarrow m \equiv 4 \bmod 5 \\ x_R &= 4^2-0-0 \equiv 16 \equiv 1 \bmod 5 \\ y_R &= -2+4\cdot(0-1) \equiv -6 \equiv 4 \bmod 5 \\ &\Rightarrow 2P = (1:4:1). \end{aligned}$$

Beispiel 2.7 :
Wie addiert man die Punkte in Abb. 2.2 ?

2.2 Der Algorithmus

2.2.1 Der Grundalgorithmus

Im vorherigen Kapitel beschäftigten wir uns mit elliptischen Kurven über Körpern, für den eigentlichen Algorithmus benötigen wir aber elliptische Kurven über dem

Ring $\mathbb{Z}/N\mathbb{Z}$.
Sei $N \in \mathbb{N}$, mit $2 \nmid N$ und $3 \nmid N$.
Für $a, b \in \mathbb{Z}/N\mathbb{Z}$ betrachten wir die Kurve $E = E_{a,b}$ über $\mathbb{Z}/N\mathbb{Z}$ mit Gleichung

$$y^2 = x^3 + ax + b.$$

Die Menge der Punkte auf dieser Kurve ist:

$$E(\mathbb{Z}/N\mathbb{Z}) = \{(x : y : z) \in \mathbb{P}^2(\mathbb{Z}/N\mathbb{Z}) \mid y^2 z = x^3 + axz^2 + bz^3\}.$$

Wenn $(4a^3 + 27b^2) \in (\mathbb{Z}/N\mathbb{Z})^*$ ist, dann nennen wir E eine elliptische Kurve über $\mathbb{Z}/N\mathbb{Z}$. In diesem Fall gibt es für $E(\mathbb{Z}/N\mathbb{Z})$ ein natürliches abelsches Gruppengesetz, das allerdings durch wesentlich allgemeinere Formeln als die in Definition 2.4 beschrieben wird, denn in Gleichung (2.3) und (2.4) aus Definition 2.4 müssen Inverse berechnet werden, die aber in Ringen nicht immer existieren.
Für den Faktorisierungsalgorithmus brauchen wir aber die Gruppenstruktur nur dann, wenn die Inversen existieren.
Deshalb benutzen wir die Addition aus Definition 2.4, die nun nur noch eine partiell definierte Pseudoaddition ist, weiter und arbeiten auf einer Teilmenge V_N von $E(\mathbb{Z}/N\mathbb{Z})$. Sie besteht aus allen endlichen Punkten mit $z = 1$ und dem Nullpunkt $\mathcal{O}$:

$$V_N := \{(x : y : 1) \mid x, y \in (\mathbb{Z}/N\mathbb{Z}),\ y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}.$$

Bemerkung 2.8 :
V_N kann eine echte Teilmenge von $E(\mathbb{Z}/N\mathbb{Z})$ sein.
z.B.: $N = 30,\ a = 1,\ b = 11$
$(5 : 1 : 5) \in E(\mathbb{Z}/30\mathbb{Z})$, denn

$$\begin{aligned} 1 \cdot 5 &\equiv 5^3 + 1 \cdot 5 \cdot 5^2 + 11 \cdot 5^3 \bmod 30 \\ &\equiv 5 + 5 + 25 \equiv 35 \equiv 5 \bmod 30 \end{aligned}$$

$(5 : 1 : 5) \notin V_N$, da es kein Inverses zur z-Koordinate gibt.

Algorithmus 2.9 :

1. **Wähle** zufällig eine elliptische Kurve E und einen Punkt $P \neq \mathcal{O}$ darauf.
   Die Kurve wird bestimmt, indem man drei Zufallszahlen x, y und a aus dem Intervall $[0, N-1]$ nimmt und

   $$b = y^2 - x^3 - ax$$

berechnet. Jetzt müssen wir noch prüfen, ob wir wirklich eine elliptische Kurve modulo jeder Primzahl $p \mid N$ erzeugt haben, d.h. $4a^3+27b^2 \neq 0 \mod p,\ \forall\, p \mid N$. Das gilt genau dann, wenn der $\mathrm{ggT}(4a^3+27b^2, N) = 1$ ist, was in der Praxis fast immer der Fall ist. Liegt der ggT zwischen 1 und N liegt, so haben wir einen Teiler gefunden und sind FERTIG, andernfalls (ggT gleich N) müssen wir neue Zufallszahlen generieren.

2. **Wähle** $B_1, B_2 \in \mathbb{N}$ und berechne $k \cdot P$, mit

$$k = \prod_{\mathbb{P} \ni p \leq B_1} p^{e_p}, \quad e_p = \max\{l \in \mathbb{N} : p^l \leq B_2\}$$

Analog zur Berechnung von $a^e \mod N$ in Pollards p-1 – Algorithmus durch die "repeated squaring and multiplying" Methode (s. [Horster]) kann man auch $k \cdot P$ berechnen; hier müßte man es mit "repeated doubling and adding" bezeichnen, da statt der Multiplikation die Addition als Gruppenoperation benutzt wird. Dabei ist es nicht nötig, k explizit auszurechnen, denn wenn $k = k_1 \cdot k_2 \cdots$, dann ist $k \cdot P = (\ldots(k_2(k_1 P))\ldots)$. Ist z.B. $k = 2^4 \cdot 3^3$, dann kann man $k \cdot P$ wie folgt berechnen:

$$2P, 2(2P), 2(4P), 2(8P), 3(16)P, 3(48P), 3(144P).$$

Bei jeder Division modulo N in dieser Berechnung (s. (2.3), (2.4)) benutzt man den euklidischen Algorithmus, um das Inverse modulo N zu finden. Liefert der euklidische Algorithmus an irgendeiner Stelle der Berechnung kein Inverses, dann liegt der ggT von N und dem Nenner entweder zwischen 1 und N — in diesem Fall haben wir einen nichttrivialen Teiler von N gefunden und sind FERTIG — oder der ggT ist gleich N und wir müssen von vorne anfangen.

3. **GOTO 1**:
Die ggT-Berechnung wie in Schritt (3) des Algorithmus 1.4 ist unnötig, da die z-Koordinate von $k \cdot P$ entweder 1 oder 0 ist.

Im Gegensatz zu Algorithmus 1.4 hoffen wir hier, daß die Berechnung von $k \cdot P$ fehlschlägt, um einen Teiler von N zu finden. Wenn sich $k \cdot P$ berechnen läßt, haben wir leider keinen Erfolg, denn das Ergebnis ist unbrauchbar.

2.2.2 Die Wahl der Parameter und Laufzeit

Um die Laufzeit und optimale Parameter für ECM zu berechnen, betrachten wir zunächst die Laufzeit der einzelnen Operationen auf der Kurve. Wie bei den vorherigen Algorithmen benutzen wir als Arbeitseinheit eine Multiplikation modulo.

1. Addition zweier Punkte:
   Wir betrachten nur den Fall (4) der Definition 2.4 (S.25); in den anderen Fällen werden nur Vorzeichen bzw. Flags gesetzt. Bei der Berechnung von m in (2.3) und (2.4) ist jeweils ein Inverses modulo N zu berechnen. Dies kann man mit dem erweiterten euklidischen Algorithmus (s. Anhang B.1) tun. Nehmen wir an, daß die Berechnung eines Inversen mit dem erweiterten euklidischen Algorithmus K Multiplikationen modulo N benötigt. Somit erfordert die Berechnung von $P+Q$ $(4+K)$ Multiplikationen modulo, wenn $P=Q$ ist (2.4), und $(3+K)$ Multiplikationen modulo im anderen Fall (2.3). Ausführliche Tests mit Zahlen zwischen 100 und 150 Dezimalstellen ergaben, daß $K \approx 18$ ist.

2. Multiplikation eines Punktes mit einem Skalar:
   Hierzu kann man den binären Algorithmus (s. [Horster]) benutzen. Um mit diesem Algorithmus $k \cdot P$ zu berechnen, benötigt man zwischen $\log k$ und $2 \cdot \log k$ Punktadditionen, also im Mittel $\frac{3}{2}\log k$ Punktadditionen, wovon $\log k$ Verdopplungen (d.h. $P+P$) sind.

Für die Berechnung der optimalen Parameter benötigen wir zusätzlich noch einige Ergebnisse über die Verteilung der Primfaktoren zufälliger natürlicher Zahlen. Sei $M \in \mathbb{N}$ fest. Betrachten wir nun eine zufällig gewählte natürliche Zahl nahe bei M mit Primfaktoren $n_1 \geq n_2 \geq n_3 \geq \ldots$. Für $\alpha \geq 1$ definieren wir:

$$\rho(\alpha) := \lim_{M \to \infty} P(n_1 < M^{\frac{1}{\alpha}}), \tag{2.5}$$

wobei wir mit $P(x)$ die Wahrscheinlichkeit für das Eintreten des Ereignisses x bezeichnen. Eine präzise Definition für "eine natürliche Zahl nahe bei M" und die nun folgenden Ergebnisse über die ρ-Funktion kann man in [Knuth/Pardo] nachlesen. Anschaulich gibt die ρ-Funktion an, mit welcher Wahrscheinlichkeit der größte Teiler n_1 einer Zahl unterhalb einer gewissen Schranke ($M^{\frac{1}{\alpha}}$) liegt.

Die ρ-Funktion erfüllt die Differentialgleichung

$$\alpha\rho'(\alpha) + \rho(\alpha - 1) = 0 \tag{2.6}$$

und kann durch numerische Integration aus

$$\rho(\alpha) = \begin{cases} 1, & \text{für } 0 \leq \alpha \leq 1 \\ \dfrac{1}{\alpha} \displaystyle\int_{\alpha-1}^{\alpha} \rho(t)dt, & \text{für } \alpha > 1 \end{cases} \tag{2.7}$$

berechnet werden. Siehe dazu Algorithmus B.2 und das [Mathematica]-Programm *rho.m* (S. 88). Abbildung 2.4 zeigt das Verhalten der ρ-Funktion. Werte findet man in Tabelle 2.1 und [Knuth/Pardo] S.340.

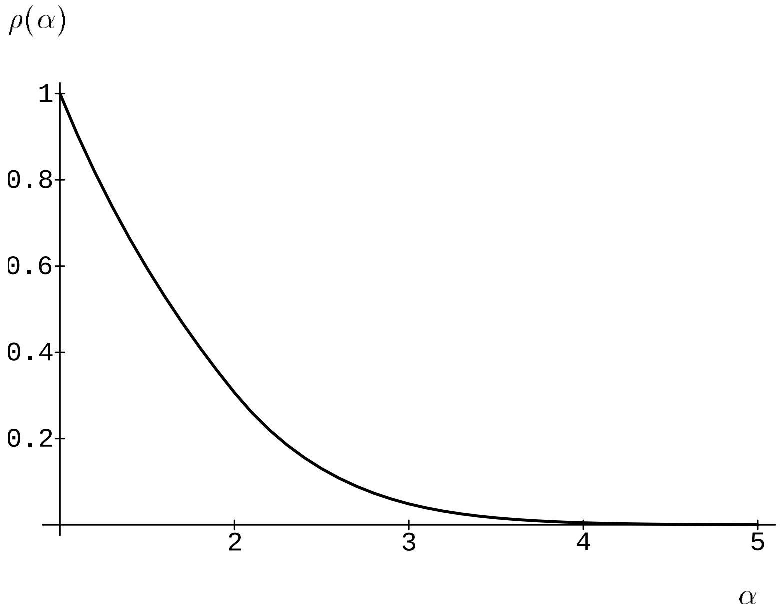


Abbildung2.4: ρ-Funktion

Tabelle2.1: *ρ-Funktion*

α	1	1.5	2	2.5	3	3.5	4
$\rho(\alpha)$	1	0.594537	0.306856	0.130322	0.0486103	0.0162306	0.00491138

Außerdem gilt für $\alpha \to \infty$:

$$\ln \rho(\alpha) = -\alpha(\ln \alpha + \ln \ln \alpha - 1) + o(\alpha) \tag{2.8}$$

$$\frac{\rho(\alpha - 1)}{\rho(\alpha)} = \alpha(\ln \alpha + O(\ln \ln \alpha)) \tag{2.9}$$

Wir wissen bereits, daß die Gruppenordung von $E(\mathbb{Z}/p\mathbb{Z})$ in $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$ liegt (s. 1.3). Für unsere weiteren Betrachtungen müssen wir annehmen, daß die Gruppenordung einer zufälligen Kurve (wie in Algorithmus 2.9 gewählt) auch eine zufällige Zahl in obigem Intervall ist. Das konnte bisher noch nicht bewiesen werden! Außerdem müssen wir die Tatsache, daß die ρ-Funktion nur für $M \to \infty$ definiert ist, vernachlässigen, da wir nicht mit unendlich großen Zahlen rechnen.
ECM findet den Faktor p von N, wenn die Gruppenordnung von $E(\mathbb{Z}/p\mathbb{Z})$ in Primfaktoren $n_1 \geq n_2 \geq \ldots$, die kleiner als B_1 sind, zerfällt. Die Wahrscheinlichkeit dafür, daß dies im ersten Versuch geschieht, ist:

$$\begin{aligned} & \rho(\alpha) = \lim_{p \to \infty} P(n_1 < p^{\frac{1}{\alpha}}) \\ \text{mit} \quad & p^{\frac{1}{\alpha}} = B_1 \\ \Leftrightarrow \quad & \frac{1}{\alpha} \ln p = \ln B_1 \end{aligned} \tag{2.10}$$

$$\Leftrightarrow \quad \alpha = \frac{\ln p}{\ln B_1}$$

Erinnern wir uns daran, daß man mit B_2 die Größe der Exponenten bestimmt (s. Algorithmus 2.9 S. 27). Nehmen wir nun einmal an, daß die Gruppenordung von $E(\mathbb{Z}/p\mathbb{Z})$ gleich $p+1+2\sqrt{p}$ ist, und nehmen wir weiter an, daß $p+1+2\sqrt{p}$ in Primfaktoren kleiner als B_1 zerfällt, denn sonst würden wir höchstwahrscheinlich keinen Faktor finden. Um alle Extremfälle abzufangen (z.B.: $p+1+2\sqrt{p}$ zerfällt in 2^n, $n \in \mathbb{N}$, oder $p+1+2\sqrt{p}$ zerfällt in $c \cdot q^2$, $c \in \mathbb{N}$, $q \in \mathbb{P}$ und q nahe bei B_1), müßten wir $B_2 = p+1+2\sqrt{p}$ wählen. Allerdings sind diese Extremfälle sehr selten, deshalb wählt man in der Praxis $B_2 = B_1$, denn das senkt die Kosten für einen Versuch erheblich, ohne die Erfolgswahrscheinlichkeit stark zu reduzieren.
Deshalb setzen wir im folgenden voraus, daß $B_1 = B_2$ ist. Dann gilt:

$$\begin{aligned} k &= \prod_{\mathbb{P} \ni p \leq B_1} p^{k_p}, \quad k_p = \max\{l \in \mathbb{N} : p^l \leq B_1\} \\ \Rightarrow k &\approx \prod_{i=1}^{\frac{B_1}{\ln B_1}} B_1 = B_1^{\frac{B_1}{\ln B_1}} \qquad \text{s. Satz A.8} \\ \Rightarrow \ln k &\approx B_1 \end{aligned} \tag{2.11}$$

Für die Berechnung von $k \cdot P$ benötigen wir:

$$\begin{aligned} & (4+K) \cdot \log k \;+\; (3+K)\frac{1}{2}\log k \quad \text{Multiplikationen modulo} \\ & \text{(s. Punkt 2 in Kapitel 2.2.2)} \\ = \;& \left(4 + K + \frac{3}{2} + \frac{K}{2}\right) \log k = \frac{3}{2}\left(\frac{11}{3} + K\right) \log k \\ = \;& \frac{3}{2}\left(\frac{11}{3} + K\right) \frac{\ln k}{\ln 2} = \frac{3}{2 \ln 2}\left(\frac{11}{3} + K\right) \ln k \\ \approx \;& \frac{3}{2 \ln 2}\left(\frac{11}{3} + K\right) \cdot B_1 \quad \text{(wegen 2.11)} \\ = \;& c_1 \cdot B_1, \text{ mit } \quad c_1 := \frac{3}{2 \ln 2}\left(\frac{11}{3} + K\right). \end{aligned} \tag{2.12}$$

Da die Wahrscheinlichkeit, daß wir p mit einem Versuch finden, $\rho(\alpha)$ ist, müssen wir im Mittel $\frac{1}{\rho(\alpha)}$ Versuche machen. Der Aufwand W_1, d.h. die Anzahl der Multiplikationen modulo, ist demnach:

$$W_1(\alpha) = \frac{c_1 \cdot B_1}{\rho(\alpha)} \approx \frac{c_1 \cdot p^{\frac{1}{\alpha}}}{\rho(\alpha)}. \tag{2.13}$$

Um $W_1(\alpha)$ zu minimieren, differenzieren wir $W_1()$ und setzen das Ergebnis gleich Null.

$$W_1'(\alpha) = \frac{c_1 p^{\frac{1}{\alpha}} \cdot \frac{-1}{\alpha^2} \ln p \cdot \rho(\alpha) - c_1 p^{\frac{1}{\alpha}} \rho'(\alpha)}{\rho(\alpha)^2} = 0$$

$$
\begin{aligned}
\Leftrightarrow\quad & \frac{-1}{\alpha^2}\ln p \cdot \rho(\alpha) - \rho'(\alpha) = 0 \\
\Leftrightarrow\quad & \ln p = -\alpha^2 \cdot \frac{\rho'(\alpha)}{\rho(\alpha)} \\
\Leftrightarrow\quad & \ln p = \alpha\frac{\rho(\alpha-1)}{\rho(\alpha)} \qquad \text{wegen Definition von } \rho(\alpha) \text{ s. (2.6)} \qquad (2.14)
\end{aligned}
$$

In der Praxis ist p im voraus nicht bekannt. Nehmen wir trotzdem einmal an, daß wir die Größenordnung von p aus irgendeiner Quelle kennen. Dann können wir die optimalen Parameter für ECM berechnen, in dem wir α so wählen, daß Gleichung(2.14) gilt und dann mit (2.10) B_1 ausrechnen. α findet man z.B. durch Intervallschachtelung.

Beispiel 2.10 Optimale Parameter zum Finden von 18-stelligen Faktoren.
Sei $p \approx 5 \cdot 10^{17} \ \Rightarrow \ln p \approx 40.7534$
Wähle α so, daß gilt:

$$\ln p \approx \alpha\frac{\rho(\alpha-1)}{\rho(\alpha)} \qquad \text{(s. 2.14)}$$

Das erreicht man z.B. durch Intervallschachtelung. Hier ergibt sich:

$$
\begin{aligned}
& \alpha \approx 4.05 \\
& \Rightarrow B_1 = p^{\frac{1}{\alpha}} \approx 23449 \qquad \text{(s. 2.10)}
\end{aligned}
$$

Wie bereits erwähnt, wählen wir $B_2 = B_1$. Im Mittel muß man mit diesen Parametern

$$\frac{1}{\rho(\alpha)} \approx 231$$

Versuche machen, um einen 18-stelligen Faktor zu finden; das entspricht

$$\frac{c_1 \cdot B_1}{\rho(\alpha)} \approx 2.539 \cdot 10^8$$

Multiplikationen modulo, wobei $c_1 \approx 46.89$ (s. 2.12) mit $K \approx 18$ (s. Kapitel 2.2.2 1).

In Tabelle 2.2 findet man auch für andere Faktoren optimale Parameter.

Leider ist normalerweise die Größenordnung von p nicht bekannt. [Brent] und [Lenstra] schlagen deshalb vor, ECM mit niedrigen Parametern zu starten und diese dann schrittweise zu erhöhen. Meiner Erfahrung nach ist dieses Vorgehen aber nur

Tabelle2.2: *optimale Parameter für ECM*

#Stellen	α	B_1	#Kurven
6	2.63	147	10
9	3.07	682	24
12	3.45	2 462	55
15	3.75	8 318	111
18	4.05	23 462	231
21	4.31	63 502	445
24	4.55	161 730	833
27	4.77	395 808	1 501
30	4.97	945 922	2 594

mit großer Schrittweite effizient (z.B. optimale Parameter für 10, 20 und 28 Stellen), da kleinere Faktoren auch schnell von ECM-Läufen mit hohem B_1 gefunden werden, umgekehrt jedoch nicht.
Wie wichtig eine gute Parameterwahl ist, zeigt Abb. 2.5.

Schätzen wir nun noch die Laufzeit von ECM ab, in dem wir $\alpha \to \infty$ betrachten; vergessen wir dabei aber nicht, daß α in dem für uns interessanten Bereich zwischen 3 und 5 liegt (siehe Tabelle 2.2). Wir wissen bereits:

$$\begin{aligned} \ln p &= \alpha \frac{\rho(\alpha-1)}{\rho(\alpha)} && \text{(s. 2.14)} \\ &= \alpha(\ln\alpha + O(\ln\ln\alpha)) && \text{(einsetzen von 2.9)} \end{aligned}$$

Daraus folgt, daß

$$\alpha \approx \sqrt{\frac{2\ln p}{\ln\ln p}} \tag{2.15}$$

ist, was man mit der Grenzwertbetrachtung

$$\lim_{\alpha\to\infty} \frac{\alpha}{\sqrt{\frac{2\ln p}{\ln\ln p}}} = 1$$

nachrechnen kann. Weiterhin gilt:

$$\begin{aligned} \ln W_1(\alpha) &\approx \ln\left(\frac{c_1 p^{\frac{1}{\alpha}}}{\rho(\alpha)}\right) && \text{(s. 2.13)} \\ &= \ln c_1 + \frac{1}{\alpha}\ln p - \ln\rho(\alpha) \end{aligned}$$

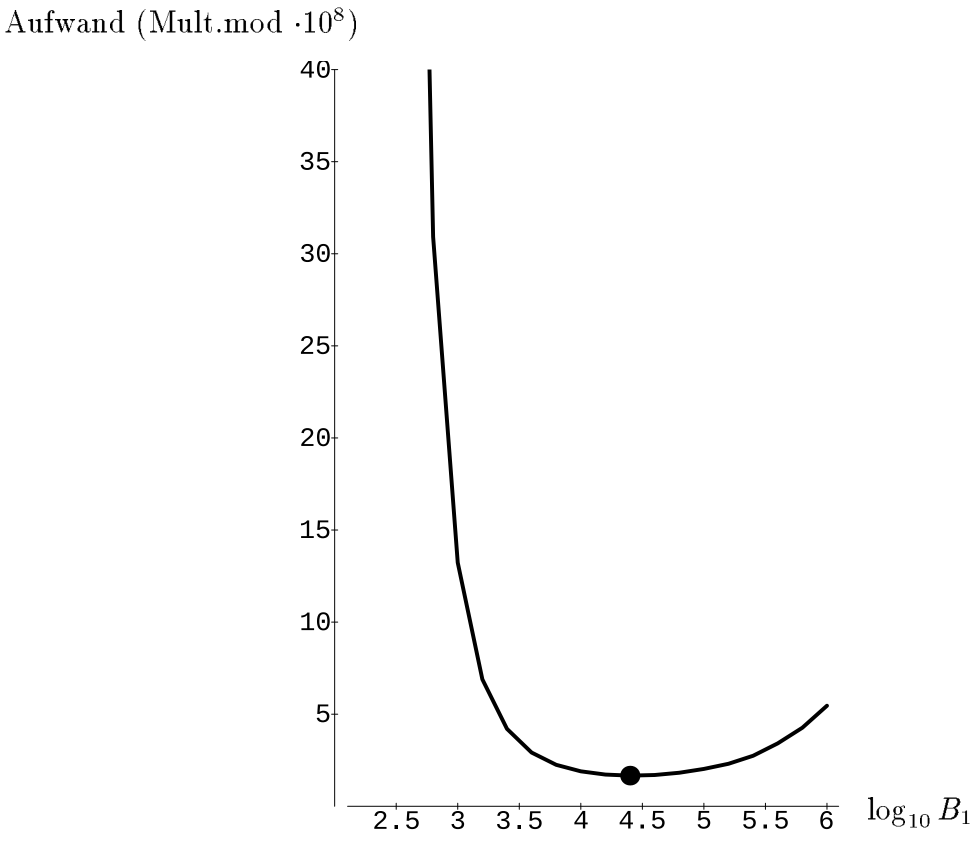


Abbildung2.5: Finden eines 18 stelligen Faktors in Abhängigkeit von B_1

$$
\begin{aligned}
&= \ln c_1 + \frac{\rho(\alpha-1)}{\rho(\alpha)} - \ln \rho(\alpha) \qquad\qquad \text{(wegen 2.14)} \\
&\approx \frac{\rho(\alpha-1)}{\rho(\alpha)} - \ln \rho(\alpha) \\
&= \alpha(\ln\alpha + O(\ln\ln\alpha)) + \alpha(\ln\alpha + \ln\ln\alpha - 1) - o(\alpha) \\
&\quad \text{(einsetzen von 2.8 und 2.9)} \\
&\approx 2\alpha\ln\alpha \\
&\approx 2\sqrt{\frac{2\ln p}{\ln\ln p}}\ \ln\sqrt{\frac{2\ln p}{\ln\ln p}} \qquad\qquad \text{(einsetzen von 2.15)} \\
&= \sqrt{\frac{2\ln p}{\ln\ln p}}\,(\ln 2 + \ln\ln p - \ln\ln\ln p) \\
&\approx \sqrt{\frac{2\ln p}{\ln\ln p}}\ \ln\ln p
\end{aligned}
$$

$$= \sqrt{2 \ln p \ln \ln p}$$

Die Laufzeit von ECM ist somit:

$$T_1(p) = W_1(\alpha) = e^{\sqrt{2 \ln p \ln \ln p}\ (1+o(1))} \tag{2.16}$$

2.2.3 Beschleunigung von ECM

In diesem Kapitel werden zwei wichtige Tricks beschrieben, die die Geschwindigkeit von ECM erheblich steigern. Zum einen geht es hierbei um die Beschleunigung der Addition zweier Punkte und zum andern um die geschickte Wahl einer Kurve.

Beschleunigung der Addition

Wie wir in Kapitel 2.2.2 (1) sahen, stellt die Inversenberechnung mit zirka 18 Multiplikationen modulo den größten Anteil der Kosten für eine Addition zweier Punkte. Mit folgendem Trick (s. [Montgomery]) kann man diese Kosten auf asymptotisch drei Multiplikationen modulo drücken.
Nehmen wir dazu an, daß wir die Inversen von x_1, x_2, x_3 und x_4 berechnen wollen. Zuerst berechnen wir dazu das Produkt $x_1 \cdot x_2 \cdot x_3 \cdot x_4$ und merken uns die Zwischenergebnisse. Also:

$$\begin{array}{lcll}
z_1 & = & x_1 & \\
z_2 & = & z_1 \cdot x_2 & \text{(1 Mul.mod.)} \\
z_3 & = & z_2 \cdot x_3 & \text{(1 Mul.mod.)} \\
z_4 & = & z_3 \cdot x_4\ (= x_1 \cdot x_2 \cdot x_3 \cdot x_4) & \text{(1 Mul.mod.)}
\end{array}$$

Nun ermittelt man mit Hilfe des erweiterten euklidischen Algorithmus das Inverse von z_4 und damit die Inversen von x_1, x_2, x_3, x_4 wie folgt:

$$\begin{array}{lcll}
x_4^{-1} & = & z_3 \cdot z_4^{-1} & \text{(1 Mul.mod.)} \\
z_3^{-1} & = & x_4 \cdot z_4^{-1} & \text{(1 Mul.mod.)} \\
x_3^{-1} & = & z_2 \cdot z_3^{-1} & \text{(1 Mul.mod.)} \\
z_2^{-1} & = & x_3 \cdot z_3^{-1} & \text{(1 Mul.mod.)} \\
x_2^{-1} & = & z_1 \cdot z_2^{-1} & \text{(1 Mul.mod.)} \\
z_1^{-1} & = & x_2 \cdot z_2^{-1} & \text{(1 Mul.mod.)} \\
x_1^{-1} & = & z_1^{-1} &
\end{array}$$

Insgesamt benötigt man jetzt zum Ausrechnen der Inversen einen erweiterten ggT und neun Multiplikationen modulo, dabei ist x_1^{-1} quasi kostenlos abgefallen. Die Berechnung von n Inversen erfordert somit $(n-1) \cdot 3$ Multiplikationen modulo plus den Kosten für einen erweiterten ggT, die man vernachlässigen kann, wenn n groß

ist; d.h.: Eine Zahl modulo N kann mit asymptotisch drei Multiplikationen modulo invertiert werden (s. Tabelle2.3).
Auf ECM angewandt bedeutet dies, daß man mehrere Kurven gleichzeitig betrachten muß und alle Inversen auf einmal ausrechnet. Ist z_4 nicht invertierbar, so ist der $\mathrm{ggT}(z_4, N) \neq 1$ und damit ein nichttrivialer Teiler von N. Bei der Parameterberechnung ändert sich dadurch die Konstante

$$c_1 := \frac{3}{2 \ln 2} \left(\frac{11}{3} + K \right) \qquad \text{(s. 2.12)},$$

denn die Anzahl K der Multiplikationen modulo für eine Invertierung verringert sich von 18 (s. Kapitel 2.2.2 (1)) auf 3, was ECM enorm beschleunigt.

Tabelle2.3: *Beschleunigung durch ggT-Trick*

#Inverse gleichzeitig	#Mul.mod. pro Invertierung	c_1	Geschw. Faktor
1	18.	46.89	1.0
2	10.5	30.66	1.5
3	8.	25.25	1.9
5	6.	20.92	2.2
10	4.5	17.67	2.7
20	3.75	16.05	2.9
30	3.5	15.51	3.0
50	3.3	15.08	3.1
100	3.15	14.75	3.2
1000	3.015	14.46	3.2
1000000	3.00001	14.43	3.2

Der Geschwindigkeitsfaktor berechnet sich aus:

$$\frac{W_{1_{max}}}{W_1} = \frac{c_{1_{max}} B_1}{c_1 B_1} = \frac{c_{1_{max}}}{c_1}.$$

Der Geschwindigkeitsfaktor 3.2 wird auch in der Praxis knapp erreicht. In meiner Implementierung beschleunigt dieser Trick ECM um den Faktor 2.9, wenn 20 Kurven gleichzeitig und Zahlen mit 100 Dezimalstellen benutzt werden; rechnet man mit 80 Kurven gleichzeitig, so erreicht man einen Speedup von 3.1.

Geschickte Wahl einer Kurve

In Schritt (1) des ECM-Algorithmus 2.9 wählen wir eine zufällige elliptische Kurve. [Montgomery] schlägt folgendes verbessertes Verfahren vor:
Wähle drei verschiedene Quadrate $s_1^2,\ s_2^2,\ s_3^2;\ s_1, s_2, s_3 \in \mathbb{N}$ und benutze den Punkt

$$(x_1, y_1) = (\frac{s_1^2 + s_2^2 + s_3^2}{3},\ s_1 s_2 s_3)$$

auf der Kurve

$$\begin{aligned} y^2 &= (x + s_1^2 - x_1)(x + s_2^2 - x_1)(x + s_3^2 - x_1) \\ \Leftrightarrow y^2 &= x^3 + x\,\underbrace{(a_1 a_2 + a_1 a_3 + a_2 a_3)}_{=a} + \underbrace{a_1 a_2 a_3}_{=b} \end{aligned}$$

mit $a_i = s_i^2 - x_1,\ i = 1, 2, 3$. Die Gruppenordung, der durch diese Kurve generierte Gruppe ist durch 4 teilbar, was p auf $p/4$ in den Abschätzungen von Kapitel 2.2.2 reduziert. Die Praxis zeigt, daß diese Kurve immer noch ausreichend zufällig für ECM ist.

Tabelle2.4: *optimale Parameter für ECM mit verbessertem Kurvengenerator*

#Stellen	α	B_1	#Kurven
6	2.53	104	8
9	2.99	511	20
12	3.37	1 964	46
15	3.69	6 616	96
18	3.99	19 284	199
21	4.25	53 571	382
24	4.49	139 414	711
27	4.73	329 257	1 347
30	4.93	798 397	2 323

Im nächsten Kapitel stelle ich noch eine weitere Möglichkeit vor, ECM zu beschleunigen.

Kapitel 3

Die Continuations (Fortsetzungen)

Im vorherigen Kapitel habe ich einige Tricks beschrieben, die den Grundalgorithmus beschleunigen; es gibt aber noch eine völlig andere Methode, ECM zu beschleunigen. Betrachten wir dazu den Fall, daß ein Durchlauf des ECM-Algorithmus erfolglos war. Dann haben wir folgendes berechnet:

$$Q = kP \quad \text{mit} \quad k = \prod_{\mathbb{P} \ni p \leq B_1} p^{k_p}, \quad k_p = \max\{l \in \mathbb{N} : p^l \leq B_2\}.$$

Sei $\mathrm{ord}_{E(\mathbb{Z}/N\mathbb{Z})}(P) = n_1 \cdot n_2 \cdots n_l$ mit $n_i \in \mathbb{P}$, $i = 1, \ldots l$; $n_1 \geq n_2 \geq \ldots \geq n_l$. Fanden wir keinen Teiler, weil $n_1 > B_1$ (d.h. $n_1 \nmid k$), dann können wir eine sogenannte Continuation anschließen. Hierbei wird mit schnellen Verfahren nach dem größten Teiler n_1 der Ordnung von P gesucht. Eine Continuation kann also nur dann erfolgreich sein, wenn

$$n_2 \cdots n_l \mid k \quad \text{und} \quad n_1 \nmid k.$$

3.1 Die Standard-Continuation

Ursprünglich von Pollard für seine p-1 – Methode vorgeschlagen, wurde sie von [Montgomery] auf ECM übertragen.

3.1.1 Der Algorithmus

Sei $B_1 < n_1 < B_3$, für ein $B_3 \in \mathbb{N}$ groß genug. Wir testen nun mit jeder Primzahl q zwischen B_1 und B_3, ob

$$q \cdot Q = q \cdot kP = \mathcal{O} \text{ in } E(\mathbb{Z}/p\mathbb{Z})$$

ist. Das erledigt im Prinzip der folgende Algorithmus:

Algorithmus 3.1 :

$q \leftarrow$ `erste Primzahl` $> B_1$

<u>`while`</u> $(q < B_3)$

  $Q \leftarrow q \cdot Q$

  <u>`if`</u> `Teiler von` N `während der obigen Multiplikation gefunden`

  <u>`then`</u> `FERTIG`

  $q \leftarrow$ `nächste_Primzahl()`

<u>`endwhile`</u>

Jede Primzahl q zwischen B_1 und B_3 wird hier mit einem Punkt multipliziert; das entspricht $\frac{3}{2} \log q$ Punktadditionen (s. Kapitel 2.2.2) pro Primzahl. Das ist natürlich kein Fortschritt gegenüber dem ECM-Grundalgorithmus. Mit einem einfachen, überaus eleganten Trick lassen sich jedoch die Kosten drastisch senken. Dazu nutzt man die folgende Beobachtung:

Die Differenz zweier aufeinanderfolgender Primzahlen ist klein.

Ist q_j die j-te Primzahl, so scheint die Differenz $q_{j+1} - q_j$ $\mathrm{O}((\log q_j)^2)$ nicht zu überschreiten (s. [Montgomery]). Konkret ist die maximale Differenz zweier aufeinanderfolgender Primzahlen bis 10 000 000 gleich 154. Die längste bekannte Primzahllücke (1982) schließt sich an die Primzahl 61 003 096 898 749 an, wonach genau 681 zusammengesetzte Zahlen folgen (s. [Bundschuh]).
Wir speichern nun $i \cdot Q$ für $i = 2, \ldots,$maximale Primzahldifferenz in einer Tabelle. Es gilt:

$$q_{j+1} \cdot Q = q_j \cdot Q + (q_{j+1} - q_j) \cdot Q. \tag{3.1}$$

$q_j \cdot Q$ kennen wir aus dem Schritt vorher und $(q_{j+1} - q_j) \cdot Q$ finden wir in der Tabelle; also können wir $q_{j+1} \cdot Q$ mit einer Punktaddition berechnen.

Algorithmus 3.2 Standard Continuation

```
      /* berechne Tabelle */
      Tabelle[0] ← 𝒪
(1)   for i ← 1 to MAX_PRIMZAHLDIFFERENZ
            Tabelle[i] ← Tabelle[i − 1] + Q
            /* Q = kP am Ende von ECM */

(2)   /* berechne q_j · Q, Startwert für Gleichung 3.1 */
      q_alt ← erste Primzahl > B_1
      Q = q_alt · Q

(3)   /* Iteration */
      q ←nächste_Primzahl()
      while(q < B_3)
            Q = Q+ Tabelle[q − q_alt]
            if Teiler von N während obiger Addition gefunden
            then FERTIG
            q_alt ← q
            q ←nächste_Primzahl()
      endwhile
```

3.1.2 Die Wahl der Parameter und Laufzeit

<u>Laufzeit:</u> Die Laufzeit wird im wesentlichen von der Iteration in Schritt (3) bestimmt.

$$\begin{array}{lll} \text{Iteration (3):} & \pi(B_3) - \pi(B_1) & \text{Punktadditionen} \\ & \approx \dfrac{B_3}{\ln B_3} - \dfrac{B_1}{\ln B_1} & \text{Punktadditionen} \\ & = \left(\dfrac{B_3}{\ln B_3} - \dfrac{B_1}{\ln B_1}\right) \underbrace{3+K}_{:=c_2} & \text{Mult. modulo (s. Kapitel 2.2.2)} \end{array}$$

da die Punkte, die bei der Iteration addiert werden ungleich sind.
Bei dem Grundalgorithmus wählten wir $B_2 = B_1$ (s. S. 31). Benutzen wir aber ECM mit Standard-Continuation, so ist es meiner Erfahrung nach besser, $B_2 = B_3$

zu setzen. Die beiden großen Faktoren (s. Kapitel 6.2) wurden mit dieser Parameterwahl gefunden.
Um die Kosten des Grundalgorithmus zu bestimmen, müssen wir zuerst k berechnen (vgl. S. 31).

$$k = \prod_{\mathbb{P} \ni p \leq B_1} p^{k_p}, \quad k_p = \max\{l \in \mathbb{N} : p^l \leq B_3\}$$

Für p^{k_p} mit $p \leq \sqrt{B_3}$ gilt: $\quad p^{k_p} \approx B_3$.
Für p^{k_p} mit $\sqrt{B_3} < p \leq B_1$ gilt: $\quad p^{k_p} \approx B_1$.
Somit gilt:

$$\begin{aligned}
k &\approx \prod_{i=1}^{\frac{\sqrt{B_3}}{\ln\sqrt{B_3}}} B_3 \cdot \prod_{j=1}^{\frac{B_1}{\ln B_1} - \frac{\sqrt{B_3}}{\ln\sqrt{B_3}}} B_1 \\
&= B_3^{\frac{\sqrt{B_3}}{\ln\sqrt{B_3}}} \cdot B_1^{\frac{B_1}{\ln B_1} - \frac{\sqrt{B_3}}{\ln\sqrt{B_3}}} \\
\Rightarrow \ln k &\approx \frac{\sqrt{B_3}}{\ln\sqrt{B_3}} \ln B_3 + \left(\frac{B_1}{\ln B_1} - \frac{\sqrt{B_3}}{\ln\sqrt{B_3}}\right) \ln B_1 \\
&= 2\sqrt{B_3} + B_1 - \frac{\sqrt{B_3} \cdot \ln B_1}{\ln\sqrt{B_3}} \\
&= B_1 + 2\sqrt{B_3}\left(1 - \frac{\ln B_1}{\ln B_3}\right) \qquad (3.2)
\end{aligned}$$

Ein Durchlauf von ECM mit Standard-Continuation kostet somit

$$c1 \cdot \left(B_1 + 2\sqrt{B_3}\left(1 - \frac{\ln B_1}{\ln B_3}\right)\right) + c_2 \cdot \left(\frac{B_3}{\ln B_3} - \frac{B_1}{\ln B_1}\right)$$

Multiplikationen modulo.

Um die Erfolgswahrscheinlichkeit eines Versuches und daraus die Anzahl, der im Mittel benötigten Versuche, um einen Faktor gegebener Länge zu bestimmen, ausrechnen zu können, benötigen wir noch einige Erkenntnisse aus der Zahlentheorie. Wie in Kapitel 2.2.2 Gleichung (2.5) sei $M \in \mathbb{N}$ fest. Wir betrachten eine natürliche Zahl nahe bei M mit Primfaktoren $n_1 \geq n_2 \geq \ldots;\ n_i \in \mathbb{P}$. Für $\alpha - 1 \geq \beta \geq 1$ definieren wir (s. [Brent], [Knuth/Pardo])

$$\mu(\alpha, \beta) := \lim_{M \to \infty} P(n_2 < M^{\frac{1}{\alpha}},\ n_1 < M^{\frac{\beta}{\alpha}}) \qquad (3.3)$$

Anschaulich gibt die μ-Funktion die Wahrscheinlichkeit dafür an, daß der zweitgrößte Primfaktor n_2 kleiner einer Schranke $M^{\frac{1}{\alpha}}$ (in unserm Fall B_1) und der größte Primfaktor der Zahl kleiner der Schranke $M^{\frac{\beta}{\alpha}}$ (in unserm Fall B_3) ist.
Die μ-Funktion kann mit Hilfe der in Kapitel 2.2.2 definierten ρ-Funktion durch numerische Integration berechnet werden (s. Anhang D.1.2):

$$\mu(\alpha, \beta) = \rho(\alpha) + \int\limits_{\alpha-\beta}^{\alpha-1} \frac{\rho(t)}{\alpha - t} dt.$$

Kehren wir nun wieder zu der Parameterberechnung zurück. Die Wahrscheinlichkeit dafür, daß unser 2-Phasen-Algorithmus (ECM + Continuation) einen Faktor p von N findet, ist:

$$\mu(\alpha, \beta) := \lim_{p \to \infty} P(n_2 < p^{\frac{1}{\alpha}},\ n_1 < p^{\frac{\beta}{\alpha}}) \quad \text{mit } p^{\frac{1}{\alpha}} = B_1,\ p^{\frac{\beta}{\alpha}} = B_3.$$

Die erwarteten Kosten, den Faktor p zu finden, sind somit:

$$W_2(\alpha, \beta) = \frac{c1 \cdot \left(B_1 + 2\sqrt{B_3}\left(1 - \frac{\ln B_1}{\ln B_3}\right)\right) + c_2 \cdot (\frac{B_3}{\ln B_3} - \frac{B_1}{\ln B_1})}{\mu(\alpha, \beta)}, \tag{3.4}$$

da wir im Mittel $\frac{1}{\mu(\alpha,\beta)}$ Versuche machen müssen. Um optimale Parameter zu berechnen, müssen wir das Minimum von $W_2(\alpha, \beta)$ finden. Das erledigt das Mathematica-Programm ecm2par (s. Anhang D.2.2). Die Konstanten

$$\begin{aligned} c_1 &= \frac{3}{2 \ln 2}\left(\frac{11}{3} + K\right) \\ c_2 &= 3 + K \end{aligned}$$

haben wir bisher theoretisch abgeschätzt und dabei nur die Anzahl der Multiplikationen modulo berücksichtigt. Wesentlich realistischere Werte erhält man, wenn man die Konstanten anhand konkreter Programmläufe ermittelt. Beispielsweise könnte man versuchen eine Primzahl zu faktorisieren (dann macht ECM mit Sicherheit einen erfolglosen Versuch). Im ersten Versuch wählen wir Parameter B_1, B_3 und messen die Zeit t, im zweiten Versuch wählen wir Parameter B_1', B_3' und messen die Zeit t'. In beiden Versuchen muß man natürlich die gleiche Anzahl Kurven betrachten! Dann löst man das lineare Gleichungssystem

$$\begin{aligned} c1 \cdot \left(B_1 + 2\sqrt{B_3}\left(1 - \frac{\ln B_1}{\ln B_3}\right)\right) + c_2 \cdot (\frac{B_3}{\ln B_3} - \frac{B_1}{\ln B_1}) &= t \\ c1 \cdot \left(B_1' + 2\sqrt{B_3'}\left(1 - \frac{\ln B_1'}{\ln B_3'}\right)\right) + c_2 \cdot (\frac{B_3'}{\ln B_3'} - \frac{B_1'}{\ln B_1'}) &= t' \end{aligned}$$

und erhält somit "Praxiswerte" für c_1 und c_2. Es empfiehlt sich, viele Versuche mit stark unterschiedlichen Parametern zu machen und dann die Werte für c_1 und c_2 zu mitteln.

3.2 Die verbesserte Standard-Continuation

Mit etwas Geschick und vielen Vorberechnungen kann man die Punktaddition aus der Standard-Continuation durch eine Multiplikation modulo ersetzen.

3.2.1 Der Algorithmus

Wie im vorherigen Kapitel suchen wir wieder eine Primzahl q zwischen B_1 und B_3, für die gilt:

$$q \cdot Q = \mathcal{O} \quad \text{in } E(\mathbb{Z}/p\mathbb{Z}).$$

$Q(= kP)$ ist das Ergebnis aus dem vorherigen ECM-Lauf. q läßt sich darstellen als

$$q = vw - u; \quad u, v, w \in \mathbb{N}$$

mit $w = \lceil\sqrt{B_3}\rceil, \quad 0 \leq u < w, \quad \lfloor \frac{B_1}{w} \rfloor \leq v \leq \lceil \frac{B_3}{w} \rceil.$

Nun gilt:

$$\begin{aligned}
& q \cdot Q = \mathcal{O} \qquad \text{in } E(\mathbb{Z}/p\mathbb{Z}) \\
\Leftrightarrow\ & (vw - u) \cdot Q = \mathcal{O} \\
\Leftrightarrow\ & vw \cdot Q - u \cdot Q = \mathcal{O} \\
\Leftrightarrow\ & (x_1, y_1) := vw \cdot Q \quad = \quad u \cdot Q =: (x_2, y_2) \qquad \text{in } E(\mathbb{Z}/p\mathbb{Z}) \\
\Rightarrow\ & y_1 \equiv y_2 \bmod p \\
\Leftrightarrow\ & y_1 - y_2 \equiv 0 \bmod p \\
\Leftrightarrow\ & p \mid (y_1 - y_2) \\
\Rightarrow\ & \mathrm{ggT}(y_1 - y_2, N) \geq p.
\end{aligned}$$

Die gleichen Überlegungen gelten auch für die x-Koordinaten der beiden Punkte. Da $(x, -y)$ das Inverse von (x, y) ist, testen wir mit dem $\mathrm{ggT}(x_1 - x_2, N)$ gleichzeitig die Inversen der beiden Punkte, was die Erfolgswahrscheinlichkeit ohne zusätzlichen Aufwand erhöht.

Algorithmus 3.3 Verbesserte Standard-Continuation

(1) `berechne` $w \leftarrow \lceil\sqrt{B_3}\rceil$

(2) `berechne` $v_1 \leftarrow \left\lfloor \dfrac{B_1}{w} \right\rfloor, \quad v_3 \leftarrow \left\lceil \dfrac{B_3}{w} \right\rceil$

```
(3)  berechne die Werte von
         u·Q,  ∀u mit 0 ≤ u < w und
         vw·Q, ∀v mit v1 ≤ v ≤ v3
     und speichere sie in einer Tabelle ab.

(4)  q ← erste Primzahl > B1
     while (q < B3)
         v ← ⌈B3/w⌉;   u ← vw − q     /* reine Integer-Operationen */
         c ← x-Koordinate(vw·Q) - x-Koordinate(u·Q)
         if ggT(c, N) > 1 then FERTIG
         q ← nächste_Primzahl()
     endwhile
```

Leider ist der Speicherplatzbedarf sehr groß. Wählt man $B_1 = 500\,000$, $B_3 = 30\,000\,000$ und benutzt 20 Kurven gleichzeitig um eine 100-stellige Zahl zu faktorisieren, so benötigt man 9.5 Megabyte, um beide Tabellen abzuspeichern. Mit den beiden folgenden Tricks kann man den Speicherverbrauch stark reduzieren:

1. Das Speichern der $vw \cdot Q$ entfällt, wenn man die Primzahlen in auf- bzw. absteigender Reihenfolge benutzt, da man $(v+1)w \cdot Q = vw \cdot Q + w \cdot Q$ bei Bedarf direkt berechnen kann.

2. Die $u \cdot Q$ müssen nur gespeichert werden, wenn der $\mathrm{ggT}(u, w) = 1$ ist, da $p = vw - u$ eine Primzahl ist.

Diese Tatsache läßt sich folgendermaßen zur Speicherplatzeinsparung benutzen. Man wähle w so, daß es durch einige kleine Primzahlen teilbar ist.
z.B.: $w = 2 \cdot 3 \cdot 5 \cdot c = 30 \cdot c,\ c \in \mathbb{N}$.
Jetzt muß man nur noch die $u \cdot Q$ speichern, für die u weder durch 2 noch durch 3 oder 5 teilbar ist, und das sind

$$\frac{\varphi(30)}{30} w = \frac{8}{30} w = 0.2\overline{6} w$$

viele. Wählt man z.B.: $w = 30c$, so reduziert das den Speicherbedarf von 9.5 Megabyte auf 1.3 Megabyte.
Ist die Tafel immer noch zu groß, so kann man w einfach kleiner wählen, wodurch sich allerdings die Rechenzeit erhöht.

3.2.2 Die Wahl der Parameter und Laufzeit

Laufzeit:

$$
\begin{array}{lll}
\text{Tafel berechnen (3):} & O(\sqrt{B_3}) & \text{Punktadditionen} \\
\text{Iteration (4):} & \pi(B_3) - \pi(B_1) & \text{ggT s} \\
 & \approx \dfrac{B_3}{\ln B_3} - \dfrac{B_1}{\ln B_1} & \text{ggT s}
\end{array}
$$

Mit einem kleinen Trick können wir die Kosten für den ggT auf eine Multiplikation modulo drücken. Betrachten wir dazu:

$$
\begin{array}{ll}
 & \mathrm{ggT}(c, N) = d \\
\Rightarrow & d \mid c \\
\Rightarrow & d \mid k \cdot c, \text{ für beliebiges } k \in \mathbb{N}.
\end{array}
$$

Die Teilereigenschaft von q geht nicht verloren, wenn wir die c aufmultiplizieren und z.B. nur jedes 10. Mal den ggT berechnen. Asymptotisch gesehen, fallen dann die Kosten für den ggT unter den Tisch und wir müssen in der Iteration nur einmal zusätzlich multiplizieren. Somit kostet die Iteration

$$
\frac{B_3}{\ln B_3} - \frac{B_1}{\ln B_1} \quad \text{Multiplikationen modulo.}
$$

Die erwarteten Kosten, den Faktor zu finden, sind somit

$$
W_2(\alpha, \beta) = c1 \cdot \left(B_1 + 2\sqrt{B_3} \left(1 - \frac{\ln B_1}{\ln B_3} \right) \right) + c_2 \cdot \sqrt{B_3} + c_3 \cdot \left(\frac{B_3}{\ln B_3} - \frac{B_1}{\ln B_1} \right). \tag{3.5}
$$

Optimale Parameter berechnet man genauso wie in Kapitel 3.1.2. Um das lineare Gleichungssystem aufzustellen, muß man hier natürlich drei Versuche mit verschiedenen Parametern machen.

$$
\begin{array}{rcl}
c1 \cdot \left(B_1 + 2\sqrt{B_3} \left(1 - \dfrac{\ln B_1}{\ln B_3} \right) \right) + c_2 \cdot \sqrt{B_3} + c_3 \cdot \left(\dfrac{B_3}{\ln B_3} - \dfrac{B_1}{\ln B_1} \right) & = & t \\
c1 \cdot \left(B_1' + 2\sqrt{B_3'} \left(1 - \dfrac{\ln B_1'}{\ln B_3'} \right) \right) + c_2 \cdot \sqrt{B_3'} + c_3 \cdot \left(\dfrac{B_3'}{\ln B_3'} - \dfrac{B_1'}{\ln B_1'} \right) & = & t' \\
c1 \cdot \left(B_1'' + 2\sqrt{B_3''} \left(1 - \dfrac{\ln B_1''}{\ln B_3''} \right) \right) + c_2 \cdot \sqrt{B_3''} + c_3 \cdot \left(\dfrac{B_3''}{\ln B_3''} - \dfrac{B_1''}{\ln B_1''} \right) & = & t''
\end{array}
$$

Optimale Parameter für ECM mit verbesserter Standard-Continuation zeigt Tabelle 3.1.

Tabelle3.1: *optimale Parameter für ECM mit verbesserter Std.Continuation*

#Stellen	B_1	B_3	#Kurven
10	411	16 185	8
11	627	28 027	10
12	928	45 579	14
13	1 362	72 108	18
14	1 992	120 452	23
15	2 910	184 110	29
16	4 175	269 720	38
17	5 889	414 299	48
18	8 177	617 457	61
19	11 424	840 307	78
20	15 752	1 219 430	98
21	21 451	1 726 998	123
22	29 505	2 465 816	153
23	40 155	3 445 936	190
24	55 323	4 869 607	231
25	73 836	6 540 019	288
26	99 906	8 900 812	352
27	134 059	11 903 355	431
28	178 459	15 653 379	530
29	230 251	22 193 058	648
30	309 156	29 284 729	779
31	412 370	38 074 821	939
32	520 635	52 157 849	1154
33	702 456	68 307 733	1360
34	897 154	94 807 273	1634
35	1 167 479	117 338 713	1982
36	1 549 219	148 234 399	2346
37	1 944 642	200 109 835	2830
38	2 620 165	255 864 848	3269
39	3 254 381	339 886 233	3952
40	4 021 442	448 476 926	4784

3.3 Die Geburtstags-Paradoxon-Continuation

Diese Continuation beruht auf einer ganz anderen Faktorisierungsmethode und ist auch unter den Bezeichnungen "Pollards ρ-Methode" oder "Monte Carlo Methode" bekannt. Betrachten wir zunächst Pollards ursprünglichen Algorithmus und übertragen ihn dann auf ECM.

3.3.1 Pollards ρ-Methode

Diese Methode beruht auf der Tatsache, daß sich eine Folge von zufälligen ganzen Zahlen modulo p gewöhnlich nach nur $c\sqrt{p}$ ($c \in \mathbb{R}^+,\ p \in \mathbb{P}$) Schritten wiederholt. Das kann man leicht einsehen, wenn man die Lösung des sogenannten "Geburtstags-Paradoxon"-Problems betrachtet. Hier möchte man wissen, wieviele Personen man zufällig auswählen muß, damit die Wahrscheinlichkeit, daß mindestens zwei von ihnen am gleichen Tag Geburtstag haben, $1/2$ übersteigt.

Lösung:

Die Wahrscheinlichkeit, daß q Personen an verschiedenen Tagen Geburtstag haben ist:

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

Dieser Ausdruck ist kleiner als $1/2$, wenn q größer als 23 ist.
Allgemeiner:
Wieviele zufällige ganze Zahlen muß man wählen, bis mindestens zwei von ihnen mit einer Wahrscheinlichkeit größer als $1/2$ kongruent modulo p sind?
Das ist der Fall, wenn

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

$$\Rightarrow q \approx \sqrt{2\ln 2}\cdot\sqrt{p} \ \approx\ 1.18\sqrt{p} \quad \text{(s. [Riesel] S.179)}$$

Im Prinzip funktioniert Pollards ρ-Methode folgendermaßen:

1. Generiere eine periodische Folge x_i von zufälligen ganzen Zahlen modulo p.
2. Suche die Periode, d.h. finde i und j, so daß $x_i \equiv x_j \bmod p$.
3. Berechne $d = \mathrm{ggT}(x_i - x_j, N)$;
   denn es gilt:

$$\begin{aligned} & x_i \equiv x_j \bmod p \\ \Rightarrow\quad & p \mid x_i - x_j \end{aligned}$$

zu 1) Eine mögliche Folge ist:

$$x_{i+1} \equiv x_i^2 + a \bmod p \tag{3.6}$$

Praktische Beobachtungen zeigten, daß die Folgeglieder zufällig genug für den Algorithmus sind (wenn a weder 0 noch -2 ist); allerdings konnte das bisher noch nicht bewiesen werden.
Weil p zu diesem Zeitpunkt noch unbekannt ist, erzeugen wir die Folge modulo N. Da p ein Teiler von N ist, wird dadurch die Folge modulo p induziert.

zu 2) Sei x_i eine Folge, bei der x_i nur durch x_{i-1} definiert wird (z.B.: Folge in (3.6)). Diese Folge wird periodisch wiederholt, sobald ein Glied x_j mit einem der vorhergehenden Gliedern identisch ist. Um die Periode zu finden, müßte man also jedes neue Folgenglied mit allen alten vergleichen. Das ist indes nicht möglich, wenn die Periode sehr lang ist. In diesem Fall kann Floyds Zyklus-Finde-Algorithmus angewendet werden. Er beruht auf der Erkenntnis, daß die Periode einer Folge mit Sicherheit durch wiederholtes Testen von

$$x_{2i} \equiv x_i \bmod N$$

entdeckt wird (Beweis s. [Riesel]).
Wie kann man nun die x_{2i} mit den x_i vergleichen, ohne alle x_i zu speichern? Man berechnet einfach die x_i parallel zu den x_{2i}. Sei nun $x_{i+1} = f(x_i)$, dann kann man Floyds Zyklus-Finde-Algorithmus wie folgt programmieren:

Algorithmus 3.4 Floyds Zyklus-Finde-Algorithmus

$x \leftarrow x_1$

$y \leftarrow f(x_1)$

<u>while</u> $(x \neq y)$

  $x \leftarrow f(x)$

  $y \leftarrow f(y);\ \ y \leftarrow f(y)$

<u>endwhile</u>

Das alles zusammengesetzt ergibt den folgenden Algorithmus:

Algorithmus 3.5 Pollards ρ-Algorithmus

<u>Eingabe:</u> N
<u>gesucht:</u> p

```
(1)   wähle a      /* siehe Gleichung 3.6 */
      wähle r      /* max.  Anzahl Versuche den Zyklus zu finden */
      wähle x0     /* Startwert für Iteration */
      x ← x0;  y ← x0

(2)   while (r > 0)
            x ← x^2 + a mod N
            y ← y^2 + a mod N
            y ← y^2 + a mod N
            if ((d ← ggT(x − y, N)) > 1) then goto (3)
            r ← r − 1
      endwhile
      FERTIG, aber keinen Faktor gefunden

(3)   if (d = N) then goto (1) und wähle ein anderes a
      else FERTIG, Faktor gefunden
```

Laufzeit:
In der Schleife von Schritt (2) werden 3 Multiplikationen modulo und ein ggT berechnet. Mit dem gleichen Trick wie in Kapitel 3.2.2 kann man die Kosten des ggT's auf eine Multiplikation modulo drücken. Somit kostet ein Schleifendurchlauf vier Multiplikationen modulo. Da die Schleife im Schnitt $c\sqrt{p}$-mal durchlaufen wird, ergibt sich für die Gesamtlaufzeit der ρ-Methode $O(\sqrt{p})$ Multiplikationen modulo.

3.3.2 Die Continuation

Übertragen wir nun Pollards ρ-Methode auf ECM. Sei dabei wie vorher $Q(= kP)$, das von ECM gelieferte Ergebnis. Welche Ordnung hat Q?

Sei wieder $\mathrm{ord}_{E(\mathbb{Z}/N\mathbb{Z})}(P) = n_1 \cdots n_k,\ n_i \in \mathbb{P},\ i = 1, \ldots, k,\ k \geq 2,\ n_1 > n_2 \geq n_3 \geq \ldots,\ n_1 > B_1,\ n_i \leq B_1$ für $i = 2, \ldots, k$.
D.h.: $n_2 \cdots n_k \mid k,\ n_1 \nmid k$. Dann gilt:

$$\begin{aligned} & n_1 \cdots n_k \cdot P = \mathcal{O} \\ \Rightarrow\ & n_1 \cdot kP = \mathcal{O} \\ \Rightarrow\ & n_1 \cdot Q = \mathcal{O} \\ \Rightarrow\ & \mathrm{ord}(Q) = n_1,\ \text{da } n_1 \in \mathbb{P} \text{ und somit kein Vielfaches von } \mathrm{ord}(Q) \text{ ist.} \end{aligned}$$

Die Geburtstag-Paradoxon-Continuation funktioniert jetzt im Prinzip wie die Pollard ρ-Methode (vgl. S.47):

1. Generiere eine periodische Folge Q_i von zufälligen Punkten.
2. Suche die Periode, d.h. finde i und j so, daß $Q_i = Q_j$ in $E(\mathbb{Z}/p\mathbb{Z})$. Dazu braucht man im Mittel $O(\sqrt{n_1})$ Schritte.
3. Berechne $d = \mathrm{ggT}(x_i - x_j, N)$ mit $Q_i = (x_i, y_i)$ und $Q_j = (x_j, y_j)$.

zu 1) Dazu benötigen wir eine geeignete Zufallsfunktion:

$$\begin{aligned} Q_1 &:= Q \\ Q_j &:= \begin{cases} 2 \cdot Q_{j-1}, & \text{mit Wahrscheinlichkeit } 1/2 \\ Q + 2 \cdot Q_{j-1}, & \text{mit Wahrscheinlichkeit } 1/2 \end{cases} \end{aligned}$$

$Q_1, \ldots, Q_r$ sind somit zufällig gewählte Punkte und können mit $O(r)$ Punktadditionen berechnet werden.

zu 2) & 3) Leider läßt sich Floyds Zyklus-Finde-Algorithmus nicht anwenden, da aus $Q_i = Q_j$ nicht folgt, daß $Q_{i+1} = Q_{j+1}$ ist. Stattdessen sei

$$\begin{aligned} Q_j &:= (x_j, y_j) \quad \text{und} \\ d &:= \prod_{i=1}^{r-1} \prod_{j=i+1}^{r} (x_i - x_j) \bmod N. \end{aligned}$$

Ist nun für $i < j \leq r \quad Q_i = Q_j$ in $E(\mathbb{Z}/p\mathbb{Z})$, dann gilt:

$$\begin{aligned} x_i &\equiv x_j \bmod p \\ &\Rightarrow p \mid x_i - x_j \\ &\Rightarrow p \mid d \\ &\Rightarrow \mathrm{ggT}(d, N) \geq p. \end{aligned}$$

Laufzeit und optimale Parameter findet man in [Brent]. In der Praxis ist dieser Algorithmus etwas schneller als die Standard-Continuation aber deutlich langsamer als die verbesserte Standard-Continuation; deshalb habe ich ihn nicht programmiert. Allerdings gibt [Brent] noch einige Verbesserungen an, mit der diese Continuation ähnlich schnell wie die verbesserte Standard-Continuation werden sollte.

Kapitel 4

Die Programme

Wenden wir uns nun dem praktischen Teil zu.

ECM ist zur verteilten Anwendung auf mehreren Rechnern gut geeignet. Nehmen wir an, daß wir mit gewissen Parametern B_1, B_2 und B_3 im Mittel 1000 Kurven testen müßten, um einen Faktor zu finden. Diese Kurven kann man natürlich auf einem einzelnen Rechner hintereinander testen; man könnte aber auch 1000 Rechner jeweils eine Kurve bearbeiten lassen, da die Kurven zufällig gewählt werden. Verfügt man über 50 Rechner, so erhält jeder Rechner nur 20 Kurven.
Deshalb habe ich ein Programm für einen einzelnen UNIX-Rechner und ein Programmpaket für ein UNIX-Rechnernetz (bestehend aus mindestens 2 Maschinen) erstellt. Beide Programme benutzen die gleichen mathematischen Funktionen; lediglich die Hauptprogramme unterscheiden sich.
Die Programme haben folgende Merkmale:

- ECM mit ggT-Trick (s. Kapitel 2.2.3, S.35),
- Kurvengenerator mit Gruppenordnung durch 4 teilbar (s. Kapitel 2.2.3, S.37),
- verbesserte Standard-Continuation (S. Kapitel 3.2, S.43).

Zusätzlich habe ich auch die normale Standard-Continuation implementiert. Soll diese benutzt werden, da z.B. nicht genügend Speicherplatz für die Tafel der verbesserten Standard-Continuation vorhanden ist, so muß der entsprechende Funktionsaufruf in den Hauptprogrammen ersetzt werden.

4.1 Installation & Bedienung

4.1.1 Die Installation

Folgende Schritte sind zur Installation der beiden Programme notwendig:

1. Speichern der Include-Files *allg.h, ecmp-types.h, larith+.h* in einem Verzeichnis.
   In dem Include-File "allg.h" können folgende Programmparameter eingestellt werden:

   - STOP_ECM "string" (nur für verteiltes ECM)
     Erzeugt man eine Datei mit Namen "string" in dem Verzeichnis, aus dem das verteilte ECM gestartet wurde, so wird ECM kontrolliert abgebrochen, d.h. auf allen Maschinen wird ECM beendet.
   - STRATEGYFILES "string1 string2 string3 ..."
     Die Dateien enthalten die Strategie, mit der ECM nach Faktoren sucht. Das Programm versucht zuerst die Datei "string1" zu lesen; funktioniert das nicht, so versucht es Datei "string2" zu lesen u.s.w., bis eine der angegebenen Dateien gelesen werden kann. Setzt man

     STRATEGYFILES "ecm-strategy /home/verwalter/universal-strategy"

     so kann jeder Benutzer seine eigene Strategie in der Datei "ecm-strategy" in dem Verzeichnis, aus dem er ECM aufruft, abspeichern. Fehlt diese Datei, so wird die Strategie aus der Datei "/home/verwalter/universal-strategy" benutzt.
   - PARAMETERFILE "string1 string2 string3 ..."
     Hier gibt man den Namen für die Datei, die die Parametertabelle enthält, an. Ansonsten gilt das für STRATEGYFILES Gesagte.
   - PRIMTAFEL "string1 string2 ..."
     Wie vorher, nur gibt man hier den Namen der Primzahltafel mit komprimierten Primzahlen an. Ein Programm zur Erzeugung einer Primzahltafel steht in Anhang E.1.3 (S.107).
   - HOSTNAMELEN zahl (nur verteiltes ECM)
     Länge des längsten Rechnernames, der benutzt wird. Dazu muß man die CLIENT_CONF-Datei konsultieren.

   Alle anderen Konstanten sollten nicht verändert werden.

2. Erzeugen der Librarys *larith+.a, elliptic.a.*
   Dazu existieren zwei Makefiles (s. Anhang E.2.1, S.109 und S.131). Weiterhin braucht man die Library larith.a (s. [larith]) und zum verteilten Rechnen die Library liblips.a (s. [LIPS]).

3. Erzeugen der Hauptprogramme *ecmp, ecmp-master, ecmp-worker*.

   - ECM für einen Rechner
     Hierzu benötigt man das Makefile "ecmp.make (s. Anhang E.1.1, S.91) und das Programm ecmp.c. Man compiliere.
   - verteiltes ECM
     Hier muß man zwei Programme erzeugen. Das Steuerprogramm ecmp-master erhält man, wenn man den Quellcode ecmp-master.c (s. S. 98) mit Hilfe des Makefiles master.make (s. S. 97) compiliert; das ECM-Programm erhält man, wenn man den Quellcode ecmp-worker.c (s. S. 105) mit Hilfe des Makefiles worker.make (s. S. 104) compiliert.

Damit ist die Installation abgeschlossen.

4.1.2 Die Bedienung

Die Geschwindigkeit von ECM hängt stark von der optimalen Wahl der Parameter ab. Wie wählt man nun optimale Parameter, wenn die Größe der Faktoren unbekannt ist?
Eine mögliche Strategie ist, zunächst mit niedrigen Parametern zu beginnen und diese dann schrittweise zu erhöhen. Meiner Erfahrung nach sollte man eine große Schrittweite wählen, z.B.:

- suche 10-stellige Faktoren
- suche 15-stellige Faktoren
- suche 20-stellige Faktoren

  ⋮

Diese Strategie wird in der Datei "ecm-strategy" abgelegt (siehe auch Installation S. 52 STRATEGYFILES). Die Einträge im Strategiefile haben folgendes Format:

```
#Stellen Wahrscheinlichkeit
#Stellen Wahrscheinlichkeit
#Stellen Wahrscheinlichkeit
⋮
0 0      /* Markiert das Ende der Strategie */
```

Beispiel: Strategie-File

```
10 0.5
```

15 0.5

20 0.5

0 0

Die Zeile

10 0.5

bedeutet, daß ECM optimale Parameter B_1, B_2, B_3 bestimmt und die Anzahl der Kurven berechnet, die im Mittel benötigt werden um einen 10-stelligen Faktor mit Wahrscheinlichkeit 0.5 zu finden. Andere Werte für die Wahrscheinlichkeit sind allerdings in der jetzigen Programmversion sinnlos! Hat ECM die Strategie abgearbeitet ohne einen Faktor zu finden, so bricht es ergebnislos ab.
Außerdem benötigen die Programme noch folgende Eingaben:

- ECM für einen Rechner

  1. zu faktorisierende Zahl
  2. Limit für Probedivision
     Die kleinen Faktoren einer Zahl findet man am schnellsten durch Probedivision. Hierbei versucht man die zu faktorisierende Zahl durch alle Primzahlen unterhalb einer gewissen Grenze zu dividieren. Ist der Divisionsrest gleich 0, so hat man einen Teiler gefunden. 1 000 000 ist hierfür ein vernüftiger Wert.
  3. Restlänge
     Das Programm bricht ab, wenn es die zu faktorisierende Zahl auf Restlänge Dezimalstellen "gedrückt" hat. Das ist sinnvoll, wenn man noch andere Verfahren (z.B. das quadratische Sieb) auf die Restzahl anwenden möchte. Um die Zahl ganz zu faktorisieren, muß man hier 1 eingeben.
  4. Info
     Gibt man hier eine 1 ein, so werden ausführliche Informationen über den Programmablauf ausgegeben (u.a. auch der Zeitverbrauch). Wenn man eine 0 eingibt, so erscheinen als Ausgabe nur die Faktoren und die Restzahl.

  Praktischerweise schreibt man die Eingaben in eine Datei z.B. "eingabe" und startet ECM dann mit:

  ecmp < eingabe

  So muß in den meisten Fällen lediglich die zu faktorisierende Zahl verändert werden, da die anderen Eingaben normalerweise gleich bleiben.

- verteiltes ECM
  Hier wird zusätzlich die Datei mit Namen “CLIENT_CONF” benötigt, in der die zu benutzenden Rechner eingetragen werden müssen (s. Anhang E.4.1 S.158).

  1. zu faktorisierende Zahl
  2. Limit für Probedivision
  3. Restlänge
  4. Info
     Funktioniert hier nicht; hier muß man eine 1 eingeben.
  5. Protokoll
     Gibt man hier 1 ein, so führen die Worker über den Programmablauf Protokoll. Die Ausgaben werden in die Datei *LOGFILE.rechnername* geschrieben, wobei *rechnername* durch die UNIX-C-Funktion *get_hostname()* ermittelt wird. Gibt man eine 0 ein, so werden diese Dateien nicht angelegt.

4.2 Die Datenstrukturen

Wir rechnen mit Punkten auf elliptischen Kurven. Um den ggT-Trick (s. Kapitel 2.2.3, S. 35) anzuwenden, muß man mehrere Kurven gleichzeitig betrachten. Zur einfachen Handhabung der Routinen habe ich einige neue Typen definiert, die fast alle Zeiger auf Datenstrukturen sind. Daher wird der Speicherplatz nicht automatisch verwaltet, sondern muß durch spezielle Funktionen reserviert und wieder freigegben werden.

Beispiel:

Curve C;

definiert einen Zeiger auf eine Kurve;

C = ecalloc(10);

reserviert den Speicher für eine Kurve und setzt den Zeiger C darauf.

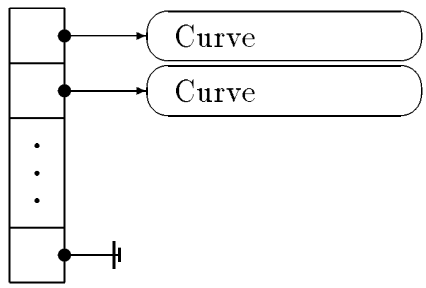
ecfree(C);

gibt den Speicher für die Kurve, auf die C zeigt, wieder frei.

Der Einfachheit halber werde ich nun keine Unterscheidung mehr zwischen dem Zeiger und der zugehörigen Datenstruktur machen, da dies leicht aus dem Kontext hervorgeht. Eine lange ganze Zahl wird in dem Datentyp LONGTYPE, den die [larith] zur Verfügung stellt, abgespeichert. Soweit nicht anders vermerkt, findet man die Typdefinitionen in dem Include-File “ecmp-types.h”und die Routinen zur Speicherverwaltung in der Library “elliptic.a”.

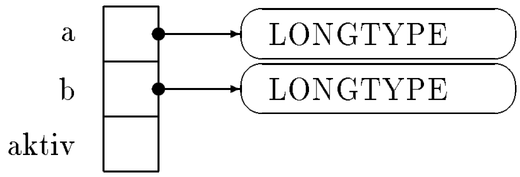
Hier nun die wichtigsten Datentypen:

Vector_of_curves:



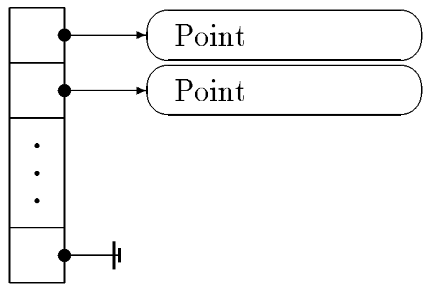
Vector_of_Curves ist ein Feld von Zeigern auf die Datenstruktur *Curve*. Das letzte Element ist der NULL-Zeiger.
Speicherverwaltung: ecvalloc(), ecvfree().

Curve:



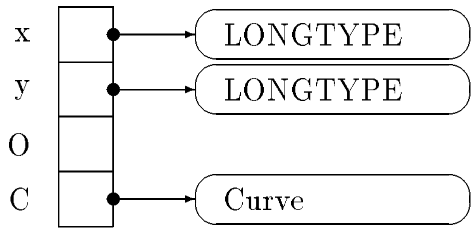
Curve ist ein struct, bestehend aus den Komponenten a und b vom Typ LONGTYPE und dem Boolean *aktiv*. a und b sind die Parameter der Kurvengleichung (s. Kapitel 2.1, S. 21). Den Parameter b benötigt man zur Punktaddition nicht! Zum Debugging ist er aber sehr nützlich, denn man benötigt ihn zum Testen, ob ein Punkt auf der zugehörigen Kurve liegt. Das übernimmt die Funktion *eponcurve()*.
Speicherverwaltung: ecalloc(), ecfree().

Vector_of_points:



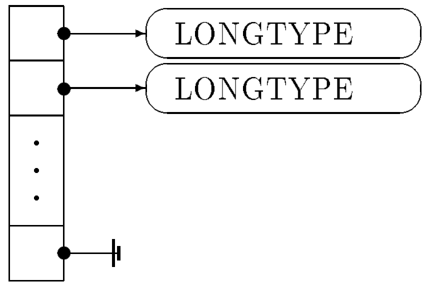
Analog zu *Vector_of_curves*.
Speicherverwaltung: epvalloc(), epvfree().

Point:



Ein *Point* ist ein struct, bestehend aus den beiden Koordinaten x, y vom Typ LONGTYPE, einem *Boolean* O und einem Zeiger auf die Kurve, auf der er liegt. Ist $O = 1$, dann ist der Punkt gleich dem Nullpunkt und x, y werden ignoriert.
Speicherverwaltung: epalloc(), epfree().
Achtung:
Mit *epvalloc()* wird nur der Speicher für den struct und die beiden LONGTYPES reserviert; der Speicher für die Kurve muß mit *ecalloc()* bzw. mit *ecvalloc()* reserviert werden. Das ist sinnvoll, da mehrere Punkte auf einer Kurve liegen können.

Vector_of_LONGTYPE:



Analog zu *Vector_of_curves*. Die Typdefinitionen findet man in *larith.h*.
Speicherverwaltung: lvalloc(), lvfree() (in *larith.a*).

4.3 Die wichtigsten Funktionen

Im Prinzip faktorisiert man mit ECM folgendermaßen:

Eingabe: N (zu faktorisierende Zahl)

(1) Probedivision
/* Hier könnten noch andere Faktorisierungsverfahren wie z.B. Pollards p-1- und ρ-Methode oder squfof eingesetzt werden. */

wiederhole bis Restzahl $\in \mathbb{P}$

(2) Kurvengenerator

(3) ECM

(4) Continuation

Ausgabe: Faktoren von N

Die Probedivision übernimmt die Funktion

$ltrialdiv(N)$

(s. Anhang E.2.2, S. 147), die in der Library *larith+.a* enthalten ist.
Nun müssen wir einige Kurven mit Punkten darauf erzeugen (Schritt (2)). Dazu gibt es die Funktion

$ecvgen(C, P, N, Faktor)$.

C ist ein Kurvenvektor, P ein Punktevektor und N die zu faktorisierende Zahl. *ecvgen()* generiert nun — mit dem in Kapitel 2.2.3 beschriebenen Verfahren — Kurven und Punkte und trägt diese in die entsprechenden Vektoren ein, wobei auch die Kurvenzeiger der Punkte auf die entsprechenden Kurven gesetzt werden. In fast allen anderen Funktionen müssen nur noch die Punktevektoren übergeben werden, da man jetzt weiß, auf welcher Kurve der Punkt liegt.
Während der Kurven- und Punktegenerierung kann schon ein Faktor von N gefunden werden, da bei diesen Berechnungen modulo N invertiert wird. Ist das der Fall, so liefert *ecvgen()* als Ergebnis eine 1 und der Faktor wird in *Faktor* zurückgegeben. Andernfalls ist das Ergebnis 0 und *Faktor* ist undefiniert.

Der eigentliche ECM-Algorithmus verbirgt sich hinter der Funktion

$ecvfindfactor(B_1,\ B_2,\ N,\ P,\ pp,\ Faktor)$.

B_1 und B_2 sind die Parameter für ECM, N die zu faktorisierende Zahl, P der von ECM generierte Punktevektor und *pp* ein Zeiger auf eine Primzahltafel (**p**ointer on **p**rimes). *ecvfindfactor* ist eine Funktion des Typs *Status* (in *ecmp-types.h* definiert) und kann somit die folgenden Werte annehmen:

- *factor_found*, wenn ein Faktor von N gefunden wurde.
  Der Faktor wird dann in *Faktor* zurückgegeben.
- *no_factor_found*, wenn kein Faktor gefunden wurde.
  $Faktor$ ist nun undefiniert, aber es wurde $k \cdot P$ berechnet (Ergebnis in P).
- *hopeless*, sonst.
  War das Ergebnis eines erweiterten ggTs bei der Inversenberechnung während einer Punktaddition gleich N, so hat man weder einen Faktor von N gefunden, noch das Inverse berechnet. Man kann also mit dieser Kurve nicht mehr weiterrechnen; sie ist nutzlos und wird kurzerhand deaktiviert (d.h., das *aktiv*-Flag im Struct *Curve* wird auf 0 gesetzt). Sind danach weniger als MINCDIM (s. Includefile *allg.h* S. 155) Kurven aktiv, so bricht *ecvfindfactor()* mit Status *hopeless* ab. In diesem Fall muß man neue Kurven generieren. Allerdings konnte ich diesen Fall noch nie beobachten.

Liefert ECM den Status *no_factor_found*, so kann man eine Continuation anschließen. Die verbesserte Standard-Continuation (s. Kapitel 3.2, S. 43) habe ich in der Funktion

$$ecm\_stdcont2(P,\ B_1,\ B_3,\ N,\ pp,\ Faktor)$$

realisiert. P ist der Punktevektor mit den Ergebnissen aus dem vorherigen ECM-Lauf, B_1 und B_3 Parameter für die Continuation, N die zu faktorisierende Zahl und *pp* der Zeiger auf die Primzahltafel. Mit der Variablen *SPmax* (im Quelltext) kann man den maximalen Speicherplatzverbrauch festlegen.
Ebenso wie *ecvfindfactor()* ist *ecm_stdcont2()* vom Typ Status und kann somit folgende Werte annehmen:

- *factor_found*, wenn ein Faktor gefunden wurde,
- *no_factor_found*, wenn kein Faktor gefunden wurde,
- *hopeless*, wenn zuviele Kurven deaktiviert wurden.

Möchte man statt der verbesserten Standard-Continuation die normale Standard-Continuation einsetzen, so kann man die Funktion

$$ecm\_stdcont(P, B_3, N, pp, Faktor)$$

verwenden.
Achtung: Im Gegensatz zu *ecm_stdcont2()* ist der Parameter B_1 nicht nötig.
Ansonsten verhält sich diese Funktion (bis auf die Geschwindigkeit) wie *ecm_std-cont2()*.

Das Programm *ecmp.c* (s. S. 93) ist ein gutes Beispiel für die praktische Anwendung dieser Funktionen. Zusätzlich verrichten noch viele andere Funktionen in den Programmen ihren Dienst. Der interessierte Leser findet sie im Anhang.

4.3.1 Die verteilte Implementierung

Zur Verteilung von ECM auf mehrere Rechner wurde mit Hilfe von [LIPS] ein Master-Worker-Prinzip realisiert. Hierbei übernimmt der sogenannte Master-Rechner Verwaltung und Steuerungsaufgaben — insbesondere läuft kein ECM auf ihm — während die Worker-Rechner die eigentliche Aufgabe erledigen. Dabei kommunizieren die Worker nur mit dem Master und nicht untereinander (s. Abbildung 4.1).

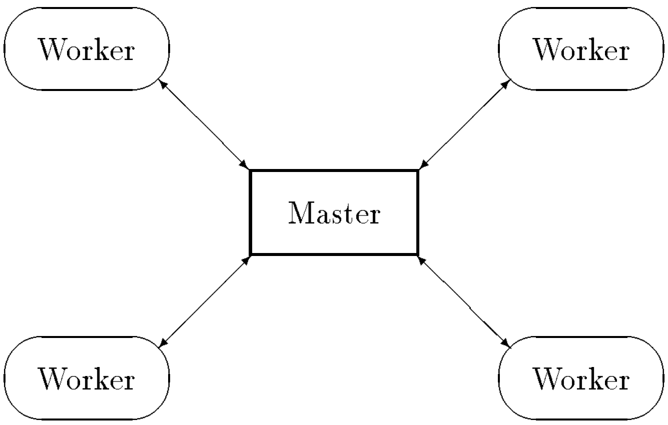


Abbildung4.1: Master-Worker-Konfiguration

Betrachten wir zunächst das Master-Programm *ecmp-master.c*. Es hat folgende Struktur:

Der Master:

```
Initialisierung der Netzwerkfunktionen

Einlesen der Eingabe

Probedivision
```

```
     Einlesen der Strategie

(M1) Aufteilen der Arbeit in einzelne Jobs für je einen Rechner.

(M2) Starten der Worker.  (Diese warten jetzt auf ihren Job.)

     Solange der Rest der zu faktorisierenden Zahl keine Primzahl ist
     und noch nicht alle Jobs bearbeitet wurden

    (M3) Verschicke einen Job an jeden Worker.
         Wiederhole
       (M4) Warte auf die Ergebnisse der Worker.
       (M5) Überprüfung des Ergebnisses:
              · wenn das Ergebnis ein echter Teiler der Zahl ist
              - dann dividiere die Zahl durch das Ergebnis,
                gib Faktor aus,
              · wenn das Ergebnis 1 ist
              - schicke neuen Job an den Worker,
                der sich zurückgemeldet hat,
         solange kein Faktor gefunden wurde und Jobs übrig sind

     Ende der Schleife

(M6) Terminiere alle Worker.

     Programmende
```

Daneben sieht das Worker-Programm vergleichsweise einfach aus.

Der Worker:

```
     Initialisierung der Netzwerkfunktionen

     wiederhole

    (W1) Warte auf Jobs.
    (W2) ECM mit Continuation
    (W3) Schicke Ergebnis an Master.

     bis ans Ende aller Tage
```

Wie arbeiten die beiden Programme zusammen?

Sei N die zu faktorisierende Zahl. Der Master sendet in Schritt (M3) jedem Worker einen Job, bestehend aus N, B_1, B_3 und der Anzahl Kurven, die vom Worker simultan bearbeitet werden ($B_2 = B_3$ s. 3.1.2, S. 41). Dann wartet der Master auf

die Ergebnisse der Worker (M4). Die Worker lesen den Job ein (W1), rufen ECM und die Continuation auf (W2), schicken das Ergebnis an den Master (W3) und warten auf ihren nächsten Job (W1).
Der Master liest das Ergebnis eines Workers ein (M4). Jetzt können folgende Fälle auftreten:

1. Das Ergebnis ist 1. Der Worker hat nichts gefunden.
2. Das Ergebnis ist ein echter Teiler von N.
3. Das Ergebnis ist kein echter Teiler von N.

In jedem Fall ist das Ergebnis ungleich N, da die entsprechende Kurve deaktiviert worden wäre.

Fall (1) tritt leider am häufigsten auf. Hier schicken wir dem Worker einfach einen neuen Job.

Im 2. Fall haben wir einen Faktor n_1 von N gefunden. N wird sooft wie möglich durch n_1 geteilt und n_1 dabei ausgegeben. Also:

> `while` (N `mod` n_1 `== 0)`
>
> $N = N/n_1$
>
> `print` n_1
>
> `endwhile`

Ist N danach eine Primzahl, so wurde die Zahl komplett faktorisiert und wir sind FERTIG, andernfalls erhalten alle Worker einen neuen Job.
Achtung: Der Master verschickt hier an <u>alle</u> Worker einen neuen Job. Den Grund hierfür erörtere ich etwas später.

Fall (3) wirkt zunächst etwas befremdend. Hat der Worker etwa falsch gerechnet? Dieser Fall tritt auf, wenn zwei (oder mehrere) Worker kurz hintereinander den gleichen Faktor n_1 zum Master schicken. Durch den ersten Faktor n_1 landet der Master in Fall (2). Danach ist N nicht mehr durch n_1 teilbar. Somit sind die nächsten n_1, die durch den Master eingelesen werden, keine Teiler mehr von N.
Hier muß kein Job verschickt werden, da der Master — kurz vorher im Fall (2) — allen Workern einen neuen Job geschickt hat.

Problem:
Sendet ein Worker einen Faktor von N zum Master, so ist ab diesem Zeitpunkt die Arbeit auf allen Workern nutzlos, da diese höchstwahrscheinlich den gleichen Faktor finden (ECM findet kleine Faktoren viel schneller als größere).
Diese beiden Probleme könnten vernachlässigt werden, wenn ECM nur kurze Zeit

zur Bearbeitung eines Jobs bräuchte. Das ist aber insbesondere dann nicht der Fall, wenn man nach großen Faktoren sucht. Deshalb sendet der Master jedem Worker einen neuen Job.
Damit der Worker merkt, daß ein neuer Job angekommen ist, muß er von Zeit zu Zeit nachschauen. Das geschieht in den Funtionen *ecvfindfactor()* und *ecm_stdcont2()* mit Hilfe von *new_job()*, falls diese mit *-DPARALLEL* übersetzt werden (verwendet man die Makefiles *master.make* und *worker.make*, so ist diese Option automatisch aktiviert).

Die Kommunikation des Master mit den Workern wird mit Funktionen aus [LIPS] bewerkstelligt. [LIPS] steuert die Worker so, daß sie nur die Idle-Time zum Rechnen nutzen. Wird auf einem Rechner gearbeitet, so werden die Prozesse so lange angehalten. In dieser Zeit können mehrere Jobs eintreffen. [LIPS] legt die angekommenen Nachrichten auf einen Stack. Somit erhält der Worker beim nächsten Einlesen den neusten Job zuerst; ältere Jobs werden später gelesen.
Wie unterscheidet man nun zwischen einem älteren und einem gerade angekommenen Job?
Der Job wird nur akzeptiert, wenn die neue zu faktorisierende Zahl kleiner als die aktuelle ist.

Es tritt noch ein weiteres Problem auf!

Da die Worker-Prozesse beliebig lange aufgehalten werden können, werden manche Jobs möglicherweise nie beendet (weitere Gründe dafür sind z.B.: Abschalten oder Reboot eines Rechners). Damit der Master nicht ewig auf solch einen Job warten muß, werden alle Jobs in einem Ring angeordnet und der Reihe nach verteilt. Meldet sich ein Worker zurück, so wird der entsprechende Job aus dem Ring entfernt. Der Master verteilt nun solange Jobs, bis der Ring leer ist oder die Zahl faktorisiert wurde.

Betrachten wir nun noch die Art und Weise, mit der der Master, ausgehend von der angegebenen Strategie, die einzelnen Jobs einteilt.
Dafür ist die Funktion *job_planing()* zuständig.
Angenommen, wir müßten folgende Strategie bearbeiten:

Strategie		Parameter aus Tabelle		
		B_1	B_3	#Kurven
10	0.5	411	16185	8
20	0.5	15752	1219430	98
30	0.5	309156	29284729	779
0	0			

Das bedeutet: Wir suchen zunächst nach 10-stelligenFaktoren, dann nach 20 stelligen und schließlich nach 30 stelligen. *job_planing()* liest nun mit Hilfe der Funktion *read_parameters* (s. S. 127) optimale Werte für den ersten Strategie-Eintrag (hier

10 Stellen) ein ($B_1 = 411$, $B_3 = 16185$, #Kurven = 8).
Wie könnte man die Arbeit auf verschiedene Rechner verteilen?
Eine naive Idee wäre, 8 Rechner mit jeweils 1 Kurve zu beschäftigen. Leider wird dann der ggT-Trick (s. S. 35) nicht genutzt. Um das zu verhindern, existieren die Konstanten *CS_MIN* (minimale Anzahl simultaner Kurven) und *CS_MAX* (s. Include-File *allg.h* S. 155). Mit *CS_MIN* legt man die minimale Anzahl der Kurven fest, die simultan auf einem Rechner getestet werden, mit *CS_MAX* die maximale Anzahl simultaner Kurven.
Mit folgendem Verfahren wird ein Strategie-Eintrag in Jobs eingeteilt:

```
kpr = round(#Kurven / #Rechner); /* Kurven pro Rechner */
if (kpr < CS_MIN) then kpr = CS_MIN;
if (kpr > CS_MAX) then kpr = CS_MAX;
#Jobs = round(#Kurven / kpr);
if (#Jobs < 1) then #Jobs = 1;
```

Beispiel:
CS_MIN = 10, CS_MAX = 15, #Rechner = 10;

1. Strategie-Eintrag:
   #Kurven = 8 $\Rightarrow$ #Jobs = 1,
   Der erste Strategie-Eintrag wird also auf einem einzigen Rechner bearbeitet.
2. Strategie-Eintrag:
   #Kurven = 98
   $\Rightarrow$ kpr = round(98/10) = 10
   $\Rightarrow$ #Jobs = round(98/10) = 10
   Hier werden also 10 Jobs mit jeweils 10 Kurven erzeugt.
3. Strategie-Eintrag:
   #Kurven = 779
   $\Rightarrow$ kpr = round(779/10) = 78
   $\Rightarrow$ kpr = 15, da kpr > CS_MAX
   $\Rightarrow$ #Jobs = round(779/15) = 52
   Hier werden 52 Jobs mit jeweils 15 Kurven erzeugt.

In dem Beispiel werden einige Kurven zuviel getestet. Da die Parameterberechnung aber lediglich approximativ ist, kann diese Ungenauigkeit vernachlässigt werden.
Alle Jobs werden nun in einen *Job_Buffer* geschrieben (Datenstruktur: Ring) und vom Master nach und nach an die Rechner verteilt.

Kapitel 5

Laufzeittests

5.1 Der ggT-Trick

Der ggT-Trick (s. S. 35) beschleunigt die Punktaddition. Er wirkt sich somit auf den ECM-Grundalgorithmus, die Standard-Continuation und die Tabellenberechnung in der verbesserten Standard-Continuation aus.
Um den Geschwindigkeitsgewinn zu testen wurde eine 100-stellige Primzahl erzeugt und die Zeitdauer eines erfolglosen Versuchs des ECM-Grundalgorithmus mit ggT-Trick auf unterschiedlich vielen Kurven simultan gemessen.

Tabelle5.1: *Beschleunigung durch ggT-Trick*

#Kurven simultan	#Mul.mod. pro Invertierung	Geschw. Faktor
1	18.	1.0
2	10.5	1.5
3	8.	1.9
5	6.	2.2
10	4.5	2.7
20	3.75	2.9
30	3.5	3.0
50	3.3	3.1
100	3.15	3.2
1000	3.015	3.2

5.2 Die Continuations

Wie stark beschleunigen die Standard-Continuation und die verbesserte Standard-Continuation ECM in der Praxis?
Um dies zu testen wurden 50 Zahlen — zusammengesetzt aus einer 20- und einer 80-stelligen Primzahl — faktorisiert. Die gemessenen Werte können nur sinnvoll mit den theoretisch vorhergesagten verglichen werden, wenn man zu jedem einzelnen 20-stelligen Faktor optimale Parameter bestimmt.
Das wiederum bewirkt, daß der Test von den "großen" 20-stelligen Faktoren dominiert wird. Deshalb wurden die zu faktorisierenden Zahlen folgendermaßen erzeugt:

1. Generiere eine 17-stellige Zufallszahl (unter Verwendung der C-Funktion *random()*; s. Funktion *lrandom()*, S.139) und addiere sie mit $5 \cdot 10^{19}$.
2. Suche in aufsteigender Richtung nach der nächsten Primzahl (s. Funktion *nextprime()*, S. 134). Diese Primzahl hat 20 Stellen und beginnt mit '500'.
3. Erzeuge eine 80-stellige Zufallszahl (wie in Schritt 1).
4. Suche nächste Primzahl (s. Schritt 2).
5. Multipliziere die Zahlen aus Schritt 2 und Schritt 4.

Das Ergebnis ist eine 99- bis 100-stellige Zahl, die aus einem 20-stelligen und einem 80-stelligen Faktor besteht. Für alle 20-stelligen Faktoren gelten jetzt die gleichen optimalen Parameter.
Die so erzeugten Zahlen wurden mit dem ECM-Grundalgorithmus, dann mit ECM + Standard-Continuation und schließlich mit ECM + verbesserter Standard-Continuation faktorisiert. Dabei wurden der Kurvengenerator aus Kapitel 2.2.3 und der ggT-Trick mit 15 Kurven simultan benutzt und es wurden optimale Parameter gewählt.
Die Tests wurden auf "SUN SPARC station ELC" (mit 21 MIPS (Million Instructions Per Second)) durchgeführt.

5.3 Der Kurvengenerator

Um die Wirksamkeit des Kurvengenerators zu testen, wurden die gleichen 50 Zahlen wie im vorherigen Test mit ECM + verbesserter Standard-Continuation faktorisiert und zwar zunächst mit dem in Algorithmus 2.9 (S. 27) beschriebenen Standardverfahren und dann mit dem verbesserten Generator aus Kapitel 2.2.3. Dabei wurden optimale Parameter gewählt.
Die Tests wurden auf "SUN SPARC station ELC" (mit 21 MIPS) durchgeführt.

Tabelle5.2: *Beschleunigung durch Continuations*

	ECM	ECM + Std.Cont.	ECM + verb.Std.Cont
B_1	38 005	20 467	15 752
B_3	—	122 190	1 219 430
erwartete #Kurven	311	182	98
#Kurven: MIN	15	15	15
#Kurven: MAX	1530	855	330
#Kurven: MITTEL	301	198	99
benötigte Zeit[m]: MIN	91	34	44
benötigte Zeit[m]: MAX	13 861	5 099	1 842
benötigte Zeit[m]: MITTEL	2 662	1 161	538
Faktor	1	2.3	5

Tabelle5.3: *Beschleunigung durch Kurvengenerator*

	Standard-Generator	verbesserter Generator
B_1	19 169	15 752
B_3	1 468 779	1 219 430
erwartete #Kurven	113	98
#Kurven: MIN	15	15
#Kurven: MAX	660	330
#Kurven: MITTEL	133	99
benötigte Zeit [m]: MIN	75	44
benötigte Zeit [m]: MAX	4 334	1 842
benötigte Zeit [m]: MITTEL	860	538
Faktor	1	1.6

5.4 Die Größe der zu faktorisierenden Zahl

Die Laufzeit von ECM hängt in erster Linie von der Größe der gesuchten Faktoren ab. Die Geschwindigkeit der einzelnen Rechenoperationen hängt aber von der Größe der zu faktorisierenden Zahl ab.
Um diese Abhängigkeit zu untersuchen, wurde eine x-stellige Primzahl erzeugt ($x \in \{50, 75, \ldots, 200\}$) und die Zeitdauer eines erfolglosen Versuchs von ECM (verbesserte Std.Cont., 15 Kurven simultan, verbesserter Kurvengenerator) unter Verwendung optimaler Parameter für 20-stellige Faktoren gemessen.

Tabelle5.4: *Abhängigkeit der Laufzeit von der Größe der zu faktorisierenden Zahl*

#Stellen	50	75	100	125	150	175	200
Faktor	0.4	0.6	1	1.4	1.9	2.6	3.1

5.5 Die Größe der gesuchten Faktoren

Wieviel Zeit benötigt ECM im Mittel, um nach einem x-stelligen Faktor zu suchen? Um das herauszufinden wurde eine 100-stellige Primzahl erzeugt und die Zeitdauer eines erfolglosen Versuchs von ECM (mit verbesserter Std.Cont., 15 Kurven simultan, verbesserter Kurvengenerator, optimale Parameter für x-stelligen Faktor) gemessen.
Die Gesamtzeit berechnet sich aus:

$$\text{Gesamtzeit} = \text{Zeit für einen Versuch} \cdot \frac{\#\text{Kurven}}{15},$$

da in einem Versuch 15 Kurven simultan betrachtet werden.
Die Tests wurden auf "SUN SPARC station ELC" (mit 21 MIPS) durchgeführt.
Mit Hilfe von Tabelle 5.4 kann man approximativ auf die Laufzeit für nicht 100-stellige Zahlen schließen.

Tabelle5.5: *Zeitbedarf*

#Stellen	B_1	B_3	Zeit pro Versuch	Gesamtzeit
10	411	16 185	90 s	1 m 30 s
12	928	45 579	316 s	5 m 16 s
14	1 992	120 452	712 s	18 m 11 s
16	4 175	269 720	1 391 s	58 m 43 s
18	8 177	617 457	2 704 s	3 h 3 m 16 s
20	15 752	1 219 430	5 047 s	9 h 9 m 33 s
22	29 505	2 465 816	9 082 s	25 h 43 m 56 s
24	55 323	4 869 607	16 967 s	72 h 34 m 51 s
26	99 906	8 900 812	29 803 s	194 h 16 m 17 s
28	178 459	15 653 379	52 129 s	511 h 38 m 11 s
30	309 156	29 284 729	90 504 s	1305 h 36 m 14 s

Kapitel 6

Der RSA Faktorisierungs-Wettbewerb

6.1 Spielregeln

Seit dem 1.4.1991 veranstaltet "RSA Data Security" einen Wettbewerb zur Faktorisierung großer Zahlen, den "RSA Factoring Challenge". Er basiert auf zwei Zahlenlisten — der RSA-Liste und der Partitions-Liste.

Die RSA-Liste enthält Zahlen der Art, wie sie in RSA-Cryptosystemen verwendet werden. Diese Zahlen bestehen aus zwei Primfaktoren ungefähr gleicher Größenordnung. Daher sind sie für ECM nicht geeignet. Man vermutet, daß diese Zahlen, die am schwierigsten zu faktorisierenden sind. Die größte bisher (Dezember 1992) faktorisierte RSA-Zahl hat 110 Dezimalstellen und wurde am 14.4.1992 von Arjen K. Lenstra faktorisiert.

Die Partitions-Liste enthält Partitionszahlen. Die Partitionszahl $p(n)$ ist die Anzahl der Möglichkeiten, n als Summe natürlicher Zahlen darzustellen, wobei die Möglichkeiten, die durch Vertauschung der Summanden entstehen , nicht mitgezählt werden.

Beispiel:
$p(5) = 7$, denn es gibt folgende Möglichkeiten:

$$\begin{array}{l} 5 \\ 4+1 \\ 3+2 \\ 3+1+1 \\ 2+2+1 \\ 2+1+1+1 \\ 1+1+1+1+1 \end{array}$$

Partitionszahlen scheinen sich — bezüglich der Schwierigkeit sie zu faktorisieren — mehr oder weniger zufällig zu verhalten. Einige von ihnen, u.a. auch sehr große, sind einfach zu faktorisieren, andere scheinen sehr schwierig zu sein.
Die kleinste Partitionszahl der Liste ist $p(8681)$ (100 Dezimalstellen); eine obere Schranke gibt es nicht.
Um die Partitionsliste auf eine vernüftige Größe zu reduzieren, werden nur Partitionszahlen $p(n)$, mit $n \in \mathbb{P}$, in die Liste aufgenommen. Damit gibt es circa 20 Partitionszahlen gleicher Länge.
In jedem Quartal eines Jahres stehen 1750\$ Preisgeld aus. Für jede Faktorisierung erhält der Faktorisierer eine gewisse Anzahl Punkte. Am Ende des Quartals wird das Preisgeld unter allen Faktorisierern im Verhältnis der verdienten Punkte verteilt. Dabei werden die Punkte für eine faktorisierte Zahl nach folgendem Verfahren vergeben:
Sei D die Anzahl der Dezimalstellen, der kleinsten noch nicht faktorisierten Partitionszahl. Für die Faktorisierung einer Zahl mit $D + d$ Dezimalstellen gibt es

$$\begin{array}{rl} 8 \text{ Punkte,} & \text{falls } 0 \le d \le 9, \\ 4 \text{ Punkte,} & \text{falls } 10 \le d \le 19, \\ 2 \text{ Punkte,} & \text{falls } 20 \le d \le 29, \\ 1 \text{ Punkt,} & \text{falls } 30 \le d \le 49, \end{array}$$

Beispiel:
Angenommen, die kleinste noch nicht faktorisierte Partitionszahl hätte 101 Dezimalstellen. Für eine 114-stellige Zahl wäre $d = 13$ und somit gäbe es 4 Punkte für ihre Faktorisierung.

Das Preisgeld berechnet sich am Ende des Quartals folgendermaßen:

$$\text{mein Preis} = 1750\$ \cdot \frac{\text{meine Punkte in diesem Quartal}}{\text{alle Punkte des Quartals}}$$

Ausführliche Informationen zu diesem Wettbewerb kann man per Email erhalten: *challenge-administrator@rsa.com*

6.2 Erfolge

Seit Februar 1992 habe ich — zunächst alleine, später zusammen mit Thomas Denny, der den "quadratischen Sieb"-Faktorisierungsalgorithmus (s. [Denny]) implementiert hat — an dem Partitionszahlen-Wettbewerb teilgenommen. Die Laufzeit des quadratischen Siebes hängt im wesentlichen von der Größe der zu faktorisierenden Zahl ab und nicht wie bei ECM von der Größe der Faktoren.

Somit ergibt sich folgende Arbeitsteilung:

1. Zunächst versuchen wir die Zahl mit ECM zu faktorisieren. Dabei werden in den meisten Fällen einige Faktoren der Zahl gefunden, oft wird sie auch ganz faktorisiert.

2. Wenn der unfaktorisierte Rest der Zahl nicht zu groß ist (≤ 103 Dezimalstellen), faktorisieren wir den Rest mit dem quadratischen Sieb.

Ein gutes Beispiel für die Zusammenarbeit von ECM und quadratischem Sieb (QS) ist die Partitionszahl $p(16061)$:

p(16061) =	1357153565271906410567488178694112281797325899285084891292376191472647 3018495663123856411014510552215074053652608270803159515983426037527 (137)	
=	3 ∗ 7	(Probedivision)
∗	170187049 (9)	(ECM)
∗	3012281611 (10)	(ECM)
∗	3695653834427 (13)	(ECM)
∗	421678025035168873387659731 (27)	(ECM)
∗	104123697943922465529161208861387481741 (39)	(QS)
∗	7769011802081511836441540847585396776749 (40)	(Rest)

$p(16061)$ ist mit 137 Dezimalstellen viel zu groß um von dem quadratischen Sieb faktorisiert zu werden. Nachdem ECM die kleinen Faktoren gefunden hat, kann die Restzahl (79 Dezimalstellen) leicht mit dem quadratischen Sieb faktorisiert werden.

Für den Wettbewerb benutzte ich die verteilte Implementierung von ECM auf 6 - 24 Sun Sparc Rechnern.
Folgende Erfolge konnten wir verbuchen: (Auszug aus der Orginal-Liste; erhältlich per Email: *challenge-news@rsa.com*)

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


        (1st Quarter, 1992)
                122 points, Arjen Lenstra, $1405
                 12 points, Franz-Dieter Berger, Johannes Buchmann, $138
                  8 points, Peter L. Montgomery, $92
                  6 points, Peter L Montgomery & Paul C Leyland, $69
                  4 points, Lambert E. Bright, $46
                --------------
                152 points total

        (2nd Quarter, 1992)
                 78 points, Arjen Lenstra, $989
                 48 points, Franz-Dieter Berger, Thomas Denny,
                            Johannes Buchmann, $609
                  5 points, Peter L. Montgomery, $63
                  4 points, Peter L Montgomery & Paul C Leyland, $51
```

```
          3 points, Paul C Leyland, $38
        --------------
        138 points total

(3rd Quarter, 1992)
          1 point, Heinz-Bernd Eggenstein, $11
         84 points, Arjen K. Lenstra, $907
          1 point, Paul C Leyland & Arjen K Lenstra, $11
         58 points, Franz-Dieter Berger, Thomas Denny,
                   Johannes Buchmann, $626
         12 points, Paul C Leyland, $130
          2 points, Francois MORAIN, Reynald LERCIER, $22
          4 points, FactOregon (Joe Buhler, Peter Montgomery,
                                Robby Robson, Russell Ruby), $43
        --------------
        162 points total

(4th Quarter, 1992)
         79 points, Arjen K. Lenstra, $634
          4 points, Paul Leyland, Arjen K. Lenstra, $32
          4 points, Bruce Dodson, Arjen K. Lenstra, $32
         60 points, Franz-Dieter Berger, Thomas Denny,
                    Johannes Buchmann, $482
          6 points, FactOregon (Joe Buhler, Peter Montgomery,
                                 Robby Robson, Russell Ruby), $48
         52 points, Bruce Dodson, Matt Fante, $418
          5 points, Paul C Leyland, $40
          3 points, Heinz-Bernd Eggenstein, $24
          5 points, Reynald Lercier, Francois Morain, $40
        --------------
        218 points total
```

Insgesamt haben wir bisher (8.4.1993) 60 Partitionszahlen faktorisiert. 29 Partitionszahlen konnten alleine mit ECM faktorisiert werden.

In folgenden Partitionszahlen fand ich Faktoren mit mehr als 30 Dezimalstellen:

$$
\begin{array}{rll}
p(11833) = & 184467285865046351113629574228875258436884220573010497102492616447201 5 & \\
& 79319663093781776233435018256119921109493164362\ (117) & \\
= & 2 * 13 * 197 & \text{(Probedivision)} \\
* & 150730009\ (9) & \text{(ECM)} \\
* & 7303940514605475106925897122243\ (31) & \text{(ECM)} \\
* & 201739947642050852926859105393645149\ (36) & \text{(QS)} \\
* & 162155217607734022041916747408934450267\ (39) & \text{(Rest)} \\
\\
p(12289) = & 36509043637951027921639488171896799624034076897792412434271636399643 23 & \\
& 07884423213133965745115830346570635581345403302 65\ (119) & \\
= & 5 * 23 * 47 * 1151 * 77279 & \text{(Probedivision)}
\end{array}
$$

$*$	1504829922334822998616683989281 (31)	(ECM)
$*$	504638665183653463683967885897419833746454839 01261160085134414952158625552237 (77)	(Rest)
p(13183) =	88123157638995167454479435452332368796324047940524754538856079416744 16 376112518470061725697857313644549315097674925406668 94 (123)	
=	2 $*$ 11 $*$ 389	(Probedivision)
$*$	6846629530160084499061770536857 (31)	(ECM)
$*$	120095643304272159964331725449782632238558887 6 1457190157576613864621860151407386 33 (82)	(Rest)
p(13259) =	20453216284308522401567690551927032284184759319478110821635486121636 66 9378281467874774250076265006453851285145949473085050 55 (124)	
=	3 $*$ 3 $*$ 3 $*$ 3 $*$ 3 $*$ 3 $*$ 5 $*$ 97 $*$ 5923	(Probedivision)
$*$	1152974459211011500192387953697 (31)	(ECM)
$*$	8470926713171671089637276002006614065751977827 06105435941829181560276804243538657337 (84)	(Rest)
p(14503) =	1427580940259080549169607547199498185754694668571713101653784185225946 018805694263430085854407107267303172502295571522692393118736 (130)	
=	2 $*$ 2 $*$ 2 $*$ 2 $*$ 3 $*$ 163 $*$ 16787	(Probedivision)
$*$	5198799459937662960498090256961 (31)	(ECM)
$*$	170643781632241953719405424485582201 (36)	(QS)
$*$	12251951719172747947825890805487332846862116992709680827 (56)	(Rest)
p(13411) =	1093944565956458905254799216531318742271380657571794359760863338053027 565457012852564313970537828668956200381957691188550 4268 (125)	
=	2 $*$ 2 $*$ 3 $*$ 3 $*$ 983	(Probedivision)
$*$	1848227 (7)	(ECM)
$*$	479690049143023 (15)	(ECM)
$*$	23451809309139809275480162988321 (32)	(ECM)
$*$	1486780726945345416312042221840611432368724452 6667110166011554403521 (68)	(Rest)
p(14969) =	1901905674691375293136796195126264805824654971247608424130915771054028 065549463993640174769019217426099415635404359921392984216441 35 (132)	
=	5 $*$ 17 $*$ 191	(Probedivision)
$*$	16921129 (8)	(ECM)
$*$	65078234903869234058651531927033 (32)	(ECM)
$*$	50188377931059682981333 (23)	(QS)
$*$	2119670814124640982881277275706350392283039681 45634142802841869761 (66)	(Rest)
p(17327) =	3617815886048403102392304776841251403378547247809406431748925919072370 2636673516121006963772046170014393331454792185760284714345076240844726 76 (142)	
=	2 $*$ 2 $*$ 7 $*$ 331	(Probedivision)
$*$	3031697811319059303467100829 (28)	(ECM)

	∗	41157695128075275649555482225821 (32)	(ECM)
	∗	31284086660644332760030897311901059368791114978691015329831828424904096 71217173 (79)	(Rest)
p(15331) =		80730227937093753921919106733733205029900913884624431148265934228540666812899544324635911701620202736613969936575242107361910325802 11 (133)	
	=	7 ∗ 23 ∗ 1069	(Probedivision)
	∗	1290529759 (10)	(ECM)
	∗	7309617612102220163101553680020539 (34)	(ECM)
	∗	4972444168859388903191603243524421466526310358 85696221713594597318046038965754132 2579 (85)	(Rest)
p(19997) =		24538449781502559387588015934978438372594530738257742724693177912862609586215364110306539593407153904932672685091289125585923732501995956428 9811964383337 (153)	
	=	7 ∗ 11 ∗ 29 ∗ 41 ∗ 101 ∗ 3467 ∗ 4373	(Probedivision)
	∗	5688864305048653702791752405107044435136231 (43)	(ECM)
	∗	3076764636874201276407517268928886129890878071316637548474204994035569 591290900992021909241973 49 (96)	(Rest)

Während der Teilnahme an dem Wettbewerb, lieferte ECM einige Überraschungen:

- Bei der Faktorisierung von p(19997) fand ich einen 43-stelligen Faktor mit Parametern für 27 Stellen ($B_1 \approx 140\,000$, $B_2 = B_3$, $B_3 \approx 14\,200\,000$) . Das ist weltweit der größte von ECM gefundene Faktor (Stand: März 1993).
- Der 34-stellige Faktor in $p(15331)$ wurde ebenfalls innerhalb kurzer Zeit (1,2 MIPS-Jahre) mit Parametern für 26 Stellen ($B_1 \approx 110000$, $B_2 = B_3$, $B_3 \approx 8\,800\,000$) gefunden.
- In $p(14969)$ fand ich den 32-stelligen Faktor, stoppte dann ECM, da es mir sehr unwahrscheinlich erschien, einen noch größeren Faktor zu finden und übersah den 23-stelligen Faktor, der dann durch das quadratische Sieb gefunden wurde. Ähnliche, aber bei weitem nicht so extreme Fälle gab es bei $p(15497)$ (ECM 28, QS 26) und $p(16001)$ (ECM 26, QS 25).

Kapitel 7

Ausblick

Folgende Verbesserungen können ECM noch beschleunigen.

- Kurvengenerator mit Gruppenordnung, die durch 16 teilbar ist
  Hier werden Kurven erzeugt, deren Gruppenordnung durch 16 teilbar ist (s. [Atkin/Morain]). Vergleicht man die optimalen Parameter und die mittlere Anzahl zu testender Kurven für 30 stellige Faktoren, so sieht man, daß hier noch eine beträchtliche Beschleunigung zu erwarten ist.

Gruppenordnung teilbar durch	B_1	B_3	#Kurven
4	309 156	29 284 729	779
16	267 785	24 087 397	686

- Parametrisierung nach Montgomery (s. [Montgomery])
  Durch diese Parametrisierung kann man auf den ggT-Trick verzichten. Montgomery wählt bestimmte Kurven, auf denen er mit einer speziellen (nicht der allgemeinen) Punktaddition rechnen kann. Diese Addition ist auf einer einzigen Kurve genauso schnell wie die allgemeine Addition mit ggT-Trick im optimalen Fall. Da ich aber nicht beliebig viele Kurven gleichzeitig betrachten kann, sondern mich konkret auf 15 Kurven beschränkt habe, wird die Montgomery-Parametrisierung die erste Phase von ECM (nicht die Continuation) um einige Prozent beschleunigen. Der eigentliche Vorteil liegt aber in der besseren Parallelisierbarkeit von ECM.

- FFT-Continuation (s. [Montgomery2])
  Sie soll für Faktoren mit mehr als 25 Dezimalstellen schneller sein.

- Beschleunigung der Berechnung $k \cdot P$
  Hier lassen sich durch geschickte Wahl von k 18% Rechenzeit sparen
  (s. [Dixon/A. Lenstra]).

Anhang A

Definitionen und Sätze

Satz A.1 (Fundamentalsatz der Arithmetik) *:*
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: s. [Bundschuh]. ■

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

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

Ordnung *der Gruppe (s. [Lamprecht]).*

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

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

die Ordnung des Elementes a (s. [Lamprecht]).

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

Beweis: s. [Lamprecht]. ■

Satz A.5 *:*
Ist $(G;\cdot)$ eine endliche Gruppe mit neutralem Element 1 und $a \in G$, so ist

$$a^{|G|} = 1$$

Beweis: s. [Lamprecht]. ■

Satz A.6 (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 findet man in [Bundschuh]. ■

Definition A.7 (π-Funktion)

$$\begin{aligned} \pi : \mathbb{R} &\rightarrow \mathbb{N}_0 \\ \pi(x) &:= \#\{p \in \mathbb{P} : p \leq x\} \end{aligned}$$

$\pi(x)$ gibt die Anzahl der Primzahlen $\leq x$ aus.

Satz A.8 *:*
Es gilt:

$$\lim_{x\to\infty} \pi(x) = \frac{x}{\ln x}.$$

Beweis: Einen Beweis findet man in [Bundschuh]. ■

Definition A.9 (Eulersche φ-Funktion)

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

$\varphi(n)$ ist die Anzahl, der natürlichen, n nicht übersteigenden Zahlen, die zu n teilerfremd sind.

Anhang B

Zahlentheoretische Algorithmen

Algorithmus B.1 Berechnung von Inversen modulo mit Hilfe des erweiterten euklidischen Algorithmus (ggT mit Darstellung)
Sei $g = \mathrm{ggT}(a, N)$ mit $a, N \in \mathbb{Z}$. Der erweiterte euklidische Algorithmus findet Zahlen $u, v \in \mathbb{Z}$, so daß gilt:

$$a \cdot u + N \cdot v = g.$$

Ist $g = 1$, dann gilt:

$$a \cdot u \equiv 1 \bmod N,$$

denn $N \cdot v \equiv 0 \bmod N$.
Das heißt:

$$u \equiv a^{-1} \bmod N.$$

Wenn $g \neq 1$ ist, existiert das Inverse nicht! In diesem Fall hat man einen Teiler von N gefunden. Eine ausführliche Beschreibung des Algorithmus findet man in [Knuth].

Algorithmus B.2 Berechnung der ρ-Funktion

$$\rho(\alpha) = \begin{cases} 1, & \text{für } 0 \leq \alpha \leq 1 \\ \dfrac{1}{\alpha} \displaystyle\int_{\alpha-1}^{\alpha} \rho(t)dt, & \text{für } \alpha > 1 \end{cases}$$

Das Integral lässt sich mit der summierten Trapezregel (s. [Rannacher]) berechnen. Sei n die Anzahl der Teilintervalle; die Intervallänge ist

$$h = \frac{(\alpha - (\alpha - 1))}{n} = \frac{1}{n}$$

Sei $\alpha > 1$, dann ist

$$\begin{aligned}
\rho(\alpha) &\approx \frac{1}{\alpha}\cdot\frac{h}{2}\cdot\left(\rho(\alpha-1)+2\sum_{i=1}^{n-1}\rho(x_i)\ +\rho(\alpha)\right), \qquad x_i = \alpha - i\cdot h \\
&= \frac{h}{2\alpha}\rho(\alpha)+\frac{h}{2\alpha}\left(\rho(\alpha-1)+2\sum_{i=1}^{n-1}\rho(x_i)\right) \\
\Leftrightarrow \quad \rho(\alpha) &\approx \frac{h}{2\alpha-h}\left(\rho(\alpha-1)+2\sum_{i=1}^{n-1}\rho(x_i)\right).
\end{aligned}$$

Auf der rechten Seite der Gleichung sind alle Argumente der ρ-Funktion kleiner als α. Für $0 \le \alpha \le 1$ ist $\rho(\alpha) = 1$. Somit kann man mit obiger Formel $\rho(1+h)$ berechnen. Diesen Wert speichert man in einer Tabelle. Nun kann man $\rho(1+2h)$ ausrechnen, den Wert abspeichern, $\rho(1+3h)$ berechnen u.s.w., bis man sich zu $\rho(\alpha)$ vorgearbeitet hat. Die Tabelle muß n Einträge groß sein, da alle Funktionswerte zwischen $\alpha - 1$ und α gespeichert werden müssen.

Anhang C

Die Funktionen

C.1 Funktionen für elliptische Kurven

Für die in den Funktionsaufrufen verwendeten Variablen gelten folgende Vereinbarungen:

LONGTYPE	N,	die zu faktorisierende Zahl
	factor;	Faktor von N
Curve	C;	
Point	P, Q;	
Vector_of_curves	VC;	
Vector_of_points	VP, VQ, VErg;	
int	size, quantity;	
unsigned long	ul, B_1, B_2, B_3;	

C.1.1 Arithmetik

Boolean ecgen(C, P, N, factor) — elliptic curve generator

Ergebnis:	TRUE,	falls ein Faktor gefunden wurde
	FALSE,	sonst

ecgen(C, P, n, factor) generiert — mit dem in Kapitel 2.2.3 beschriebenen Verfahren — eine elliptische Kurve C, deren Diskriminate prim zu N ist, und einen Punkt P, der auf der Kurve liegt. Wird dabei schon ein Faktor von N gefunden, so wird er mit factor zurueckgeliefert. Der *Curve*-Zeiger des Punktes wird auf die entsprechende Kurve gesetzt.

Boolean ecvgen(VC, VP, N, factor) — elliptic curve vector generator

Ergebnis:	TRUE,	falls ein Faktor gefunden wurde
	FALSE,	sonst

Wie ecgen(), aber hier werden mehrere Kurven mit Punkten generiert.

int epcomp(P, Q) — elliptic point compare: (P == Q)?

Ergebnis:	0,	falls (P == Q)
	1,	sonst

int epnegcomp(P, Q) — elliptic point negation compare: (P == -Q)?

Ergebnis:	0,	falls (P == -Q)
	1,	sonst

void epsto(P,Q) — elliptic point store (P $\rightarrow$ Q)

Speichert alle Komponenten von Punkt P in den Punkt Q.

Status epvadd(VP, VQ, VErg, N, factor) — elliptic point vector addition

Ergebnis:	factor_found,	falls ein Faktor gefunden wurde
	hopeless,	wenn <= MINCDIM Kurven aktiv sind
	no_factor_found,	sonst

VErg = VP + VQ. Das Ergebnis stimmt nur dann, wenn epvadd() den Status *no_factor_found* geliefert hat.

Status epvmul(ul, VP, VErg, N, factor) — elliptic point vector multiplication

Ergebnis:	factor_found,	falls ein Faktor gefunden wurde
	hopeless,	wenn <= MINCDIM Kurven aktiv sind
	no_factor_found,	sonst

VErg = ul * VP. Multipliziert einen Punktevektor mit einem Skalar. VErg ist nur dann korrekt, wenn *epvmul()* den Status *no_factor_found* liefert.

Status ecvfindfactor(B_1, B_2, N, VP, pp, factor) — find a factor with ECM

Ergebnis:	factor_found,	falls ein Faktor gefunden wurde
	hopeless,	wenn <= MINCDIM Kurven aktiv sind
	no_factor_found,	sonst

ECM-Algorithmus (ohne Continuation).

Status ecm_stdcont(VP, B_3, N, pp, factor) — Standard-Continuation

Ergebnis:	factor_found,	falls ein Faktor gefunden wurde
	hopeless,	wenn <= MINCDIM Kurven aktiv sind
	no_factor_found,	sonst

Status ecm_stdcont2(VP, B_1, B_3, N, pp, factor) — verbesserte Standard-Continuation

Ergebnis:	factor_found,	falls ein Faktor gefunden wurde
	no_factor_found,	sonst

Der Status *hopeless* wird niemals zurückgegeben!

C.1.2 Speicherverwaltung

Curve ecalloc(size) — elliptic curve allocate

Reserviert Speicherplatz für eine elliptische Kurve.
size bestimmt die Größe der *LONGTYPEs*.
Das Ergebnis ist ein Pointer auf den reservierten Speicher.

void ecfree(C) — elliptic curve free

Gibt den mit *ecalloc()* reservierten Speicher wieder frei.

Curve ecvalloc(quantity, size) — elliptic curve vector allocate

Reserviert Speicherplatz für *quantity* viele elliptische Kurven.
size bestimmt die Größe der *LONGTYPEs*.
Das Ergebnis ist ein Pointer auf den reservierten Speicher.

void ecvfree(C) — elliptic curve vector free

Gibt den mit *ecvalloc()* reservierten Speicher wieder frei.

Curve epalloc(size) — elliptic curve allocate

Reserviert Speicherplatz für einen Punkt.
size bestimmt die Größe der *LONGTYPEs*.
Das Ergebnis ist ein Pointer auf den reservierten Speicher.

void epfree(C) — elliptic curve free

Gibt den mit *epalloc()* reservierten Speicher wieder frei.

Curve epvalloc(quantity, size) — elliptic curve vector allocate

Reserviert Speicherplatz für *quantity* viele Punkte.
size bestimmt die Größe der *LONGTYPEs*.
Das Ergebnis ist ein Pointer auf den reservierten Speicher.

void epvfree(C) — elliptic curve vector free

Gibt den mit *epvalloc()* reservierten Speicher wieder frei.

C.1.3 Sonstige Funktionen

void ecwrite(C) — elliptic curve write

Die Komponenten der elliptische Kurve (d.h. die Parameter a und b) werden ausgegeben. Ist die Kurve nicht aktiv, so wird die Nachricht "Curve is not activ!" ausgegeben.

void epwrite(P) — elliptic point write

Gibt die x- und y-Koordinate eines Punktes aus, dann wird mit Hilfe von *ecwrite()* die dazugehörige Kurve ausgegben.

int eponcurve(P, N) — elliptic point on curve

Ergebnis:	1,	falls P auf der dazugehörigen Kurve liegt
	0,	sonst

Testet ob ein Punkt auf der dazugehörigen Kurve liegt.

int epvlenght(VP) — elliptic point vector lenght

Gibt die Anzahl der Punkte von *VP* aus.

int epvactiv_curves(VP) — elliptic point vector active curves

Liefert die Anzahl der Punkte aus *VP*, die auf aktiven Kurven liegen.

C.2 Funktionen auf langen Zahlen

Diese Funktionen halten sich eng an die in der [larith] getroffenen Konventionen. Soweit nicht anders vermerkt, sind sie vom Typ *int* und liefern im Fehlerfall -1, andernfalls 0.
Außerdem gelten folgende Variablen-Vereinbarungen:

LONGTYPE	x, y, erg, n;
int	i;

C.2.1 Rechnen in $\mathbb{Z}/p\mathbb{Z}$

llmod(x, n, erg) — erg ← x mod n

Das Ergebnis ist immer 0.

lladdmod(x, y, n, erg) — erg ← (x+y) mod n

Die Modulo-Reduktion wird — wenn nötig — durch eine Addition oder eine Subtraktion mit n erreicht. Deshalb müssen x und y bei Aufruf der Funktion schon modulo n reduziert sein!!
Das Ergebnis ist immer 0.

llsubmod(x, y, n, erg) — erg ← (x-y) mod n

Die Modulo-Reduktion wird — wenn nötig — durch eine Addition oder eine Subtraktion mit n erreicht. Deshalb müssen x und y bei Aufruf der Funktion schon modulo n reduziert sein!!
Das Ergebnis ist immer 0.

llmulmod(x, y, n, erg) — erg ← (x*y) mod n

Die Modulo-Reduktion wird mit einer Division erledigt.
Das Ergebnis ist immer 0.

void lpotmod(x, y, n, erg) — erg ← x^y mod n

Hier wird der binäre Algorithmus (s. [Horster]) verwendet.

lmul2mod(x, n) — x ← (x*2) mod n

Die Modulo-Reduktion wird — wenn nötig — durch eine Addition oder eine Subtraktion mit n erreicht. Deshalb muß x bei Aufruf der Funktion schon modulo n reduziert sein!! Da x zuerst nach links geshiftet und dann reduziert wird, ist das Zwischenergebnis 1 Bit größer als x. Der Speicherplatz für x muß für diese Operation genügend groß sein.
Das Ergebnis ist immer 0.

lsaddmod(x, i, n, erg) — erg ← (x+i) mod n

Die Modulo-Reduktion wird — wenn nötig — durch eine Addition oder eine Subtraktion mit n erreicht. Deshalb muß x bei Aufruf der Funktion schon modulo n reduziert sein!!
Das Ergebnis ist immer 0.

lssubmod(x, i, n, erg) — erg ← (x-i) mod n

Die Modulo-Reduktion wird — wenn nötig — durch eine Addition oder eine Subtraktion mit n erreicht. Deshalb muß x bei Aufruf der Funktion schon modulo n reduziert sein!!
Das Ergebnis ist immer 0.

lsmulmod(x, i, n, erg) — erg ← (x*i) mod n

Die Modulo-Reduktion wird mit einer Division erledigt.
Das Ergebnis ist immer 0.

llcompmod(x, y, n) — x ≡ y mod n ?

Ergebnis:	1,	falls x > y
	0,	falls x == y
	-1,	falls x < y

x und y müssen dazu modulo n reduziert sein.

C.2.2 Sonstige Arithmetikfunktionen auf langen Zahlen

Boolean listprim(x) — ist x eine Primzahl?

Ergebnis:	1,	falls x Primzahl
	0,	sonst

lrandom(i, x) — Zufallszahlengenerator

Erzeugt eine Zufallszahl mit i Dezimalstellen.

unsigned int ltrialdiv(N) — Probedivision

void initltrialdiv(limit) — Initialisierung

void resetltrialdiv(limit) — Reset

ltrialdiv(N) dividiert N durch alle Primzahlen $<$ *limit* (angefangen bei 2). Wurde ein Faktor gefunden, so wird er von *ltrialdiv()* zurückgeliefert. Ein erneuter Aufruf von *ltrialdiv(N)* setzt die Arbeit an der abgebrochenen Stelle fort.
initltrialdiv(limit) setzt den Limit für die Probedivision.
resetltrialdiv(limit) setzt einen neuen Limit; der nächste Aufruf von *ltrialdiv()* beginnt wieder mit der Division durch 2.

Boolean lvinvert(a, N, erg, vsize) — Inversenberechnung modulo

Vector_of_LONGTYPES a, erg;
int vsize; /* Länge des Vektors */
Alle Elemente von a werden modulo N invertiert. Dabei wird die simultane ggT-Berechnung angewandt (s. S. 35).

Ergebnis:	1,	falls die Inversen berechnet werden konnten
	0,	sonst

C.2.3 Speicherverwaltung

Vector_of_LONGTYPES lvalloc(quantity, size) — LONGTYPE vector allocate
Reserviert Speicher für einen Vector, bestehend aus *quantity* LONGTYPES der Länge *size*.

void lvfree(L) — LONGTYPE vector free

Anhang D

Mathematika-Programme

D.1 Funktionen

D.1.1 ρ-Funktion

```
$RecursionLimit=1000
rho[a_] := 1 /; a <= 1
rho[a_] :=
   rho[a] = Block[{i, h = 0.01, n = 1/h},
     (h*rho[a - 1] + 2*h*Sum[rho[a - 1 + i*h], {i, n - 1}])/(2*a - h)]
```

D.1.2 μ-Funktion

```
$RecursionLimit=1000
Clear[mue]
mue[ah_, bh_] := 0.0000000001 /; ah<bh+1 || ah<2
mue[ah_, bh_] := Block[{i, h=.01, n=(bh-1)/h, a, b},
a = N[Round[ah*100]/100];
b = N[Round[bh*100]/100];
Return[rho[a] + h/2 (rho[a-b]/b + 2 Sum[rho[a-b+i*h]/(b-i h),
                {i, n-1}] + rho[a-1])]
]
```

D.2 optimale Parameter

D.2.1 optimale Parameter für ECM

```
Clear[ecm1par]
ecm1par[log10p_, cg_] :=
Block[{step, h, p, lnp, a, B1},
p = 10^log10p/cg;
step = 2.56;
a = 3.44;
```

```
lnp = N[Log[p]];
While[step >= 2/100,
step =step/2;
h = N[a*rho[a-1]/rho[a]];
If[h > lnp, a = a-step, a = a + step];
];
B1 = Round[p^(1/a)];
Return[{a, B1, Round[1/rho[a]]}]
]
ecm1par::usage = "ecm1par[log10p, cg] berechnet die Parameter fuer den
1-Phasen-ECM-Algorithmus.  log10p ist die Anzahl der Dezimalstellen
des zu suchenden Faktors, cg der Divisor vom Kurvengenerator.
Die Ausgabe ist eine Liste mit folgenden Werten: {alpha, B1, Anz.Kurven}.
Achtung: Sinnvolle Werte werden nur bis 45 Stellen berechnet!"
```

D.2.2 optimale Parameter für ECM mit Continuation

Benutzt man ECM mit Standard-Continuation, so berechnen sich die Parameter $c1\_$ und $c2\_$ folgendermaßen:

$$
\begin{aligned}
c1\_ &= 0, \qquad \text{da keine Tafel berechnet wird,}\\
c2\_ &= \frac{c_2}{c_1}, \quad \text{(s. S. 31).}
\end{aligned}
$$

Für ECM mit verbesserter Standard-Continuation gilt:

$$
\begin{aligned}
c1\_ &= \frac{c_2}{c_1}, \quad \text{(s. S. 45),}\\
c2\_ &= \frac{c_3}{c_1}.
\end{aligned}
$$

```
Clear[ecm2par]
ecm2par[log10p_, cg_, c1_, c2_] :=
Block[{step, p, a, b, w, wh},
p = (10^log10p)/cg;
step = .64;
a = 4.06;
b = 2.0;
W[a_, b_] :=
Block[{B1=N[p^(1/a), 30], B3=N[p^(b/a), 30]},
      N[(2*Sqrt[B3] + B1 - Sqrt[B3]*Log[B1]/Log[Sqrt[B3]]
         + c1 Sqrt[B3] + c2 (B3/Log[B3] - B1/Log[B1]))/ mue[a,b],30]
];
defined[al_, be_] := al<6 && al>=2 && be<5 && be>=1 && al-1>=be ;
w=W[a,b];
While[step >= 1/100,
If[defined[a-step,b] && (wh=W[a-step, b])<w, w=wh; a -= step,
If[defined[a+step,b-step] && (wh=W[a+step, b-step])<w, w=wh;  a += step; b -= step,
If[defined[a+step,b] && (wh=W[a+step, b])<w, w=wh; a += step,
If[defined[a+step,b+step] && (wh=W[a+step, b+step])<w, w=wh; a += step; b += step,
If[defined[a-step,b-step] && (wh=W[a-step, b-step])<w, w=wh; a -= step; b -= step,
If[defined[a,b+step] && (wh=W[a, b+step])<w, w=wh; b += step,
If[defined[a,b-step] && (wh=W[a, b-step])<w, w=wh; b -= step,
If[defined[a-step,b+step] && (wh=W[a-step, b+step])<w, w=wh; a -= step; b += step,
step =step/4]]]]]]]];
];
Return[{a, b, Round[p^(1/a)], Round[p^(b/a)], Round[1/mue[a,b]]}]
]
ecm2par::usage = "ecm2par[ log10p, cg, c1, c2 ] berechnet die
Parameter fuer den 2-Phasen-ECM-Algorithmus. log10p ist die Anzahl der
Dezimalstellen des zu suchenden Faktors, cg der Divisor vom
Kurvengenerator, c1 die Konstante fuer 1.Phase und c2 die Konstante
```

```
fuer die 2.Phase. Sinnvolle Werte sind z.B. c1=1, c2=0.066 fuer verb.
Std.Cont. bzw. c1=0, c2=1.6 fuer Std.Cont. Ausgabe: Liste {alpha, beta, B1,
B2, Anz.Kurven}. Achtung: Sinnvolle Werte nur bis ca. 45 Stellen!!!"
```

Anhang E

Listings

E.1 Hauptprogramme

E.1.1 ECM für einen Rechner

```
###########################################################################
#
# makefile zum Erzeugen von ecmp
#
# s. SunOS 4.0; Band 9; Kapitel 8; S.161
#
#  FDB 17.7.91
###########################################################################

# Name des fertigen Programms
PROGRAM= ../../RECHNEN/ecmp
# .c-Files aus denen es besteht
SOURCES= ecmp.c

# eigene Librarys Name mit Pfad fuer alle Versionen
ULIBS= ../LIBSOURCE/ECMP/elliptic.a ../LIBSOURCE/LARITH+/larith+.a
ULIBSDEB=../LIBSOURCE/ECMP/DEBUG/elliptic.a \
         ../LIBSOURCE/LARITH+/DEBUG/larith+.a
ULIBSPROF=../LIBSOURCE/ECMP/PROF/elliptic.a \
          ../LIBSOURCE/LARITH+/PROF/larith+.a
ULIBSGPROF=../LIBSOURCE/ECMP/GPROF/elliptic.a \
           ../LIBSOURCE/LARITH+/GPROF/larith+.a

# System-Librarys (gesamter Pfad)
SLIBS= ../LARITH/larith.a /usr/lib/libm.a
# Includefile-Directory
INCLUDE= ../INCLUDE

#-------------------------------------------------------------------------

CFLAGS= -O4 -I$(INCLUDE) -Bstatic
CPPFLAGS=
LDFLAGS=

OBJECTS= $(SOURCES:.c=.o)
DEBUG= $(PROGRAM:%=DEBUG/%)
PROF= $(PROGRAM:%=PROF/%)
GPROF= $(PROGRAM:%=GPROF/%)
VARIANTS.o= $(OBJECTS)
# Variantendirectory
ULIBSVAR = $(ULIBS)
VARIANTS=
```

```
.KEEP_STATE:
.INIT:
@-mkdir -p DEBUG PROF GPROF

all: $(PROGRAM)
debug: $(DEBUG)
prof: $(PROF)
gprof: $(GPROF)
variants: debug prof gprof

debug := CFLAGS= -g -I$(INCLUDE) -D$(MACHINE)
debug := ULIBSVAR= $(ULIBSDEB)
debug := VARIANTS= debug
prof := CFLAGS= -p -O -I$(INCLUDE) -D$(MACHINE)
prof := ULIBSVAR= $(ULIBSPROF)
prof := VARIANTS= prof
gprof := CFLAGS= -pg -O -I$(INCLUDE) -D$(MACHINE)
gprof := ULIBSVAR= $(ULIBSGPROF)
gprof := VARIANTS= gprof
$(DEBUG) := VARIANTS.o = $(OBJECTS:%=DEBUG/%)
$(PROF) := VARIANTS.o = $(OBJECTS:%=PROF/%)
$(GPROF) := VARIANTS.o = $(OBJECTS:%=GPROF/%)


$(PROGRAM) $(DEBUG) $(PROF) $(GPROF): $$(VARIANTS.o) $(ULIBS)
$(LINK.c) -o $@ $(VARIANTS.o) $(ULIBSVAR) $(SLIBS)

DEBUG/%.o PROF/%.o GPROF/%.o: %.c
$(COMPILE.c) -o $@ $<

$(ULIBS): FORCE
cd $(@D); $(MAKE) $(VARIANTS)

FORCE:

clean:
rm -f $(PROGRAM) DEBUG/$(PROGRAM) PROF/$(PROGRAM) GPROF/$(PROGRAM)\
      $(OBJECTS) DEBUG/$(OBJECTS) PROF/$(OBJECTS) GPROF/$(OBJECTS)
```

```
/* 007
   ECM elliptic-curve-method
   Folgende Eingaben werden von stdin angefordert:
        zu faktorisierende Zahl       (string)
        upper limit for trialdivision (ULONG)
        restlength  (int) // ist die Zahl, nach Division <= restlength (#Dezi-
                                  malstellen), dann fertig
        Info   (BOOLEAN)

   - benoetigt das File "ecm-strategy"

   Alle Ausgaben erfolgen auf stdout.


#include <time.h>
#include "allg.h"
#include <stdio.h>
#include "larith.h"
#include "ecmp-types.h"
#include "larith+.h"


main()
{
   ULONG              B, C, D, sim_curves, tfactor, trialdiv_limit,
                      job_buffer[MAXJOBS][2];
   LONGTYPE           n, nuebrig, factor, rest;
   int                randgenset, info, ldim, simultan, i, jobs, jobs_avail,
                      job_nr, job_ready, restlength, rmachine_number = 2;
   int                timer1[2], timer2[2], ecm_time = 0, stdc_time = 0,
                      total_time;
   Boolean            factor_found = FALSE, prim = FALSE;
   char               masterhost[20], n_string[MAXDEZSTEL], logfile[20];
   float              strategy[MAXSTRATEGY][2];
   Status             condition;
   PRIMETABLE        *ppt;
   Vector_of_points   P;
   Vector_of_curves   EC;

   ppt = initprimes(NULL);

   scanf("%s", n_string);
   ldim = (int) (2.302585093 * strlen(n_string) / 32) + 2;
   linit(ldim * 2);
   n = lalloc(ldim);
   nuebrig = lalloc(ldim);
   factor = lalloc(ldim);
   rest = lalloc(ldim);

   /* Eingabe */
   string_to_l(n_string, n);

   scanf("%lu", &trialdiv_limit);
   scanf("%d", &restlength);
   randgenset = 0;               /* randgenset=0: Startwert fuer Zufallszahl
                                    wird aus time_struct gewonnen; sonst :
                                    randgenset ist der Startwert */
   scanf("%d", &info);

   if (info)
   {
      printf("\nECM Version vom 2.5.92\n\n");
      printf("ldim: %d\n", ldim);
   }
   if (strlen(n_string) <= restlength)
   {
      printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
      lstelwrite(n);
      exit(0);
   }

   /* ---------- Probedivision ---------- */
   initltrialdiv(trialdiv_limit);
   if (info)
```

```
{
   printf("Probedivision bis %lu\n", trialdiv_limit);
   lstelwrite(n);
}

while ((tfactor = ltrialdiv(n)) > 1)
{
   if (info)
      printf("Faktor: %d\n", tfactor);
   else
      printf("%d\n", tfactor);
   lsdiv(n, tfactor, n, &i);
   factor_found = TRUE;
}
if (factor_found)
{
   if (info)
   {
      printf("uebrig: ");
      lstelwrite(n);
   }
   factor_found = FALSE;
}
lstring(n, n_string);
if (strlen(n_string) <= restlength)
{
   printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
   lstelwrite(n);
   exit(0);
}

/* ---------- einlesen der Strategie ---------- */
read_strategy(strategy);

if (info)
{
   printf("=========================================================\n\n");
   printf("zu faktorisieren durch ECM bis auf %d Dezimalstellen:\n",
          restlength);
   lstelwrite(n);
   printf("Generator:        %d\n", randgenset);
   printf("Anzahl Rechner:   %d\n\n", rmachine_number);
}
jobs = job_planing(rmachine_number, strategy, job_buffer, info);
jobs_avail = jobs;

if (info)
{
   printf("\n=========================================================\n\n");
   fflush(stdout);
}

ecgenset(randgenset);

prim = listprim(n);

start_timing(timer1);
get_timing(timer1);
start_timing(timer2);
get_timing(timer2);

while ((!prim) && (jobs_avail > 0) && !test1(n))
{
   sim_curves = job_buffer[job_nr][0];
   P = epvalloc(sim_curves, ldim);
   EC = ecvalloc(sim_curves, ldim);

   if (info)
      printf("%3d. Job in Arbeit - ", job_nr);
   fflush(stdout);

   if (!ecvgen(EC, P, n, factor))
   {
      B = job_buffer[job_nr][1];
      C = B * 10;
```

```
      D = B * 80;

      start_timing(timer1);
      condition = ecvfindfactor(B, C, n, P, ppt, factor);
      get_timing(timer1);
      ecm_time += timer1[0] / 100;
      if (info)
         printf("Ecm-Time: %s - ", strtime("hms", timer1[0] / 100));
      fflush(stdout);

      if (condition != 2)
      {
         if (condition == 1) /* no_factor_found */
         {
            start_timing(timer2);
            condition = ecm_stdcont2(P, B, D, n, ppt, factor);
            /* ecm_stdcont(P, D, n,ppt, factor); */
            get_timing(timer2);
            stdc_time += timer2[0] / 100;
            if (info)
               printf("Cont-Time: %s\n", strtime("hms", timer2[0] / 100));
            fflush(stdout);

            if (condition != 2)
               set1(factor);
         }
         else
            set1(factor);
      }
   }

   if (info)
   {
      printf("Noch %d Jobs uebrig.\n", jobs_avail);
      fflush(stdout);
   }

   if (test0(factor) || test1(factor))
      factor_found = FALSE;
   else
   {
      /* dividiere n durch den gefundenen Faktor, so oft es geht */
      lldiv(n, factor, nuebrig, rest);
      factor_found = FALSE;
      while (test0(rest) && llcomp(n, nuebrig) != 0)
      {
         factor_found = TRUE;
         if (info)
         {
            printf("Faktor: ");
            lstelwrite(factor);
         }
         else
            lwrite(factor);
         if (info)
         {
            printf("uebrig: ");
            lstelwrite(nuebrig);
         }
         llsto(nuebrig, n);  /* n= nuebrig */
         lldiv(n, factor, nuebrig, rest);
      }
   }
   lstring(n, n_string);
   if (strlen(n_string) <= restlength)
      jobs_avail = 0;
   ++job_nr;
   --jobs_avail;
   if (info)
      printf("job_nr %d\n", job_nr);
   prim = listprim(n);
}                             /* End while */

if (info)
```

```
   {
      printf("uebrig, %s : ", prim ? "weil Primzahl" : "aber keine Primzahl");
      lstelwrite(n);
   }
   else
   {
      lwrite(n);
      if (!prim)
         printf("uebrig, aber keine Primzahl\n");
   }

   total_time = ecm_time + stdc_time;
   if (info)
   {
      printf("Phase1-Ecm-Zeit : %s\n", strtime("hms", ecm_time));
      printf("Phase2-Cont-Zeit: %s\n", strtime("hms", stdc_time));
      printf("Gesamt-Zeit: %s\n", strtime("dhms", total_time));
   }

}
```

E.1.2 verteiltes ECM

Der Master

```
###########################################################################
#
#  makefile zum Erzeugen von ablauffaehigen Programmen
#
# s. SunOS 4.0; Band 9; Kapitel 8; S.161
#
#  FDB 17.7.91
###########################################################################

# Name des fertigen Programms
PROGRAM= ecmp-master
# .c-Files aus denen es besteht
SOURCES= ecmp-master.c

# eigene Librarys Name mit Pfad fuer alle Versionen
ULIBS= ../LIBSOURCE/ECMP/elliptic.a ../LIBSOURCE/LARITH+/larith+.a
ULIBSDEB= ../LIBSOURCE/ECMP/DEBUG/elliptic.a\
  ../LIBSOURCE/LARITH+/DEBUG/larith+.a
ULIBSPROF= ../LIBSOURCE/ECMP/PROF/elliptic.a\
   ../LIBSOURCE/LARITH+/PROF/larith+.a
ULIBSGPROF= ../LIBSOURCE/ECMP/GPROF/elliptic.a\
    ../LIBSOURCE/LARITH+/GPROF/larith+.a
# System-Librarys (gesamter Pfad)
SLIBS= ../LARITH/larith.a /home/denny/lib/liblips.a -lm
# Includefile-Directory
INCLUDE= ../INCLUDE
#----------------------------------------------------------------------------

CFLAGS= -I$(INCLUDE) -DPARALLEL
CPPFLAGS=
LDFLAGS=

OBJECTS= $(SOURCES:.c=.o)
DEBUG= $(PROGRAM:%=DEBUG/%)
PROF= $(PROGRAM:%=PROF/%)
GPROF= $(PROGRAM:%=GPROF/%)
VARIANTS.o= $(OBJECTS)
# Variantendirectory
ULIBSVAR = $(ULIBS)
VARIANTS=

.KEEP_STATE:
.INIT:
@-mkdir -p DEBUG PROF GPROF

all: $(PROGRAM)
debug: $(DEBUG)
prof: $(PROF)
gprof: $(GPROF)
variants: debug prof gprof

debug := CFLAGS= -g -I$(INCLUDE) -D$(MACHINE)
debug := ULIBSVAR= $(ULIBSDEB)
debug := VARIANTS= debug
prof := CFLAGS= -p -O -I$(INCLUDE) -D$(MACHINE)
prof := ULIBSVAR= $(ULIBSPROF)
prof := VARIANTS= prof
gprof := CFLAGS= -pg -O -I$(INCLUDE) -D$(MACHINE)
gprof := ULIBSVAR= $(ULIBSGPROF)
gprof := VARIANTS= gprof
$(DEBUG) := VARIANTS.o = $(OBJECTS:%=DEBUG/%)
$(PROF) := VARIANTS.o = $(OBJECTS:%=PROF/%)
$(GPROF) := VARIANTS.o = $(OBJECTS:%=GPROF/%)


$(PROGRAM) $(DEBUG) $(PROF) $(GPROF): $$(VARIANTS.o) $(ULIBS)
$(LINK.c) -o $@ $(VARIANTS.o) $(ULIBSVAR) $(SLIBS)
```

```
DEBUG/%.o PROF/%.o GPROF/%.o: %.c
$(COMPILE.c) -o $@ $<

$(ULIBS): FORCE
cd $(@D); $(MAKE) $(VARIANTS) PAR=-DPARALLEL

FORCE:

clean:
rm -f $(PROGRAM) DEBUG/$(PROGRAM) PROF/$(PROGRAM) GPROF/$(PROGRAM)\
      $(OBJECTS) DEBUG/$(OBJECTS) PROF/$(OBJECTS) GPROF/$(OBJECTS)
```

```
/* 005 Version mit kompletter Zeitmessung in MIPS und Rechner-Gruppierungen
   ecmp-master: startet Rechner remote (ecmp_worker) und verwaltet die von
   diesen Rechnern gefundenen Ergebnisse.

   Folgende Eingaben werden von stdin angefordert:
zu faktorisierende Zahl       (string)
upper limit for trialdivision (ULONG)
restlength  (int)  // ist die Zahl, nach Division <= restlength (#Dezi-
                              malstellen), dann fertig
Statistik                     (BOOLEAN)
Logfile     (Boolean) // wenn true, dann wird die Ausgabe des workers
                         in ein Logfile geschrieben, sonst nicht

   benoetigt das File "ecm-strategy"

   Alle Ausgaben erfolgen auf stdout.
   22.5.92
   3.11.92 neues LIPS eingebunden!
   7.11.92 neue Parameterwahl*/

#include "allg.h"
#include <stdio.h>
#include <string.h>
#include "larith.h"
#include "ecmp-types.h"
#include "larith+.h"
#include "/home/thsetz/lips_2.1Beta/include/lips.h"

#define VERSION "ECMP-Master mit Zeitmessung in MIPS vom 7.11.92"
#define HOSTNAMELEN 30
#define MAXMACHINES 200

Boolean           test_factor();

main()
{
   ULONG             B, C, tfactor, trialdiv_limit, job_buffer[MAXJOBS][3];
   LONGTYPE          n, factor, nuebrig, rest;
   int               randgenset, ldim, simultan, i, j, jobs, jobs_avail,
                     job_nr, job_ready, restlength, phase, working_comp,
                     time,
                     totaltime = 0, /* Gesamtzeit fuer alle Rechner */
                     gestartet, rmachine_status[MAXMACHINES];
   Boolean           statistik, factor_found = FALSE, prim = FALSE, logfiling;
   char              symbol[6], n_string[MAXDEZSTEL], logfile[HOSTNAMELEN];


   float             strategy[MAXSTRATEGY][2];

   printf("\n%s\n", VERSION);

   /* Initialisierung der Netzwerkfunktionen */
   /* ====================================== */
   printf("initlips: %d\n", initlips("CLIENT_CONF", NULL, NULL));

   /* Einlesen der Eingabe */
   /* ==================== */
   scanf("%s", n_string);
   ldim = (int) (2.302585093 * strlen(n_string) / 32) + 2;
   printf("ldim: %d\n", ldim);
```

```
   linit(ldim);
   n = lalloc(ldim);
   nuebrig = lalloc(ldim);
   factor = lalloc(ldim);
   rest = lalloc(ldim);

   /* Eingabe */
   string_to_l(n_string, n);

   scanf("%lu", &trialdiv_limit);
   scanf("%d", &restlength);
/*   scanf("%d", &randgenset);        Kurvenstart */
   scanf("%d", &statistik);
   scanf("%d", &logfiling);

   if (strlen(n_string) <= restlength)
   {
      printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
      lstelwrite(n);
      exit(0);
   }

   system("date");

   /* ---------- Probedivision ---------- */
   initltrialdiv(trialdiv_limit);
   printf("Probedivision bis %lu\n", trialdiv_limit);
   fflush(stdout);
   lstelwrite(n);
   while ((tfactor = ltrialdiv(n)) > 1)
   {
      printf("Faktor: %d\n", tfactor);
      lsdiv(n, tfactor, n, &i);
      factor_found = TRUE;
   }
   if (factor_found)
   {
      printf("uebrig: ");
      lstelwrite(n);
      factor_found = FALSE;
   }
   lstring(n, n_string);
   if (strlen(n_string) <= restlength)
   {
      printf("Anzahl der Dezimalstellen <= %d!\n", restlength);
      lstelwrite(n);
      exit(0);
   }
   fflush(stdout);

   /* ---------- einlesen der Strategie ---------- */
   printf("einlesen der Strategy\n");
   fflush(stdout);
   read_strategy(strategy);

   /* ----- (1) Aufteilen der gesamten Arbeit in Jobs ----- */
   printf("jobplaning\n");
   fflush(stdout);

   if (statistik)
   {
      printf("===========================================================\n\n");
      printf("zu faktorisieren durch ECM bis auf %d Dezimalstellen:\n", restlength);
      lstelwrite(n);
      printf("Generator:       %d\n", randgenset);
      printf("Anzahl Rechner:   %d\n", rmachine_number);
      printf("Strategie:\n");

      jobs = job_planing(rmachine_number, strategy, job_buffer);
      jobs_avail = jobs;

      printf("\n=========================================================\n\n");
      fflush(stdout);
   }
```

```
  prim = listprim(n);

  if (!prim)
  {
     /* --------------- Starten der Worker -------------------- */
     if (rmachine_number < 1)
     {
printf("Keine Remote-Maschinen zum Rechnen da!!!!!\n");
exit(1);
     }
     printf("rstart von:\n");
     for (i = 0; i < rmachine_number; i++)
     {
printf("%20s: ", machine[i].name);
rmachine_status[i] = start_machine(i, logfiling);
printf("%d\n", rmachine_status[i]);
fflush(stdout);
     }
  }
  while ((!prim) && (jobs_avail > 0))
  {

     /* ----- verteile die Aufgaben an alle remote Maschinen ------- */

     lstring(n, n_string);
     job_nr = 0;

     for (i = 0; i < rmachine_number; i++)
     {
if (rmachine_status[i] == 0)
{
   while (job_buffer[job_nr][0] == 0)
      job_nr = (job_nr + 1) % jobs;

   printf("%3d. Job an %s\n", job_nr, machine[i].name);
   fflush(stdout);
   sprintf(symbol, "%d", i);
   if (out(machine[i].name, symbol, "isllili", lstellen(n) + 2,
              n_string, job_buffer[job_nr][1], job_buffer[job_nr][2],
              randgenset, job_buffer[job_nr][0], job_nr))
      l_error("Fehler bei OUT von START : ");
   else
      job_nr = (job_nr + 1) % jobs;
}
     }
     fflush(stdout);

     do
     {
/* -------- warte auf Ergebnisse und lese sie ein ---------- */
while (1)
{
   if (fexist(STOP_ECM))
   {
      unlink(STOP_ECM);
      printf("\nECM MUTWILLIG GESTOPPT!!!!!!!!!\n");
      fflush(stdout);
      goto ENDE;
   }
   fflush(stdout);

   i = inp("result", "SSIII", n_string, symbol, &job_ready,
   &time, &phase);
   fflush(stdout);
   if (i == 0)
      break;
   else
      sleep(10);
}

if (delete_job(job_ready, job_buffer, atoi(symbol)) == 0)
{
   totaltime += (time / 100);
}
```

```
 else
    continue;

 system("date");
 printf("%s: Job %d ist fertig! Phase: %d  Erg: %s\n",
                machine[atoi(symbol)].name, job_ready, phase, n_string);
 printf("Zeit in Mips: %s - ", strtime("hms", time / 100));
 printf("Gesamtzeit in Mips: %s\n", strtime("ydhms", totaltime));
 fflush(stdout);

 string_to_l(n_string, factor);
 factor_found = test_factor(factor, n);

 if (job_ready == -1 || (!factor_found && !test1(factor)))
    continue;

 --jobs_avail;
 printf("Noch %d Jobs uebrig.\n", jobs_avail);
 fflush(stdout);

 lstring(n, n_string);
 if (strlen(n_string) <= restlength)
    jobs_avail = 0;

 if (!factor_found && jobs_avail > 0)
 {
    while (job_buffer[job_nr][0] == 0)
       job_nr = (job_nr + 1) % jobs;
    printf("%3d. Job an %s\n\n", job_nr, machine[atoi(symbol)].name);
    fflush(stdout);

    if (out(machine[atoi(symbol)].name, symbol, "isllili",
                lstellen(n) + 2, n_string, job_buffer[job_nr][1],
                job_buffer[job_nr][2], randgenset, job_buffer[job_nr][0],
                job_nr))
       l_error("Fehler bei out in Hauptschleife: ");
    else
       job_nr = (job_nr + 1) % jobs;
 }
      }
      while (!factor_found && jobs_avail > 0);

      printf("\n-----------------------------------------------------\n\n");
      fflush(stdout);

      lstring(n, n_string);
      prim = listprim(n);
      factor_found = FALSE;
   } /* End while */

ENDE:
   printf("uebrig, %s : ", prim ? "weil Primzahl" : "aber keine Primzahl");
   lstelwrite(n);

   kill_all_clients();

   /* lese noch einmal alle Rechenzeiten ein */
   system("date");
   j = 0; /* zaehlt Anzahl sleeps */

   /* zaehle  Rechner, die arbeiten */
   working_comp = 0;
   for (i = 0; i < rmachine_number; i++)
      if (rmachine_status[i] == 0)
 ++working_comp;

   do
   {
      i = inp("result", "SSIII", n_string, symbol, &job_ready, &time,
      &phase);
      if (i == 0)
      {
 working_comp--;
 totaltime += (time / 100);
```

```
 printf("%s: Phase: %d  Erg: %s   ", machine[atoi(symbol)].name,
               phase, n_string);
 printf("Zeit in Mips: %s\n", strtime("hms", time / 100));
 fflush(stdout);
      }
      else
      {
 j++;
/*  printf("j= %d ", j); fflush(stdout);*/
 sleep(10);
      }
   }
   while (j < 30 && working_comp > 0);

   printf("Gesamtzeit in Mips: %s\n", strtime("ydhms", totaltime));

   exitlips();
   printf("ENDE\n");
}


/* loescht einen Job aus dem Job-Buffer
   Ergebnis:
   0, falls alles OK
   1, falls Job-Verdopplung */

delete_job(job, job_buffer, hostindex)
   int              job, hostindex;
   ULONG            job_buffer[MAXJOBS][3];
{
   int              rw;

   if (job != -1) /* wenn job=-1 dann nicht loeschen, da nicht
   zu ende gerechnet */
   {
      if (job_buffer[job][0] != 0)
      {
 job_buffer[job][0] = 0;/* loesche diesen Job aus Buffer */
 job_buffer[job][1] = hostindex;
 rw = 0;
      }
      else
      {
 if (job_buffer[job][1] == hostindex)
 {
    printf("\nACHTUNG: Jobverdopplung ist aufgetreten - ignoriert!\n\n");
    rw = 1;
 }
 rw = 0;
      }
   }
   else
      rw = 0;

   return (rw);
}

/* test_factor(factor, n) testet ob factor ein Faktor von n ist.
   Wenn ja, wird n so oft durch factor geteilt wie moeglich und factor dabei
   ausgegeben; Ergebnis ist factor_found
   Wenn nein wird no_factor_found zurueckgeliefert */
Boolean          test_factor(factor, n)
   LONGTYPE         factor, n;
{
   LONGTYPE         nuebrig, rest;
   int              ldim;
   Boolean          rw;

   ldim = lstellen(n) + 2;

   nuebrig = lalloc(ldim);
   rest = lalloc(ldim);

   if (test0(factor) || test1(factor))
      rw = FALSE;
```

```
   else
   {
      /* dividiere n durch den gefundenen Faktor, so oft es geht */
      lldiv(n, factor, nuebrig, rest);
      rw = FALSE;
      while (test0(rest) && llcomp(n, nuebrig) != 0)
      {
 printf("Faktor: ");
 lstelwrite(factor);
 printf("uebrig: ");
 lstelwrite(nuebrig);
 rw = TRUE;
 llsto(nuebrig, n); /* n= nuebrig */
 lldiv(n, factor, nuebrig, rest);
      }
   }
   free(nuebrig);
   free(rest);
   return (rw);
}

start_machine(i, logfiling)
   int               i, logfiling;
{
   char              logfile[HOSTNAMELEN], symbol[6];

   if (logfiling) /* sollen die worker ein Logfile anlegen? */
      sprintf(logfile, "LOGFILE.%s", machine[i].name);
   else
      sprintf(logfile, "/dev/null");

   if (rstart("ecmp-worker", machine[i].name, logfile) == -1)
   {
      l_error("FEHLER bei rstart! ");
      return (-1);
   }
   else
   {
      sprintf(symbol, "%d", i);
      if (out(machine[i].name, "start", "s", symbol) == -1)
      {
 l_error("FEHLER bei Symbol-out! ");
 return (-2);
      }
      else
 return (0);
   }
}
```

Der Worker

```
########################################################################
#
#  makefile zum Erzeugen von ablauffaehigen Programmen
#
# s. SunOS 4.0; Band 9; Kapitel 8; S.161
#
#  FDB 17.7.91
########################################################################

# Name des fertigen Programms
PROGRAM= ecmp-worker
# .c-Files aus denen es besteht
SOURCES= ecmp-worker.c

# eigene Librarys Name mit Pfad fuer alle Versionen
ULIBS= ../LIBSOURCE/ECMP/elliptic.a ../LIBSOURCE/LARITH+/larith+.a
ULIBSDEB= ../LIBSOURCE/ECMP/DEBUG/elliptic.a
    ../LIBSOURCE/LARITH+/DEBUG/larith+.a
ULIBSPROF= ../LIBSOURCE/ECMP/PROF/elliptic.a
   ../LIBSOURCE/LARITH+/PROF/larith+.a
ULIBSGPROF= ../LIBSOURCE/ECMP/GPROF/elliptic.a
    ../LIBSOURCE/LARITH+/GPROF/larith+.a
# System-Librarys (gesamter Pfad)
SLIBS= ../LARITH/larith.a /home/denny/lib/liblips.a -lm
# Includefile-Directory
INCLUDE= ../INCLUDE

#-------------------------------------------------------------------------

CFLAGS=  -DPARALLEL -I$(INCLUDE) -Bstatic
CPPFLAGS=
LDFLAGS=

OBJECTS= $(SOURCES:.c=.o)
DEBUG= $(PROGRAM:%=DEBUG/%)
PROF= $(PROGRAM:%=PROF/%)
GPROF= $(PROGRAM:%=GPROF/%)
VARIANTS.o= $(OBJECTS)
# Variantendirectory
ULIBSVAR = $(ULIBS)
VARIANTS=

.KEEP_STATE:
.INIT:
@-mkdir -p DEBUG PROF GPROF

all: $(PROGRAM)
debug: $(DEBUG)
prof: $(PROF)
gprof: $(GPROF)
variants: debug prof gprof

debug := CFLAGS= -g -I$(INCLUDE) -D$(MACHINE)
debug := ULIBSVAR= $(ULIBSDEB)
debug := VARIANTS= debug
prof := CFLAGS= -p -O -I$(INCLUDE) -D$(MACHINE)
prof := ULIBSVAR= $(ULIBSPROF)
prof := VARIANTS= prof
gprof := CFLAGS= -pg -O -I$(INCLUDE) -D$(MACHINE)
gprof := ULIBSVAR= $(ULIBSGPROF)
gprof := VARIANTS= gprof
$(DEBUG) := VARIANTS.o = $(OBJECTS:%=DEBUG/%)
$(PROF) := VARIANTS.o = $(OBJECTS:%=PROF/%)
$(GPROF) := VARIANTS.o = $(OBJECTS:%=GPROF/%)


$(PROGRAM) $(DEBUG) $(PROF) $(GPROF): $$(VARIANTS.o) $(ULIBS)
$(LINK.c) -o $@ $(VARIANTS.o) $(ULIBSVAR) $(SLIBS)

DEBUG/%.o PROF/%.o GPROF/%.o: %.c
$(COMPILE.c) -o $@ $<

$(ULIBS): FORCE
```

```
cd $(@D); $(MAKE) $(VARIANTS) PAR=-DPARALLEL

FORCE:

clean:
rm -f $(PROGRAM) DEBUG/$(PROGRAM) PROF/$(PROGRAM) GPROF/$(PROGRAM)\
      $(OBJECTS) DEBUG/$(OBJECTS) PROF/$(OBJECTS) GPROF/$(OBJECTS)
```

```
/* V003 - Version ohne Interrupt und Zeitmessung in MIPS, restliche Zeit
   wird nach kill zum Master geschickt.
   FDB 24.5.92
   6.6.92  neue STDCONT und komprimierte PRIMZAHLTAFEL
   15.6.92 Interrupt wieder ausgebaut, ecvfindfactor liefert new_message, wenn
           eine da ist
   30.10.92 Neue Lips-Version
*/

#include <stdio.h>
#include <string.h>
#include "ecmp-types.h"
#include "larith.h"
#include "larith+.h"
#include "/home/thsetz/lips_2.1Beta/include/lips.h"
#include "allg.h"

#define VERSION "ECMP-WORKER ohne Interrupt und Zeitmessung in MIPS vom 30.10.92"

/* globale Variablen */
int               g_ldim, g_job_nr, job_nr, g_randgenset, g_sim_curves,
                  timer[2], phase;
char              g_n_string[MAXDEZSTEL], myname[HOSTNAMELEN], symbol[6];
ULONG             g_B, g_D;
LONGTYPE          n; /* zu faktorisierende Zahl */

main()
{
   int               onintr(); /* Funktion wird bei Interrupt, ausgeloest
   durch alarm, aufgerufen */
   int               send_time(); /* Funktion wird durch kill vom
   Master aktiviert */

   ULONG             B, C, D, sim_curves;
   LONGTYPE          faktor;
   Status            condition;
   int               randgenset, ldim;
   char              n_string[MAXDEZSTEL];
   PRIMETABLE       *ppt;
   Vector_of_points  P;
   Vector_of_curves  EC;

   linit(20);
   ppt = initprimes(NULL);

   printf("%s\n", VERSION);
   printf("initlips: %d\n", initlips(NULL, send_time, NULL));
   gethostname(myname, HOSTNAMELEN);
   fflush(stdout);

   /* einlesen des Symbolische Namens z.Zt Index in machine struct */
   while (in("start", "S", symbol) != 0)
   {
      l_error("FEHLER bei Symbol einlesen! ");
      sleep(60);
   }
   printf("Symbolischer Name: %s\n", symbol);
   fflush(stdout);

   condition = no_factor_found;
   do
   {
      if (condition != new_message)
      {
```

```
if (in(symbol, "ISLLILI", &g_ldim, g_n_string, &g_B, &g_D,
               &g_randgenset, &g_sim_curves, &g_job_nr) != 0)
{
   l_error("Fehler bei in: ");
   fflush(stdout);
}
     }

     phase = 0;

     ldim = g_ldim;
     strcpy(n_string, g_n_string);
     B = g_B;
     D = g_D;
     C = D;
     randgenset = g_randgenset;
     sim_curves = g_sim_curves;
     job_nr = g_job_nr;

     printf("\n");
     system("date");
     printf("Job %d erhalten. Kurven: %lu   B1: %lu\n   B3: %lu", job_nr,
            sim_curves, B, D);
     printf("   noch %d Stellen zu faktorisieren\n", strlen(n_string));
     fflush(stdout);

     n = lalloc(ldim);
     faktor = lalloc(ldim);
     P = epvalloc(sim_curves, ldim);
     EC = ecvalloc(sim_curves, ldim);

     string_to_l(n_string, n); /* n soll faktorisiert werden */

     ecgenset(randgenset);

     timer[0] = timer[1] = 0;
     start_timing(timer); /*---------- Starte Zeitmessung ----------*/

     condition = factor_found; /* andernfalls wird condition geaendert */
     if (!ecvgen(EC, P, n, faktor))
     {
phase = 1;
condition = ecvfindfactor(B, C, n, P, ppt, faktor);
if (condition == no_factor_found)
{
   get_timing(timer);
   printf("Rechenzeit: %s\n", strtime("hms", timer[0] / 100));
   cont_timing(timer);

   system("date");
   printf("benutze jetzt ECM-Stdcont bis %ld!\n", D);
   fflush(stdout);
   phase = 2;
   condition = ecm_stdcont2(P, B, D, n, ppt, faktor);
   if (condition != factor_found)
      set1(faktor);
}
     }

     get_timing(timer); /*---------- Ende der Zeitmessung ----------*/

     if (condition == new_message)
     {
job_nr = -1;
set1(faktor);
     }

     printf("gesamte Rechenzeit: %s\n", strtime("hms", timer[0] / 100));

     lstring(faktor, n_string);/* faktor zum Verschicken in string umwandeln */
     system("date");
     printf("%s: Faktor: %s\n", myname, n_string);
     fflush(stdout);
     if (out(machine[0].name, "result", "ssiii", n_string, symbol, job_nr,
```

```
          (int) (timer[0] * mips()), phase))
 l_error("Fehler bei out:");
      printf("gesendet an %s!\n\n", machine[0].name);
      fflush(stdout);

      free(n);
      free(faktor);
      ecvfree(EC);
      epvfree(P);
   }
   while (1);
}


int               send_time()
{
   printf("\nKILL: Master killt mich!\n");
   get_timing(timer); /*---------- Ende der Zeitmessung ----------*/
   printf("Rechenzeit bis KILL: %s\n", strtime("hms", timer[0] / 100));
   fflush(stdout);

   out(machine[0].name, "result", "ssiii", "1", symbol, -1,
       (int) (timer[0] * mips()), phase);
   /* Job_nr = -1, weil Job nicht zuende gerechnet wurde */

   printf("Rechenzeit an %s gesendet\n\n", machine[0].name);
   printf("Auf Wiedersehen!!\n");
   fflush(stdout);
   return;
}
```

E.1.3 Erzeugen der Primzahltafel

```
/* Programm zur Erzeugung von Primzahltafeln Franz-Dieter Berger 6.6.92
   compilieren mit 'cc -o sieb -O sieb.c' */

#include <stdio.h>

char              filename[] = "KOMP-PRIMES";

void main()
{
   unsigned int      pmax;
   char             *table;
   unsigned char     dif;
   unsigned int      i, j, p, z, maxdif, palt;
   int               komp;
   FILE             *fp;

   fprintf(stderr, "Dieses Programm erzeugt eine Primzahltafel.\n");
   fprintf(stderr, "Diese kann auch komprimiert werden, dann wird nur die nur\
           die Differenz\n");
   fprintf(stderr, "zweier aufeinanderfolgender Primzahlen als ein char\
           gespeichert.\n");
   fprintf(stderr, "In diesem Fall werden die Werte in das File 'KOMP-PRIMES'\
           geschrieben, \n");
   fprintf(stderr, "andernfalls auf stdout ausgegeben. Dieser Text wird auf\
           stderr ausgegeben, \n");
   fprintf(stderr, "damit die Ausgabeumleitung mit > funktioniert.\n\n");
   fprintf(stderr, "Die Eingaben werden nicht ueberprueft!!!!!\n\n\n");
   fprintf(stderr, "Obere Schranke fuer Primzahlen (max. 3*10^7): ");
   scanf("%u", &pmax);
   fprintf(stderr, "komprimiert abspeichern? (0=Nein, 1=Ja): ");
   scanf("%d", &komp);
   if (komp)
   {
      if ((fp = fopen(filename, "w")) == NULL)
      {
```

```
      printf("FEHLER: Ich kann das File %s nicht anlegen!\n", filename);
      exit(1);
       }
   }
   table = (char *) calloc(pmax / 2, sizeof(char));
   if (table == NULL)
   {
      fprintf(stderr, "Speicher voll ");
      exit(1);
   }
   for (i = 0; i <= pmax / 2; i++)
   {
      p = i * 2 + 3;
      for (z = 3 * p; z < pmax; z += 2 * p)
     table[(z - 3) / 2] = 1;
   }

   j = 1;
   if (!komp)
   {
      printf("%d\n", 2);
      for (i = 0; i < (pmax - 3) / 2; i++)
     if (!table[i])
     {
        p = 2 * i + 3;
        printf("%u\n", p);
        j++;
     }
   }
   else
   {
      putc(2, fp);
      palt = 2;
      for (i = 0; i < (pmax - 3) / 2; i++)
     if (!table[i])
     {
        p = 2 * i + 3;
        dif = (unsigned char) (p - palt);
        putc(dif, fp);
        palt = p;
        j++;
     }
      fclose(fp);
   }
   fprintf(stderr, "Anzahl der Primzahlen : %u\n", j);
}
```

E.2 Librarys

E.2.1 elliptic.a

```
##############################################################################
#                                                                            #
#     Makefile fuer Librarys
#     11.7.91                                                                #
#     (entspricht im wesentlichen dem makefile aus SunOS 4.0 Band 9 S.163) #
#     moeglich ist: make debug, make prof, make gprof, make all variants   #
#                                                                            #
##############################################################################

# Hier die *.c-Files eintragen
SOURCES= ecalloc.c ecfree.c ecwrite.c ecgen.c \
 ecvalloc.c ecvfree.c ecvgen.c ecvfindfactor.c \
 epalloc.c epfree.c epwrite.c epsto.c epcomp.c epnegcomp.c eponcurve.c\
 epvalloc.c epvfree.c epvlength.c epvactiv_curves.c epvadd.c epvmul.c\
 ecm_parameters.c ecm_stdcont.c ecm_stdcont2.c read_strategy.c \
         job_planing.c new_job.c

# Hier den Namen der Library, die erzeugt werden soll eintragen (ohne Pfad)
LIBRARY= elliptic.a

#-------------------------------------------------------------------------------

CFLAGS = -O4 -I$(INCLUDE) $(PAR) -Bstatic
#CFLAGS = -O4 -I$(INCLUDE) -Bstatic
CPPFLAGS =
#OUTPUT_OPTION = -o $@

MEMBERS= $(SOURCES:.c=.o)
DEBUG= $(LIBRARY:%=DEBUG/%)
PROF= $(LIBRARY:%=PROF/%)
GPROF= $(LIBRARY:%=GPROF/%)
VARIANTS.o= $(MEMBERS)

INCLUDE= $(HOME)/DIPLOM/INCLUDE

.KEEP_STATE:
.PRECIOUS: $(LIBRARY)
.INIT:
@-mkdir -p DEBUG PROF GPROF

all: $(LIBRARY)
debug: $(DEBUG)
prof: $(PROF)
gprof: $(GPROF)
variants: debug prof gprof

debug := CFLAGS= -g -I$(INCLUDE) $(PAR)
prof := CFLAGS= -p -O -I$(INCLUDE) $(PAR)
gprof := CFLAGS= -pg -O -I$(INCLUDE) $(PAR)
$(DEBUG) := VARIANTS.o = $(MEMBERS:%=DEBUG/%)
$(PROF) := VARIANTS.o = $(MEMBERS:%=PROF/%)
$(GPROF) := VARIANTS.o = $(MEMBERS:%=GPROF/%)


$(LIBRARY) $(DEBUG) $(PROF) $(GPROF): $$(VARIANTS.o)
ar rv $@ $?
ranlib $@

$(LIBRARY)(%.o): %.o
$(DEBUG)(DEBUG/%.o): DEBUG/%.o
$(PROF)(PROF/%.o): PROF/%.o
$(GPROF)(GPROF/%.o): GPROF/%.o
DEBUG/%.o PROF/%.o GPROF/%.o: %.c
$(COMPILE.c) -o $@ $<

clean:
rm -rf $(LIBRARY) $(MEMBERS) DEBUG PROF GPROF
```

```
/* 001
   10.10.91
   ecalloc (elliptic curve allocate)
   allocates memory for a elliptic curve

   Def: Curve ecalloc(size);
        unsigned int size;          // size of LONGTYPE
   Res: Pointer on the allocated memory

   Bugs:
   If the allocation fails, the function lalloc from larith breaks with
   FATAL_ERROR. */

#include <malloc.h>
#include "ecmp-types.h"

Curve ecalloc(size)
   unsigned int      size;
{
   Curve             C;

   C = (Curve) malloc(sizeof(struct CURVE));
   C->a = lalloc(size + 1);
   C->b = lalloc(size + 1);
   C->activ = FALSE;

   return (C);
}
```

```
/* 001
   10.10.91
   ecfree (elliptic curve free)
   releases the memory which was allocated by ecalloc

   Def: void ecfree(C);
        Curve C;
   Res: none
*/
#include <malloc.h>
#include "ecmp-types.h"

void ecfree(C)
   Curve             C;
{
   free(C->a);
   free(C->b);
   free(C);
}
```

```
/* 001
   10.10.91
   ecwrite (elliptic curve write)
   writes the parameter of an elliptic curve (if the curve is activ) in the
   following form on the screen:
   (E: y^2 = x^3 + ax + b)
      a
      b
   if the curve is not activ, then it prints the message "Curve is not activ!".

   Def: void ecwrite(C);
        Curve E;
   Res: none;  */

#include <stdio.h>
#include "larith.h"
#include "ecmp-types.h"

void ecwrite(C)
   Curve             C;
{
```

```
   if (C == NULL)
      printf("no curve\n");
   else
   {
      if (C->activ)
      {
         lwrite(C->a);
         lwrite(C->b);
      }
      else
         printf("Curve is not activ!\n");
   }
   return;
}




/* 001
   11.10.91
   ecgen (elliptic curve generator, Montgomery S.262)
   ecgen(E, P, n, factor) generiert eine elliptische Kurve E, deren Diskri-
   minate prim zu n ist, und einen Punkt P, der auf der Kurve liegt. Wird dabei
   schon ein Faktor von n gefunden, so wird er mit factor zurueckgeliefert.

   Def: Boolean ecgen(C, P, n, factor)
        Curve    E;         (* wird erzeugt *)
        Point    P;         (* wird erzeugt *)
        LONGTYPE n,         (* die zu faktorisierende Zahl *)
        factor;    (* Faktor von n, falls zufaellig einer gefunden wird *)
   Res: TRUE, falls faktor gefunden;
        FALSE, sonst */

#include 'ecmp-types.h'
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>

Boolean           is_curve();

Boolean ecgen(E, P, n, factor)
   Curve            E;
   Point            P;
   LONGTYPE         n, factor;
{
   int              LDIM;
   long             r;
   LONGTYPE         s1, s2, s3;
   LONGTYPE         a1, a2, a3, h1;    /* Hilfsvariablen */
   Boolean          faktor_gefunden = FALSE;

   LDIM = lstellen(n) + 2;
   lwrite(n);*/
   s1 = lalloc(LDIM);
   s2 = lalloc(LDIM);
   s3 = lalloc(LDIM);
   a1 = lalloc(LDIM);
   a2 = lalloc(LDIM);
   a3 = lalloc(LDIM);
   h1 = lalloc(LDIM);

   do
   {
      do
      {
         /* Erzeuge 3 Zufallszahlen */
         r = random();
         slsto(r, s1);
         r = random();
         slsto(r, s2);
         r = random();
         slsto(r, s3);

         llmulmod(s1, s2, n, P->y);     /* berechne P->y */
```

```
         llmulmod(P->y, s3, n, P->y);

         llmulmod(s1, s1, n, s1);
         llmulmod(s2, s2, n, s2);
         llmulmod(s3, s3, n, s3);
      }
      while ((llcomp(s1, s2) == 0) || (llcomp(s1, s3) == 0) ||
             (llcomp(s2, s3) == 0));

      /* berechne P->x = (s1+s2+s3)/3 */
      slsto(3, h1);
      llgcd1(h1, n, factor, h1);/* h1 = 1/3 mod n */

      if (lscomp(factor, 1) != 0)
      {
         faktor_gefunden = TRUE;
      }
      else
      {
         lladdmod(s1, s2, n, P->x);
         lladdmod(P->x, s3, n, P->x);
         llmulmod(P->x, h1, n, P->x);

         llsubmod(s1, P->x, n, a1);     /* Hilfsvariablen belegen */
         llsubmod(s2, P->x, n, a2);
         llsubmod(s3, P->x, n, a3);

         /* Berechnung von E->a */
         llmulmod(a1, a2, n, E->a);
         llmulmod(a1, a3, n, h1);
         lladdmod(h1, E->a, n, E->a);
         llmulmod(a2, a3, n, h1);
         lladdmod(h1, E->a, n, E->a);

         /* Berechnung von E->b */
         llmulmod(a1, a2, n, E->b);
         llmulmod(E->b, a3, n, E->b);
      }
       lwrite(n);*/
   }
   while (!is_curve(E, n) && !faktor_gefunden);
   free(s1);
   free(s2);
   free(s3);
   free(a1);
   free(a2);
   free(a3);
   free(h1);
   P->C = E;                    /* P zeigt auf E */
   epunset0(P);
   E->activ = TRUE;
   return (faktor_gefunden);
}

/* setzt den Zufallszahlengenerator auf einen Startwert
   ist a=0, so wird als Startwert die Millisek aus dem struct timeb benutzt
   was ausreichend zufaellig sein duerfte;
   sonst wird a direkt eingesetzt */

void ecgenset(a)
   int              a;
{
   int              h;
   struct timeb     tp;         /* Zeiger auf Zeitstruktur fuer Zufallswert */

   if (a == 0 && (ftime(&tp) != -1))
   {
      h = ((int) tp.millitm) * time();
      srandom(h);
   }
   else
      srandom(a);
}
```

```
Boolean          is_curve(E, n)/* prueft ob die Diskriminante != 0 ist */
   Curve            E;        /* ist das der Fall, so ist die Kurve eine
                                 elliptische */
   LONGTYPE         n;
{
   int              LDIM;
   Boolean          rwert;
   LONGTYPE         h0, h1;

   LDIM = lstellen(n) * 2 + 2;
   h0 = lalloc(LDIM);
   h1 = lalloc(LDIM);
   llsto(E->a, h0);             /* berechne Diskriminante 4a^3+27b^2 */
   llmulmod(h0, h0, n, h0);     /* h0 = a^2 */
   llmulmod(h0, E->a, n, h0);   /* h0 = a^3 */
   lmul2mod(h0, n);
   lmul2mod(h0, n);             /* h0 = 4a^3 */

   llsto(E->b, h1);             /* h1 = b */
   lsquare(h1, h1);             /* h1 = b^2 */
   lsmul(h1, 27, h1);           /* h1 = 27 b^2 */
   lladd(h0, h1, h0);           /* h0 = 4a^3+27b^2 */
   lldiv(h0, n, 0, h0);         /* h0 = h0 mod n */

   if (lscomp(h0, 0) == 0)      /* h0 == 0 ? */
      rwert = FALSE;
   else
      rwert = TRUE;
   free(h0);
   free(h1);
   return (rwert);
}
```

```
 /* 001 10.10.91 ecvalloc (elliptic curve vector allocate) Allocates memory
    for a vector of curves. The last vectorelement is a NULL pointer.

 Def: Vector_of_curves ecvalloc(quantity, size); unsigned int quantity,
    // quantity of curves size;      // size of LONGTYPE Res: pointer on the
    allocated memory

 Bugs: If the allocation fails, the function lalloc from larith breaks with
    FATAL_ERROR. */

#include <stdio.h>
#include <malloc.h>
#include "ecmp-types.h"

Vector_of_curves  ecvalloc(quantity, size)
   unsigned int      quantity, size;
{
   int               j;
   Vector_of_curves  C;

   C = (Vector_of_curves) calloc(quantity + 1, sizeof(Curve));

   for (j = 0; j < quantity; j++)
      C[j] = ecalloc(size);
   C[quantity] = NULL;

   return (C);
}
```

```
/* 001
   10.10.91
   ecvfree (elliptic curve vector free)
   releases the memory which was allocated by ecvalloc
```

```
   Def: void ecvfree(C);
        Vector_of_curves C;
   Res: none */

#include <stdio.h>
#include <malloc.h>
#include "ecmp-types.h"

void ecvfree(C)
   Vector_of_curves  C;
{
   int               i;

   for (i = 0; C[i] != NULL; i++)
      ecfree(C[i]);
   free(C);
}
```

```
/* 001
   14.10.91
   ecvgen (eliptic curve vector generator)
   Generats some curves by using ecgen and stops immediatly if ecgen finds
   a factor, else the whole vector of curves will be generated.

   Def: Boolean ecvgen(C, P, n, factor)
        Vector_of_curves  C;         (* wird erzeugt *)
        Vector_of_points  P;         (* wird erzeugt *)
        LONGTYPE n,         (* die zu faktorisierende Zahl *)
        factor;    (* Faktor von n, falls zufaellig einer gefunden wird *)
   Res: TRUE, if a factor is found;
        FALSE, otherwise */

#include 'ecmp-types.h'
#include 'allg.h'
#include <stdio.h>


Boolean ecvgen(C, P, n, factor)
   Vector_of_curves  C;
   Vector_of_points  P;
   LONGTYPE          n, factor;
{
   int               i;
   Boolean           factor_found = FALSE;

   for (i = 0; C[i] != NULL && P[i] != NULL && !(factor_found); i++)
      factor_found = ecgen(C[i], P[i], n, factor);

   return (factor_found);
}
```

```
/* 005
   12.11.91
   15.6.92  Netz-version
   ecvfindfactor - elliptic curve find factor
   Tries to find factors with ECM.

   Def: Status ecvfindfactor(B, C, n, P, pp, factor)
        ULONG B,            // upper limit for primes
              C;            // upper limit for powers of primes
        LONGTYPE n,         // number to factor
                 factor;    // the result if a factor was found
        Vector_of_points P;
        PRIMETABLE *pp;     // pointer on the table of primes

   Res: no_factor_found, if no factor of n was found
        factor_found, if a factor was found
        hopeless, if more then ??? curves are not useable
```

```
   B and C are parameters of ECM. B is the upper limit for the primes and C
   the upper limit for the powers of primes.
*/

#include <stdio.h>
#include <math.h>
#include "allg.h"
#include "larith.h"
#include "ecmp-types.h"
#include "larith+.h"


Status ecvfindfactor(B, C, n, P, pp, factor)
   ULONG             B, C;      /* limits for ecm */
   LONGTYPE          n, factor;
   Vector_of_points  P;
   PRIMETABLE       *pp;
{
   Status            condition = no_factor_found;
   ULONG             primzahl;
   int               primpot, count = 0;

/*   printf("->ecvfindfactor\n"); fflush(stdout);*/
   resetprimes(pp);
   while ((primzahl = getprimes2(pp)) <= B && condition == no_factor_found)
   {
      primpot = (double) log((double) C) / (double) log((double) primzahl);
      condition = epvmul(ulpower(primzahl, primpot), P, P, n, factor);

/* bei verteilter Anwendung, wird hier geguckt, ob neue Nachrichten
   angekommen sind */
#ifdef PARALLEL
      if ((++count) % 100 == 0)
      {
         if (new_job())
            condition = new_message;
      }
#endif

   }
   return (condition);
}




/* 001
   11.10.91
   epalloc (elliptic point allocate)
   allocates memory for a point on an elliptic curve

   Def: Point epalloc(size);
        unsigned int size;           // size of LONGTYPE
   Res: Pointer on the allocated memory

   Bugs:
   If the allocation fails, the function lalloc from larith breaks with
   FATAL_ERROR. */

#include <stdio.h>
#include <malloc.h>
#include "ecmp-types.h"

Point epalloc(size)
   unsigned int      size;
{
   Point             P;

   P = (Point) malloc(sizeof(struct POINT));
   P->x = lalloc(size + 1);
   P->y = lalloc(size + 1);
   P->O = TRUE;                 /* P=O */
   P->C = NULL;

   return (P);
```

```
}
```

```
/* 001
   10.10.91
   epfree (elliptic point free)
   releases the memory which was allocated by epalloc

   Def: void epfree(P);
        Point P;
   Res: none
*/

#include <malloc.h>
#include "ecmp-types.h"

void              epfree(P)
   Point             P;
{
   free(P->x);
   free(P->y);
   free(P);
}
```

```
/* 001
   11.10.91
   epwrite (elliptic point write)
   writes the parameter from a point on an elliptic curve in the following
   form on the screen:
   if P = O: O
   else:
             x: x
             y: y
   (Ausgabe von ecwrite)

   Def: void epwrite(P);
        Point P;
   Res: none;  */

# include <stdio.h>
# include "ecmp-types.h"

void epwrite(P)
Point P;
{
   if (P==NULL)
      printf("no point\n");
   else
   {
      if (P->O)
      printf("O\n");
      else
      {
         printf("x: ");
         lwrite(P->x);
         printf("y: ");
         lwrite(P->y);
      }
      printf("on curve:\n");
      ecwrite(P->C);
   }
   return;
}
```

```
/* 001
   17.10.91
```

```
   epsto - (elliptic point store)
   epsto(P, Q) stores all components of point P in point Q (P->Q)

   Def: void epsto(P, Q);
        Point P, Q;
   Res: none;
*/

#include "larith.h"
#include "ecmp-types.h"

void epsto(P, Q)
Point P, Q;
{
   llsto(P->x, Q->x);
   llsto(P->y, Q->y);
   Q->O = P->O;
   Q->C = P->C;
}
```

```
/* 001
   18.10.91
   epcomp (elliptic point compare)
   Compares two points (P == Q)

   Def: int epcomp(P, Q)
        Point P, Q;
   Res: 0, falls P = Q
        1, sonst
*/

#include "larith.h"
#include "ecmp-types.h"

int epcomp(P, Q)
Point P, Q;
{

   if ( (eptestO(P) == eptestO(Q)) &&
        (eptestO(P) || (llcomp(P->x, Q->x) == 0 && llcomp(P->y, Q->y) == 0)) )
      return(FALSE);
   else return(TRUE);
}
```

```
/* 001
   18.10.91
   epnegcomp (elliptic point negation compare)
   P == -Q ?

   Def: epnegcomp(P, Q)
        Point P, Q;
   Res: 0, if P = -Q
        1, otherwise
*/

#include "larith.h"
#include "ecmp-types.h"

int epnegcomp(P, Q)
Point P, Q;
{

   if ( eptestO(P) == eptestO(Q) &&
       (eptestO(P) ||
        (llcomp(P->x, Q->x) == 0 &&
         llabscomp(P->y, Q->y) == 0 && lsig(P->y) != lsig(Q->y))) )
   return(0);
   else return(1);
}
```

```
/* 001
   eponcurve - elliptic point on curve

   test if the point is on the curve

   5.6.92
*/

#include "ecmp-types.h"
#include "larith+.h"
#include <stdio.h>

int eponcurve(P, n)
Point P;
LONGTYPE n;
{
   LONGTYPE h1, h2;
   int rw, ldim;

   ldim = lstellen(n)+2;

   h1 = lalloc(ldim);
   h2 = lalloc(ldim);

   llmulmod(P->x, P->x, n, h1);
   llmulmod(P->x, h1, n, h1); /* h1=x^3 */
   llmulmod(P->x, P->C->a, n, h2);
   lladdmod(h1, h2, n, h1);  /* h1=x^3+ax */
   lladdmod(h1, P->C->b, n, h1);
   llmulmod(P->y, P->y, n, h2);

   if (minus(h1)) lladd(h1, n, h1);
   if (minus(h2)) lladd(h2, n, h2);

   if (llcomp(h2, h1) != 0)
      rw=0;
   else
      rw=1;

   free(h1); free(h2);
   return(rw);
}
```

```
/* 001
   11.10.91
   epvalloc (elliptic point vector allocate)
   Allocates memory for a vector of points. The last vectorelement is a
   NULL pointer.

   Def: Vector_of_points epvalloc(quantity, size);
        unsigned int quantity,       // quantity of points
                         size;       // size of LONGTYPE
   Res: pointer on the allocated memory

   Bugs:
   If the allocation fails, the function lalloc from larith breaks with
   FATAL_ERROR. */

#include <stdio.h>
#include <malloc.h>
#include "ecmp-types.h"

Vector_of_points epvalloc(quantity, size)
unsigned int quantity, size;
{
   int j;
   Vector_of_points P;

   P = (Vector_of_points) calloc(quantity+1, sizeof(Point));

   for (j=0; j<quantity; j++)
      P[j]= epalloc(size);
   P[quantity]= NULL;
```

```
   return(P);
}
```

```
/* 001
   11.10.91
   epvfree (elliptic point vector free)
   releases the memory which was allocated by epvalloc

   Def: void epvfree(P);
        Vector_of_points P;
   Res: none */

#include <stdio.h>
#include <malloc.h>
#include "ecmp-types.h"

void epvfree(P)
Vector_of_points P;
{
  int i=0;

  while ( P[i] != NULL )
    epfree(P[i++]);
  free(P);
}
```

```
/* 001
   18.10.91
   epvlength (elliptic point vector length)
   Computes the length of a vector_of_points.

   Def: int epvlength(P);
        Vector_of_points P;
   Res: length of the vector P; P[ epvlength(p) ] = NULL;
*/

#include <stdio.h>
#include "ecmp-types.h"

epvlength(P)
   Vector_of_points  P;
{
   int               i;

   for (i = 0; P[i] != NULL; i++);

   return (i);
}
```

```
/* 001
   18.10.91
   epvactiv_curves (elliptic point vector activ curves)
   Counts the activ curves which belongs to a vector of points.

   Def: int epvactiv_curves(P);
        Vector_of_points P;
   Res: quantity of activ curves
*/

#include <stdio.h>
#include "ecmp-types.h"

epvactiv_curves(P)
   Vector_of_points  P;
{
```

```
   int              i, activ = 0;

   for (i = 0; P[i] != NULL; i++)
      if (P[i]->C->activ)
         ++activ;

   return (activ);
}




/* V001
   24.10.91
   epvadd - elliptic point vector addition
   adds a vector of points with another

   Def: Status epvadd(P, Q, Erg, n, factor)
        Vector_of_points P, Q, Erg;
        LONGTYPE n, factor;
   Res: hopeless,        if >=MINCDIM curves are not useable
        factor_found,    if a factor of n was found
        no_factor_found, otherwise
*/

#include "allg.h"
#include "ecmp-types.h"
#include "larith+.h"

enum addstate {ready, PP, PQ};

Status epvadd(P, Q, Erg, n, factor)
Vector_of_points P, Q, Erg;
LONGTYPE n, factor;
{
   Status condition = no_factor_found;
   enum addstate *as;
   Vector_of_LONGTYPES Inv;
   int Is,       /* INV_size */
       ac,       /* number of activ curves */
       pvl,      /* lenght of a Vector_of_points */
       i;
   Boolean curve_deleted;
   LONGTYPE h, h1, X, Y;

   h = lalloc(lstellen(n)+2);
   h1 = lalloc(lstellen(n)+2);
   X = lalloc(lstellen(n)+2);
   Y = lalloc(lstellen(n)+2);
   ac = epvactiv_curves(P);
   pvl = epvlength(P);

   /* hier merke ich mir wie weit die einzelnen Kurven in der Rechnung sind */
   as = (enum addstate*) calloc (pvl, sizeof(enum addstate));

   /* hier stehen die Zahlen, die invertiert werden sollen, die LONGTYPES
      werden nach Bedarf alloclert. */
   Inv = (Vector_of_LONGTYPES) calloc (ac+1, sizeof(LONGTYPE));

   do
   {
      Is = -1;
      curve_deleted = FALSE;
      for (i=0; P[i] != NULL; i++)
      {
         if ( !(P[i]->C->activ) )          /* If the curve is not activ */
            as[i] = ready;
         else if (eptest0(P[i]))           /* P = 0 */
         {
            if (Q[i] != Erg[i])
               epsto(Q[i], Erg[i]);
            as[i] = ready;
         }
         else if (eptest0(Q[i]))           /* Q = 0 */
         {
```

```
            if (P[i] != Erg[i])
               epsto(P[i], Erg[i]);
            as[i] = ready;
         }
         else if (epnegcomp(P[i], Q[i]) == 0)   /* P = -Q */
         {
            epset0(Erg[i]);
            as[i] = ready;
         }
         else if (epcomp(P[i], Q[i]) == 0)             /* P = Q */
         {
            as[i] = PP;
            Inv[++Is] = lalloc(lstellen(n)+2);
            llsto(P[i]->y, Inv[Is]);
            lmul2mod(Inv[Is], n);        /* Inv[i] = 2*y mod n */
         }
         else                                         /* P != Q */
         {
            as[i] = PQ;
            Inv[++Is] = lalloc(lstellen(n)+2);
            llsubmod(Q[i]->x, P[i]->x, n, Inv[Is]); /* Inv[Is] = x2-x1 */
         }
      }
      Inv[++Is] = NULL;  /* Ende von Inv markieren */

      if (Inv[0] != NULL)    /* vector Inv is not empty */
      {
/* der Vektor mit den zu invertierenden Zahlen ist zusammengestellt, jetzt
   kann invertiert werden. */
         if (!lvinvert(Inv, n, Inv, Is))
         {                   /* Invertierung hat nicht geklappt, das bedeuted:
                                entweder Faktor gefunden,
                                oder kein Faktor gefunden, dann muss diese
                                Kurve deaktiviert werden */
            int c;
            if ( (c = lvinv_fails(Inv, n, factor)) >=0 )
            {             /* Kurve c taugt nichts und muss gestrichen werden */
               P[c]->C->activ = FALSE;
               if (--ac < MINCDIM)
               condition = hopeless;
               curve_deleted = TRUE;
               printf("EPVADD: Kurve %d inaktiviert\n", c);
            }
            else             /* Faktor gefunden */
            {
               condition = factor_found;
/*             printf("EPVADD: Faktor gefunden: ");
               lwrite(factor);*/
            }
         }
         else
         {                             /* Invertierung hat funktioniert */
            Is=0;
            for (i=0; P[i] != NULL; i++)
            {
               if (as[i] == ready) continue;
               if (as[i] == PP)                  /* ----------- berechne 2*P */
               {
                  /* berechne x-Koordinate */
                  llmulmod(P[i]->x, P[i]->x, n, h);       /* h= x1^2 */
                  lsmulmod(h, 3, n, h);               /* h= 3*x1^2 */
                  lladdmod(h, P[i]->C->a, n, h);         /* h= 3*x1^2+a */
                  llmulmod(h, Inv[Is++], n, h);      /* h= (3*x1^2+a)/(2*y1) */
                  llmulmod(h, h, n, h1);             /* h1= ( (")/(") )^2 */

                  llsto(P[i]->x, X);
                  lmul2mod(X,n);                     /* X= 2*x1 */
                  llsubmod(h1, X, n, X);             /* X= ( ... )^2 - 2*x1 */
               }
               else                            /* ------- berechne P+Q, P!=Q */
               {
                  /* berechne x-Koordinate */
                  llsubmod(Q[i]->y, P[i]->y, n, h);       /* h= y2-y1 */
```

```
                  llmulmod(h, Inv[Is++], n, h);     /* h= (y2-y1)/(x2-x1) */

                  llmulmod(h, h, n, X);             /* X= ((")/("))^2 */
                  llsubmod(X, P[i]->x, n, X);          /* X= (")^2-x1 */
                  llsubmod(X, Q[i]->x, n, X);          /* X= (")^2-x1-x2 */
               }
               /* berechne y-Koordinate */
               llsubmod(P[i]->x, X, n, Y);          /* Y= (x1-x3) */
               llmulmod(h, Y, n, Y);             /* Y= h * (x1-x3) */
               llsubmod(Y, P[i]->y, n, Y);          /* Y= -y1+ h * (x1-x3) */

               /* speichere Ergebnis */
               llsto(X, Erg[i]->x);
               llsto(Y, Erg[i]->y);
               epunset0(Erg[i]);
               Erg[i]->C = P[i]->C;
            }
         }
      }
   }
   while (curve_deleted && condition != hopeless);

   free(h); free(h1); free(X); free(Y);
   free(as);
   lvfree(Inv);
/*   printf("epvadd->\n"); fflush(stdout);*/
   return(condition);
}




/* 001
   21.10.91 5.6.92
   epvmul - elliptic point vector multiplication
   multiplys a vector of points with a scalar

   Def: Status epmul(n, P, Erg, m, factor)
        unsigned long n;
        Vector_of_points P, Erg;
        LONGTYPE m, factor;

   Res: hopeless,        if curve is not useable
        factor_found,    if a factor of m was found
        no_factor_found, otherwise

   Algorithmus aus Horster S.299 (Die Variablennamen wurden genau so gewaehlt)
*/

#include "allg.h"
#include "larith.h"
#include "ecmp-types.h"
#include <stdio.h>

Status            epvmul(n, P, Erg, m, factor)
   ULONG             n;
   Vector_of_points  P, Erg;
   LONGTYPE          m, factor;
{
   int               ldim, cdim, i;
   Vector_of_points  Z, X;
   Status            condition = no_factor_found;

/*   printf("->epvmul\n"); fflush(stdout);*/

   ldim = lstellen(m) + 2;
   cdim = epvlength(P);

   X = epvalloc(cdim, ldim);
   Z = epvalloc(cdim, ldim);

   for (i = 0; P[i] != NULL; i++)
   {
      epsto(P[i], X[i]);
/*      Z[i]->C = P[i]->C;      Z[i] liegt auf der gleichen Kurve wie P[i] */
```

```
      epset0(Z[i]);             /* Z = neutrales element */
   }

   while (n > 0)
   {
      while (!(n & 0x00000001)) /* while n gerade */
      {
         condition = epvadd(X, X, X, m, factor);
         if (condition != no_factor_found)
         {
/*          printf("EPVMUL1: condition = %d\n", condition);*/
            goto ENDE;
         }
         else
            n >>= 1;
      }
      condition = epvadd(X, Z, Z, m, factor);
      if (condition != no_factor_found)
      {
/*          printf("EPVMUL2: condition = %d\n", condition);*/
         goto ENDE;
      }
      else
         n -= 1;
   }
   for (i = 0; Z[i] != NULL; i++)
      epsto(Z[i], Erg[i]);
ENDE:
   epvfree(X);
   epvfree(Z);
/*   printf("epvmul->\n"); fflush(stdout);*/

   return (condition);
}




/* 001
   ecm_stdcont - Standard-Continuation fuer ECM
   Standard-Continuation nach Montgomery

   Def: Status ecm_stdcont(P, D, n, pp, factor)
        Vector_of_points  P;      // Punktevektor, Ergebnis von Phase 1
        ULONG             D;      // oberes Primzahllimit
        LONGTYPE          n,      // zu fakt. Zahl
                          factor; // Faktor, wenn einer gefunden wurde
        PRIMETABLE       *pp;     // Pointer auf Primzahltafel

   Res: no_factor_found, if no factor of n was found
        factor_found, if a factor was found
        hopeless, if more then ??? curves are not useable
*/

#include <stdio.h>
#include "allg.h"
#include "larith.h"
#include "ecmp-types.h"
#include "larith+.h"

Status            ecm_stdcont(P, D, n, pp, factor)
   Vector_of_points  P;
   ULONG             D;
   LONGTYPE          n, factor;
   PRIMETABLE       *pp;
{
   Vector_of_points  v_epv[MAXPRIMDIF + 1];
   ULONG             prim, primalt;
   int               ldim, cdim, i;
   Status            condition;

   ldim = lstellen(n) + 2;
   cdim = epvlength(P);
   condition = no_factor_found;
```

```
   printf("\n HALLO: STDCONT\n");
   for (i = 0; i <= MAXPRIMDIF; i++)    /* Array init. */
      v_epv[i] = NULL;

   v_epv[1] = P;

   /* Tabelle berechnen */
   for (i = 2; i <= MAXPRIMDIF && condition == no_factor_found; i++)
   {
      v_epv[i] = epvalloc(cdim, ldim);
      condition = epvadd(P, v_epv[i - 1], v_epv[i], n, factor);
   }
   if (condition != no_factor_found)
      goto ENDE;

   prim = getprimes2(pp);
   if (prim == 1)
   {
      printf("ECM_STDCONT(1): Bin am Ende der Primzahltafel!!\n");
      condition = hopeless;
      goto ENDE;
   }
   condition = epvmul(prim, P, P, n, factor);   /* Start berechnen */

   if (condition == no_factor_found)
   {
      primalt = prim;
      do
      {
         prim = getprimes2(pp);
         if (prim == 1)
         {
            printf("ECM_STDCONT(2): Bin am Ende der Primzahltafel!!\n");
            condition = hopeless;
            goto ENDE;
         }
         condition = epvadd(P, v_epv[prim - primalt], P, n, factor);
         primalt = prim;
      }
      while (condition == no_factor_found && prim < D);
   }
ENDE:
   printf("      letzte Primzahl: %d\n", prim);
   fflush(stdout);
   /* Speicher wieder freigeben */
   for (i = 2; i <= MAXPRIMDIF; i++)
      epvfree(v_epv[i]);

   free(v_epv);
   return (condition);
}




/* 001
   ecm_stdcont2 - verb. Standard-Continuation fuer ECM (siehe Kapitel 3.2)
   Franz-Dieter Berger 1.6.92

   8.6.92 dynamisches platzsparendes Speicherkonzept
   16.6.92 Netzversion integriert

   verbesserte Standard-Continuation nach Montgomery

   Def: Status ecm_stdcont2(P, B1, B3, n, pp, factor)
        Vector_of_points  P;      // Punktevektor, Ergebnis von Phase 1
        ULONG             B1,     // obere Primzahlschranke fuer ECM
                          B3;     // obere Primzahlschranke fuer Continuation

        LONGTYPE          n,      // zu fakt. Zahl
                          factor; // Faktor, wenn einer gefunden wurde
        PRIMETABLE       *pp;     // Pointer auf Primzahltafel

   Res: no_factor_found, if no factor of n was found
```

```
       factor_found, if a factor was found
       hopeless, if more then ??? curves are not useable

*/

#include <stdio.h>
#include "ecmp-types.h"
#include "larith.h"
#include "larith+.h"
#include <math.h>

#define DIV 30                  /* w soll teilbar sein durch 30 = 2*3*5 */
#define KOMP 0.267              /* Komprimierungsfaktor fuer Tabelle: 30/8 =
                                   2.6666666666666 */


Status            ecm_stdcont2(P, B1, B3, n, pp, factor)
   Vector_of_points  P;         /* P nach ECM-Lauf */
   ULONG             B1,        /* obere Primzahlschranke fuer ECM */
                     B3;        /* obere Primzahlschranke fuer Continuation */
   LONGTYPE          n, factor;
   PRIMETABLE       *pp;        /* Pointer auf Primzahltafel */
{
   int               ldim, cdim, w, v1, i, j, v, q, u, wb3;
   unsigned long     SPmax = 1000000;   /* max. erlaubter Speicherplatz fuer
                                           Tafel */
   LONGTYPE          c, lh;
   Vector_of_points  Q;
   Vector_of_LONGTYPES *x = NULL;
   Status            condition;

   ldim = lstellen(n) + 2;
   cdim = epvlength(P);

   c = lalloc(ldim);
   lh = lalloc(ldim);
   Q = epvalloc(cdim, ldim);

   for (i = 0; i < cdim; i++)
   {
      Q[i]->C = P[i]->C;        /* Q[i] soll auf der gleichen Kurve wie P[i]
                                   liegen */
      epset0(Q[i]);             /* Vektor mit 0 init. */
   }
   condition = no_factor_found;
   wb3 = (int) ceil(sqrt((double) B3));
   w = wb3 / DIV * DIV;         /* wb3/DIV ist nach unten abgerundet, weil
                                   int-op */
   if (w + DIV - wb3 < wb3 - w)
      w += DIV;                 /* w hat den naechsten durch DIV teilbaren
                                   Wert bei wb3 */
   if (w < DIV)
      w = DIV;                  /* so gross solte w mindestens sein! */
   B3 = w * w;

   while ((int) (KOMP * w * ldim * cdim * 4) > SPmax)
      w -= DIV;                 /* solange zuviel Speicherplatz verbraucht
                                   wird, muss man w verkleinern */
   if (w < DIV)
   {
      printf("ACHTUNG: zuwenig Speicherplatz zur Verfuegung. SCHADE!\n");
      printf("w: %d\n", w);
      condition = hopeless;
      goto SCHLUSS;
   }

   v1 = B1 / w;

   /* Platz fuer Tafel allocieren: cdim Kurven, pro Kurve w LONGTYPES */
   x = (Vector_of_LONGTYPES *) calloc(cdim + 1, sizeof(Vector_of_LONGTYPES));
   for (i = 0; i < cdim; i++)
   {
      x[i] = (Vector_of_LONGTYPES) calloc(w + 2, sizeof(LONGTYPE));
      for (j = 0; j < w + 2; j++)
         x[i][j] = NULL;
```

```
      /* Speicherplatz fuer LONGTYPES wird erst bei Bedarf alloclert */
   }

   /* Tafel berechnen */
   for (j = 1; j <= w; j += 2)
   {
      if ((condition = epvadd(Q, P, Q, n, factor)) != no_factor_found)
         goto ENDE;
      if (gcd(w, j) == 1)
      {
         for (i = 0; i < cdim; i++)
         {
            x[i][j] = lalloc(ldim);
            llsto(Q[i]->x, x[i][j]);
         }
      }
      if ((condition = epvadd(Q, P, Q, n, factor)) != no_factor_found)
         goto ENDE;
   }
   /* jetzt gilt: Q = w*P */

   /* berechne jetzt den Startwert P = v1*Q = v1*w*P */
   if ((condition = epvmul(v1, Q, P, n, factor)) != no_factor_found)
      goto ENDE;

   /* jetzt geht's los */
   q = getprimes2(pp);
   v = v1;
   j = 1;
   set1(c);

   while (q < B3 && condition == no_factor_found)
   {
      if ((u = v * w - q) < 1)
      {
         v++;
         if ((condition = epvadd(P, Q, P, n, factor)) != no_factor_found)
            goto ENDE;
         u = v * w - q;
      }

      for (i = 0; i < cdim; i++)
      {
 llsubmod(P[i]->x, (x[i][u]), n, lh);        /* lh=(P[i]->x -
(x[i][j])) mod n */
 llmulmod(c, lh, n, c);      /* c= c*(P[i]->x - (x[i][u])) mod n */
      }

      if ((j = j + cdim) >= 500)/* alle 500 Mul. soll ggT berechnet werden */
      {
         j = 1;
         llgcd0(c, n, factor);
         if (!test1(factor))
            condition = factor_found;
#ifdef PARALLEL
         else if (new_job())
            condition = new_message;
#endif
      }
      q = getprimes2(pp);
   }

   if (condition == no_factor_found)
   {
      llgcd0(c, n, factor);
      if (!test1(factor))
         condition = factor_found;
   }

ENDE:
   if (llcomp(factor, n) == 0)
   {
      set1(factor);
      condition = hopeless;
```

```
   }

   /* Speicherplatz wieder freigeben */
   for (i = 0; i < cdim; i++)
   {
      for (j = 0; j <= w + 2; j++)
         if (x[i][j] != NULL)
            free(x[i][j]);
      free(x[i]);
   }
   free(x);
SCHLUSS:
   free(c);
   free(lh);
   epvfree(Q);

   return (condition);
}
/* ---------------------------------------------------------------------- */

/* berechnet den ggT von a,b
   gcd(0, 0) = 0;
   gcd(u, 0) = u;
*/
gcd(a, b)
   int              a, b;
{
   int              r, h;

   if (a == 0)
      return (abs(b));
   if (b == 0)
      return (abs(a));

   while (1)
   {
      r = a % b;
      if (r == 0)
         break;
      h = a;
      a = b;
      b = r;
      r = h;
   }

   return (abs(b));
}




/* 001
   read_parameters - reads the contents of the PARAMETERFILE

   Res: erstes Argument der eingelesenen Zeile (muss mit der gewuenschten
        Stellenanzahl uebereinstimmen)
   FDB 6.11.92
*/

#include "allg.h"
#include "larith+.h"
#include <stdio.h>


int               read_parameters(stell, B1, B3, curves)
   int              stell;
   unsigned long   *B1, *B3;
   int             *curves;
{
   FILE            *fp;
   int              i = 0;
   char             zeile[50];

   if ((fp = fopena(PARAMETERFILE, "r")) == NULL)
   {
      printf("\nFEHLER: Ich kann keines der Parameterfiles %s oeffnen!!\n",
```

```
     PARAMETERFILE);
      exit(1);
   }
   else
   {
      i = -1;
      do
      {
 fgets(zeile, 50, fp);
 if (isdigit(zeile[0]))
 {
    sscanf(zeile, "%d %lu %lu %d", &i, B1, B3, curves);
 }
      }
      while (i != stell && i != 0);

      fclose(fp);
   }
   return (i);
}



/* 001
   read_strategy - reads the contents of the STRATEGYFILE and stores it into
   an array

   FDB 2.5.92
*/

#include "allg.h"
#include "larith+.h"
#include <stdio.h>


int read_strategy(strategy)
   float             strategy[MAXSTRATEGY][2];
{
   FILE             *fp;
   int               i = 0;

   if ((fp = fopena(STRATEGYFILES, "r")) != NULL)
   {
      for (i = 0; fscanf(fp, "%f %f", &strategy[i][0], &strategy[i][1]) != EOF
   && i < MAXSTRATEGY - 1; i++);
      fclose(fp);
   }
   strategy[i][0] = 0;
   return (i);
}




/* 002
   job_planing - computes the parameters for ecm

   FDB 30.4.92
   25.8.92 info-Parameter: info = 0 keine Ausgaben, sonst schon
   6.11.92 neue Parameter (mit Mathematika berechnet, werden eingelesen
                        mit read_parameters)
*/

#include "ecmp-types.h"
#include "allg.h"
#include<stdio.h>

int               job_planing(machines, strat, buf, info)
   int               machines, info;
   float             strat[MAXSTRATEGY][2];
   unsigned long     buf[MAXJOBS][3];
{
   int               i, j, i1, i2, job = 0, anz_jobs, aj, cpm, curves,
                     c, delta, deltaold;
```

```
   unsigned long     B1, B3;

   if (info)
   {
      printf("   #   |           | max.Prim | max.Prim |#Kurven/| #Kurven\n");
      printf("Stellen|    Jobs   |    ECM   | Contin.  | gesamt | Rechner\n");
      printf("------------------------------------------------------------\n");
   }

   for (i = 0; strat[i][0] != 0; i++)
   {
      if (read_parameters((int) strat[i][0], &B1, &B3, &curves) !=
  ((int) strat[i][0]))
      {
 printf("job_planing: Fehler beim Einlesen der Parameter!!!!\n");
 printf("%f\n", strat[i][0]);
 exit(1);
      }

      cpm = curves / machines; /* curves pro machine */
      if (cpm < CS_MIN)
 cpm = CS_MIN;
      else
      {
 if (cpm > CS_MAX)
    cpm = CS_MAX;
      }

      anz_jobs = curves / cpm;
      aj = anz_jobs;
      c = cpm;
      delta = 0;
      deltaold = abs(anz_jobs * cpm - curves);
      for (i1 = aj - 2; i1 <= aj + 2; ++i1)
 for (i2 = c - 2; i2 <= c + 2; ++i2)
 {
    delta = abs(i1 * i2 - curves);
    if (delta < deltaold)
    {
       cpm = i2;
       anz_jobs = i1;
       deltaold = delta;
    }
 }

      if (info)
 printf("    %2.0f | %3d -", strat[i][0], job);

      for (j = 1; ((j <= anz_jobs) && (job < MAXJOBS)); j++)
      {
 buf[job][0] = cpm;
 buf[job][1] = B1;
 buf[job++][2] = B3;
      }

      if (info)
 printf(" %3d | %8d |%9d | %6d | %7d\n", job - 1, B1, B3, curves, cpm);

   }
   return (job);
}




/*001
  new_job - is there a new job?
  tests if a new job and if this new job is the newest */


#include <stdio.h>
#include <string.h>
#include "ecmp-types.h"
#include "larith.h"
#include "larith+.h"
```

```
#include "/home/thsetz/lips_2.1Beta/include/lips.h"
#include "allg.h"

/* globale Variablen */
extern int        g_ldim, g_job_nr, job_nr, g_randgenset, g_sim_curves,
                  timer[2], phase;
extern char       g_n_string[MAXDEZSTEL], symbol[6];
extern ULONG      g_B, g_D;
extern LONGTYPE   n; /* zu faktorisierende Zahl */


Boolean           new_job()
{
   LONGTYPE          l_n; /* local n */
   Boolean           rw = FALSE;

   if (inp(symbol, "ISLLILI", &g_ldim, g_n_string, &g_B, &g_D, &g_randgenset,
       &g_sim_curves, &g_job_nr) == 0)
   {
      printf("\n--- Neuer Job angekommen! ");
      l_n = lalloc(g_ldim);

      string_to_l(g_n_string, l_n);
      if (llcomp(l_n, n) == -1) /* nur wenn neues n kleiner als altes,
   akzeptiere neuen Job */
      {
 printf("akzeptiert\n");
 rw = TRUE;
      }
      else
      {
 printf("zu alt\n");
 rw = FALSE;
      }
      free(l_n);
   }
   return (rw);
}
```

E.2.2 larith.a

```
##############################################################################
#                                                                            #
#     Makefile fuer larith+.a
#     12.7.91                                                                #
#     (entspricht im wesentlichen dem makefile aus SunOS 4.0 Band 9 S.163) #
#     moeglich ist: make debug, make prof, make gprof, make all variants   #
#                                                                            #
##############################################################################


SOURCES= lvalloc.c lvfree.c \
        flread3.c flstelwrite.c lstelwrite.c\
        initzmod.c nextprime.c getprimes.c\
        listprim.c lpotmod.c lrandom.c logarithmen.c\
        llmod.c llmulmod.c lladdmod.c llsubmod.c lmul2mod.c llgcds2.c\
        lsmulmod.c lsaddmod.c lssubmod.c llcompmod.c\
        ltrialdiv.c lvinvert.c lvinv_fails.c \
        ulpower.c \
        zeit.c fopena.c fexist.c mips.c

# Hier den Namen der Library, die erzeugt werden soll eintragen (ohne Pfad)
LIBRARY= larith+.a

# Achtung: $MACHINE muss im .cshrc auf crypt oder grad gesetzt werden
CFLAGS = -O4 -I$(INCLUDE) -Bstatic
CPPFLAGS =
#OUTPUT_OPTION = -o $@

MEMBERS= $(SOURCES:.c=.o)
DEBUG= $(LIBRARY:%=DEBUG/%)
PROF= $(LIBRARY:%=PROF/%)
GPROF= $(LIBRARY:%=GPROF/%)
VARIANTS.o= $(MEMBERS)

INCLUDE= $(HOME)/DIPLOM/INCLUDE

.KEEP_STATE:
.PRECIOUS: $(LIBRARY)
.INIT:
        @-mkdir -p DEBUG PROF GPROF

all: $(LIBRARY)
debug: $(DEBUG)
prof: $(PROF)
gprof: $(GPROF)
variants: debug prof gprof

debug := CFLAGS= -g -I$(INCLUDE) -D$(MACHINE)
prof := CFLAGS= -p -O -I$(INCLUDE) -D$(MACHINE)
gprof := CFLAGS= -pg -O -I$(INCLUDE) -D$(MACHINE)
$(DEBUG) := VARIANTS.o = $(MEMBERS:%=DEBUG/%)
$(PROF) := VARIANTS.o = $(MEMBERS:%=PROF/%)
$(GPROF) := VARIANTS.o = $(MEMBERS:%=GPROF/%)

#-----------------------------------------------------------------------------


$(LIBRARY) $(DEBUG) $(PROF) $(GPROF): $$(VARIANTS.o)
                                ar rv $@ $?
                                ranlib $@

$(LIBRARY)(%.o): %.o
$(DEBUG)(DEBUG/%.o): DEBUG/%.o
$(PROF)(PROF/%.o): PROF/%.o
$(GPROF)(GPROF/%.o): GPROF/%.o
DEBUG/%.o PROF/%.o GPROF/%.o: %.c
        $(COMPILE.c) -o $@ $<

clean:
        rm -rf $(LIBRARY) $(MEMBERS) DEBUG PROF GPROF



/* 001
```

```
   16.10.91
   lvalloc (LONGTYPE vector allocate)
   Allocates memory for a vector of LONGTYPE. The last vectorelement is a
   NULL pointer.

   Def: Vector_of_LONGTYPES lvalloc(quantity, size);
        unsigned int quantity,       // quantity of LONGTYPES
                     size;           // size of LONGTYPE
   Res: pointer on the allocated memory

   Bugs:
   If the allocation fails, the function lalloc from larith breaks with
   FATAL_ERROR. */

#include <stdio.h>
#include <malloc.h>
#include "larith+.h"

Vector_of_LONGTYPES lvalloc(quantity, size)
unsigned int quantity, size;
{
   unsigned int j;
   Vector_of_LONGTYPES L;

   L = (Vector_of_LONGTYPES) calloc(quantity+1, sizeof(LONGTYPE));

   for (j=0; j<quantity; j++)
      L[j]= lalloc(size);
   L[quantity]= NULL;

   return(L);
}
```

```
/* 001
   16.10.91
   lvfree (elliptic curve vector free)
   releases the memory which was allocated by lvalloc

   Def: void lvfree(L);
        Vector_of_LONGTYPES L;
   Res: none */

#include <malloc.h>
#include "larith+.h"


void lvfree(L)
Vector_of_LONGTYPES L;
{
  int i=0;

  for (i=0; L[i]!=NULL; i++)
    free(L[i]);
  free(L);
}
```

```
/* 001

Lesen langer Zahlen aus Datei ( im Klartext ), liefert EOF bei Dateiende;
=========================================================================

Funktion:  int flread3(fp,x)
           FILE *fp;
           LONGTYPE x;

Die lange Zahl x wird aus der Datei fp gelesen. Die Datei muss
offen sein. Format der Datei: pro Zeile eine lange Zahl im
Dezimalsystem (ASCII). Die Datei kann z.B. mit flwrite2()
erzeugt werden.
```

```
Bei Dateiende wird EOF geliefert;

Aenderung: 19.12.89
           23.05.90
           15.08.90 von FDB
*/

#include <stdio.h>
#include "larith.h"

int flread3(fp, x)

FILE *fp;
LONGTYPE x;

{
   if (fscanf(fp,"%s\n",v_string) == EOF)
             return(EOF);
   h_str_to_l(x);
   return(0);
}




/* 001

Ausgabe einer langen Zahl und ihrer Dezimalstellenanzahl
========================================================

Funktion:  void flstelwrite(fp, x)
                          FILE *fp;
           LONGTYPE x;

Es wird auf den stream fp ausgegeben:
(Dezimalstellenanzahl von x) x selbst

Letzte Aenderung: 26.02.91
*/


#include "larith.h"
#include <string.h>
#include <stdio.h>

void flstelwrite(fp, x)
FILE *fp;
LONGTYPE x;

{
   h_str_laenge(stel(x));
   lstring(x,v_string);
   fprintf(fp, "(%d) %s\n",strlen(v_string), v_string);
}




/* 001

Ausgabe einer langen Zahl und ihrer Dezimalstellenanzahl
========================================================

Funktion:  int lstelwrite(x)
           LONGTYPE x;

Es wird auf Standardausgabe ausgegben:
Dezimalstellenanzahl von x: x selbst

Letzte Aenderung: 26.02.91
*/

#include "larith.h"
#include <string.h>
#include <stdio.h>
```

```
void lstelwrite(x)
LONGTYPE x;
{
   h_str_laenge(stel(x));
   lstring(x,v_string);
   printf("(%d) %s\n",strlen(v_string), v_string);
}
```

```
/* 001
   Diese beiden Funktionen realisieren einen Modulo-Zaehler.
   Damit die Zaehlervariablen erhalten bleiben muessen sie static definiert
   werden.

   Def: int initzmod(x)
        LONGTYPE x;
   Res: immer 0;

   Def: inczmod(i)
        int i;
   Res: 0, wenn mindestens ein Modulo-Zaehler =0 ist
        1, sonst

   Mit initzmod(x) wird er initialisiert, d.h. es wird der Divisionsrest
   bestimmt.
   Mit inczmod(i) wird werden die Modulo-Zaehler um i erhoeht.
*/

#include "portab.h"
#include "larith.h"

static           zmod3, zmod5, zmod7, zmod11, zmod13;

int              initzmod(x)
   LONGTYPE         x;
{
   zmod3 = lsmod(x, (LONG) 3);
   zmod5 = lsmod(x, (LONG) 5);
   zmod7 = lsmod(x, (LONG) 7);
   zmod11 = lsmod(x, (LONG) 11);
   zmod13 = lsmod(x, (LONG) 13);
   return (0);
}

inczmod(i)
   int              i;
{
   zmod3 = (zmod3 + i) % 3;
   zmod5 = (zmod5 + i) % 5;
   zmod7 = (zmod7 + i) % 7;
   zmod11 = (zmod11 + i) % 11;
   zmod13 = (zmod13 + i) % 13;
   return (zmod3 && zmod5 && zmod7 && zmod11 && zmod13);
}
```

```
/* V001 FDB
   7.5.91
   nextprime - searches the next prime

   Def: void nextprime(z)
   LONGTYPE z;

   Findet die naechste Primzahl > z. Circa jede 10. Zahl wird getestet.
*/

#include "larith.h"
#include "allg.h"
```

```
#include "portab.h"
int             listprim();
int             initzmod();
int             inczmod();
int             lrandom();

void            nextprime(z)
   LONGTYPE        z;
{
   if (lmod2(z) == 0)          /* ist z gerade, dann addiere 1 hinzu */
      lsadd(z, (LONG) 1, z);
   initzmod(z);                /* Modulo-Zaehler initialisieren */
   while (!listprim(z))        /* while z keine Primzahl */
   {
      do
         lsadd(z, (LONG) 2, z); /* erhoehe z um 2, solange */
      while (!inczmod(2));     /* bis es nicht mehr durch kleine Prim- */
   }                           /* zahlen teilbar ist */
}
```

```
/* 001
   initprimes, getprimes, getprimes2, resetprimes, exitprimes - manages primes
      which are stored in a (compressed) table.

   Def: PRIMETABLE* initprimes(primetables)
        char *primetables;
   Res: pointer to the beginning of the table of primes

   Def: ULONG getprimes(fp)
        PRIMETABLE *fp;
   Res: the next prime in the primetable

   Def: ULONG getprimes2(fp)
        PRIMETABLE *fp;
   Res: the next prime in the compressed primetable

   Def: void resetprimes(fp)
        PRIMETABLE *fp;
   Res: none

   Def: void exitprimes(fp)
        PRIMETABLE *fp;
   Res: none

   initprimes tries to open a file with primes. 'primetables' is a string
   containing a list of files, which can be used (see fopena). If
   primetables == NULL then the constant "PRIMTAFEL" in "allg.h" is used.
   If this file does not exist, initprimes breaks with an error.

   getprimes returns the next prime from the table of primes. If the end of
   the primetable is reached, the constant 1 is returned.

   getprimes2 returns the next prime from the compressed table of primes (only
   the difference between to consecutive primes is stored). If the end of
   the primetable is reached, the constant 1 is returned.

   resetprimes sets the pointer on the primetable on the first element of the
   primetable.

   exitprimes closes the primetable.
   15.10.91
   19.6.92 getprimes2; initprimes mit Tafelangabe
*/

#include "allg.h"
#include <stdio.h>
#include "larith.h"
#include "larith+.h"

static unsigned long prime;
static int        first;
```

```
PRIMETABLE       *initprimes(primetables)
   char             *primetables;
{
   PRIMETABLE       *fp;

   first = 1;
   if (primetables == NULL)
   {
      if ((fp = fopena(PRIMTAFEL, "r")) == NULL)
         lerror("Primzahltafel kann nicht geoeffnet werden!");
   }
   else
   {
      if ((fp = fopena(primetables, "r")) == NULL)
         lerror("Primzahltafel kann nicht geoeffnet werden!");
   }

   return (fp);
}

ULONG             getprimes(fp)
   PRIMETABLE       *fp;
{
   ULONG             primzahl;

   if (fscanf(fp, "%lu", &primzahl) != EOF)
      return (primzahl);
   else
      return (1);
}

ULONG             getprimes2(fp)
   PRIMETABLE       *fp;
{
   char              primdif;

   if (first)
   {
      prime = fgetc(fp);
      first = 0;
   }
   else
   {
      if ((primdif = fgetc(fp)) != EOF)
      {
         prime += (unsigned char) primdif;
      }
      else
         prime = 1;
   }
   return (prime);
}

void              resetprimes(fp)
   PRIMETABLE       *fp;
{
   rewind(fp);
   first = 1;
}

void              exitprimes(fp)
   PRIMETABLE       *fp;
{
   fclose(fp);
}



/* 002
   J.B. FDB
   Version vom 3.3.92
   Primzahltest nach Miller Rabin fuer lange Arithmetik

   Def: BOOLEAN listprim(n)
```

```
        LONGTYPE n;
   Res: TRUE, falls n ist Primzahl
        FALSE, sonst
*/

#include <stdio.h>
#include "larith.h"
#include "elliptic.h"
#include "allg.h"

#define ANZAHL 10

long             a[ANZAHL] = {3, 5, 7, 11, 13, 17, 19, 23, 29, 31};

void             lpotmod();
BOOLEAN          h_M_R_Test();

BOOLEAN          listprim(n)
   LONGTYPE         n;
{
   LONGTYPE         q, k;
   int              i, l, LDIM;

   LDIM = (lstellen(n) + 1) * 2;

   if (!lscomp(n, 2))
   {
      return (TRUE);
   }
   for (l = 0; l < ANZAHL; l++)
   {
      if (!lscomp(n, a[l]))
         return (TRUE);
   }

   q = lalloc(LDIM);
   k = lalloc(LDIM);

   llsto(n, q);
   lsadd(q, -1, q);
   slsto(0, k);
   while (lmod2(q) == 0)
   {
      ldiv2(q);
      lsadd(k, 1, k);
   }
   l = 0;
   while (h_M_R_Test(n, q, k, l) && (l < ANZAHL))
      l++;
   free(q);
   free(k);
   if (l >= ANZAHL)
      return (TRUE);
   else
      return (FALSE);
}


/* J.B. */
/* Eigentlicher Test nach Miller, Rabin */
int              h_M_R_Test(n, q, k, l)
   LONGTYPE         q, n, k;
   int              l;
{
   LONGTYPE         i, y, nm1, erg, y1;
   extern long      a[ANZAHL];
   BOOLEAN          rw = FALSE;
   int              LDIM;

   LDIM = (lstellen(n) + 1) * 2;
   i = lalloc(LDIM);
   y = lalloc(LDIM);
   nm1 = lalloc(LDIM);
   erg = lalloc(LDIM);
   y1 = lalloc(LDIM);
```

```
   lsadd(n, -1, nm1);
   slsto(0, i);
   slsto(a[l], erg);
   lpotmod(erg, q, n, y);
   slsto(1, erg);
   if ((llcomp(y, erg) == 0) || (llcomp(y, nm1) == 0))
      rw = TRUE;
   else
   {
      lsadd(i, 1, i);
      while (llcomp(i, k) == -1)
      {
         llmul(y, y, y1);
         lldiv(y1, n, 0, y);
         if ((llcomp(y, nm1) == 0))
         {
            rw = TRUE;
            goto ENDE;
         }
         else
         {
            slsto(1, erg);
            if (llcomp(y, erg) == 0)
            {
               rw = FALSE;
               goto ENDE;
            }
            else
               lsadd(i, 1, i);
         }
      }
   }
ENDE:
   free(i);
   free(y);
   free(nm1);
   free(erg);
   free(y1);
   return (rw);
}
```

```
/* 001
   lpotmod - potenzieren modulo

   Def: void lpotmod(x, n, m, erg)
        LONGTYPE x, n, m, erg;
   Res: keines

   berechnet erg = x^n MOD m (Horster S.299) in Zeit:
   O(log n) Aufrufe von llmul und lldiv
*/


#include"stdio.h"
#include"larith.h"
#include"allg.h"

void              lpotmod(x, n, m, erg)
   LONGTYPE          x, n, m, erg;
{
   int               err, ldim;
   LONGTYPE          nh, ergh, x2;

   ldim = lstellen(m) + 2;

   nh = lalloc(2 * ldim);
   ergh = lalloc(2 * ldim);
   x2 = lalloc(2 * ldim);

   llsto(n, nh);
```

```
   slsto(1, ergh);
   while ((lscomp(nh, 0) != 0))
   {
      while ((!lmod2(nh)) && !(test0(nh)))
      {
         llmul(x, x, x2);
         lldiv(x2, m, 0, x);
         ldiv2(nh);
      }
      llmul(x, ergh, ergh);
      lldiv(ergh, m, 0, ergh);
      lsadd(nh, -1, nh);
   }
   llsto(ergh, erg);
   free(nh);
   free(ergh);
   free(x2);
}




/* 001
   lrandom (J.B. FDB) 8.1.91
   Erzeugt eine lange Zufallszahl

   Def: int lrandom(anzahl, x)
        int anzahl;   // Dezimalstellen der zu erzeugenden Zahl
        LONGTYPE x;   // Zufallszahl, die erzeugt wird
   Res: 0 in every case


   Bugs:
   The function can't check if there is enough memory for the random number.
   So you have the responsibility for this!!!!!!!!
   Als Startwert fuer die Zufallszahl wird Millisekunden aus dem time-struct
   benutzt. D.h., dass zu schnelle Aufrufe hintereinander, die gleiche
   Zufallszahl ergeben!
*/

#include "larith.h"
#include <math.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include "allg.h"

#define LOG10V2 0.30103         /* 10er Logarithmus von 2 */

static int        last_set = 0;
int               BITS = SIZE * 8;

int               lrandom(anzahl, x)
   int               anzahl;
   LONGTYPE          x;
{

   int               aus, i, bin_anzahl, rest, ldim, set, dummy1;
   char             *s;
   long              dummy;
   struct timeb      tp;

   ldim = ceil((anzahl / LOG10V2) / (BITS));

   s = (char *) malloc(anzahl + 3);
   bin_anzahl = ceil(anzahl / LOG10V2); /* Bitanzahl fuer Dez.stellen */

   aus = bin_anzahl / BITS;     /* Umwandl. Stellenzahl in Anzahl unsigned
                                   long */
   rest = bin_anzahl % BITS;
   if (ftime(&tp) != -1)
      set = ((int) tp.millitm) * time();
   else
      set = last_set;
```

```
   x[0] = aus;                  /* if possible set the random generator */
   if (set != last_set)
      srandom(set);

   for (i = 1; i <= aus; i++)   /* build the big random number */
      x[i] = random();

   if (rest > 0)
   {
      x[0] = aus + 1;
      x[aus + 1] = random();
      x[aus + 1] = (x[aus + 1] >> (BITS - rest)) | (1 << (rest - 1));
   }
   lstring(x, s);               /* ueberpruefe ob x die richtige Laenge hat */
   if (strlen(s) > anzahl)
   {
      lsdiv(x, 10, x, &dummy1);
   }
   free(s);
   return (0);
}
```

```
/* 001 */
/* stellt folgende Funktionen zur Verfuegung:
   log_2, llog2, l_log;
   log_2 berechnet den 2er-Logarithmus von einer unsigned long Zahl. Das
   Ergebnis wird abgerundet. In Wirklichkeit wird die Stelle des hoechsten
   gesetzten Bits ermittelt. Es gilt somit:
   log_2(0) = -1;
   log_2(1) = 0;
   llog2 macht das Gleiche fuer lange Zahlen, doch hier wird aufgerundet;
   l_log(z,b) berechnet den Logarithmus von LONGTYPE z zur Basis uns.long b.
   Das geht schief, falls b=1 (Division durch 0) oder
                    falls b=0 (Ergebnis unsinnig)
   aktuell: 22.2.91 */

#include "larith.h"
#include <stdio.h>
#include <math.h>

log_2(z)
   unsigned long     z;
{
   unsigned long     maske = 0x80000000;
   int               i = 0;

   while ((!(z & maske)) && (i++ < 31))
      maske >>= 1;
   return (31 - i);
}

llog2(z)
   LONGTYPE          z;
{
   int               h, h1;

   h = lstellen(z);
   h1 = log_2(z[h]);
   return (31 * (h - 1) + h1 + 1);
}

l_log(z, basis)
   LONGTYPE          z;
   unsigned long     basis;
{
   double            h;
   h = log2((double) basis);
   return ((int) (llog2(z) / log2((double) basis)));
}
```

```
/* V001
Lange MOD-Funktion
==================

Funktion: int llmod(x,y,r)
          LONGTYPE x, y, r;

Der Divisionsrest bei Division der langen Zahl x durch die lange Zahl y
wird der langen Zahl r zugewiesen.

aktuell: 17.10.91 */

#include "larith.h"

llmod(x, y, r)
   LONGTYPE          x, y, r;
{
   lldiv(x, y, 0, r);
   return (0);
}
```

```
/* V001
lange Multiplikation modulo
===========================

Funktion: int llmulmod(x, y, n, erg)
          LONGTYPE x, y, n, erg;

erg = (x * y) mod n

aktuell: 17.10.91 */

#include "larith.h"

llmulmod(x, y, n, erg)
   LONGTYPE          x, y, n, erg;
{
   LONGTYPE          h;

   h = lalloc(lstellen(x) + lstellen(y) + 2);

   llmul(x, y, h);
   llmod(h, n, erg);

   free(h);
   return (0);
}
```

```
/* V001
Lange Addition modulo
=====================

Funktion: lladdmod(x, y, n, erg)
          LONGTYPE x, y, n, erg;

erg = (x + y) mod n

Vor:
|x|, |y| < n; n > 0
Die Modulo-Reduktion wird (falls noetig) durch eine Subtraktion mit n
realisiert. Deshalb muessen x und y schon vorher reduziert (d.h. < |n|) sein!!!

aktuell: 18.10.91 */

#include "larith.h"

lladdmod(x, y, n, erg)
   LONGTYPE          x, y, n, erg;
```

```
{
   LONGTYPE         h;

   h = lalloc(lstellen(n) + 2);

   lladd(x, y, h);
   if (llabscomp(h, n) != -1)   /* |h| >= |n| ? */
      if (llcomp(h, n) != -1)   /* h >= n ? */
         llsub(h, n, erg);
      else
         lladd(h, n, erg);
   else
      llsto(h, erg);

   free(h);
   return (0);
}




/* V001
Lange Subtraktion modulo
========================

Funktion: llsubmod(x, y, n, erg)
          LONGTYPE x, y, n, erg;

erg = (x - y) mod n

Vor:
|x|, |y| < n; n > 0
Die Modulo-Reduktion wird (falls noetig) durch eine Subtraktion mit n
realisiert. Deshalb muessen x und y schon vorher reduziert (d.h. < |n|) sein!!!

aktuell: 17.10.91 */

#include "larith.h"

llsubmod(x, y, n, erg)
   LONGTYPE         x, y, n, erg;
{
   LONGTYPE         h;

   h = lalloc(lstellen(n) + 2);

   llsub(x, y, h);
   if (llabscomp(h, n) != -1)   /* |h| >= |n| ? */
      if (llcomp(h, n) != -1)   /* h >= n ? */
         llsub(h, n, erg);
      else
         lladd(h, n, erg);
   else
      llsto(h, erg);

   free(h);
   return (0);
}




/* V001
Lange Multiplikation mit 2 modulo
=================================

Funktion: lmul2mod(x, n)
          LONGTYPE x, n;

x = x*2 mod n

Vor:
|x| < n; n > 0
Die Modulo-Reduktion wird (falls noetig) durch eine Subtraktion mit n
realisiert. Deshalb muss x schon vorher reduziert (d.h. < |n|) sein!!!
```

```
ACHTUNG: Das kann 1 Bit groesser sein als das Endergebnis. Der Speicherplatz
fuer x muss dafuer gross genug sein !!!!!!

aktuell: 17.10.91 */

#include "larith.h"

lmul2mod(x, n)
   LONGTYPE          x, n;
{
   lmul2(x);

   if (llabscomp(x, n) != -1)   /* |h| >= |n| ? */
      if (llcomp(x, n) != -1)   /* h >= n ? */
         llsub(x, n, x);
      else
         lladd(x, n, x);

   return (0);
}




/*FDB

GCD zweier langer Zahlen ( ein Koeffizient der Darstellung )
========================

Funktion:  int llgcds2(a,b,erg,x)
           LONGTYPE a, b, erg, x;

Der GCD der langen Zahlen a und b wird der langen Zahl erg
zugewiesen. Die lange Zahl x erhaelt als Wert den ersten
Koeffizienten der Darstellung des GCD ( erg = x * a + y * b ).
Dabei wird versucht, die Division und Multiplikation durch
Subtraktion zu ersetzen.

11.02.92 */

#include "larith.h"
#include "larith+.h"

int              llgcds2(a, b, erg, x)

   LONGTYPE          a, b, erg, x;

{
   LONGTYPE          r0, m0, n0, rh0, h, q, r, m, n, rh;
   LONGTYPE          x0, xs0, xss0, xs, xss;
   unsigned long     a0, b0;
   int               qh;

   a0 = stel(a);
   b0 = stel(b);

   if (b0 == 0)
   {
      sto(a, erg);
      if (minus(a))
      {
         neg(erg);
         slsto(-1, x);
      }
      else
      {
         slsto(1, x);
      }
      return (0);
   }

   if (b0 > a0)
      a0 = b0;
   else
      b0 = a0;

   q = erg;
```

```
r0 = h_get_temp(b0);
r = r0;
rh0 = h_get_temp(b0);
rh = rh0;
n0 = h_get_temp(b0);
n = n0;
m0 = h_get_temp(b0);
m = m0;

x0 = x;
xs0 = h_get_temp(b0);
xs = xs0;
xss0 = h_get_temp(b0);
xss = xss0;

sto(a, m);
betrag(m);
sto(b, n);
betrag(n);
if (llcomp(m, n) == -1)      /* damit wird sichergestellt, dass m>n */
{
   h = m;
   m = n;
   n = h;
   set0(x0);
   set1(xs);
   set1(xss);
}
else
{
   set1(x0);
   set0(xs);
}

while (1)
{
   /* versuche Division zu ersetzen! */
   llsub(m, n, r);
   qh = 1;

   if (llcomp(r, n) >= 0)
   {
      lldiv(m, n, q, r);
      ++qh;
   }
   else
      set1(q);

   if (test0(r))
   {
      sto(xs, x);
      break;
   }
   h = m;
   m = n;
   n = r;
   r = h;

   /* versuche Multiplikation zu ersetzen */
   if (qh == 1)
      llsto(xs, xss);
   else
      llmul(q, xs, xss);

   llsub(x0, xss, xss);
   h = x0;
   x0 = xs;
   xs = xss;
   xss = h;
}

sto(n, erg);
if (minus(a))
   neg(x);

h_free_temp(xss0);
```

```
   h_free_temp(xs0);

   h_free_temp(m0);
   h_free_temp(n0);
   h_free_temp(rh0);
   h_free_temp(r0);

   return (0);
}




/* V001
kurze Multiplikation modulo
===========================

Funktion: int lsmulmod(x, y, n, erg)
          LONGTYPE x, n, erg;
          int y;

erg = (x * y) mod n

aktuell: 17.10.91 */


#include "larith.h"


lsmulmod(x, y, n, erg)
   LONGTYPE          x, n, erg;
   int               y;
{
   LONGTYPE          h;

   h = lalloc(lstellen(x) + 2);

   lsmul(x, y, h);
   llmod(h, n, erg);

   free(h);
   return (0);
}




/* V001
Kurze Addition modulo
=====================

Funktion: lsaddmod(x, y, n, erg)
          LONGTYPE x, n, erg;
          int y;

erg = (x + y) mod n

Vor:
|x|, |y| < n; n > 0
Die Modulo-Reduktion wird (falls noetig) durch eine Subtraktion mit n
realisiert. Deshalb muessen x und y schon vorher reduziert (d.h. < |n|) sein!!!

aktuell: 17.10.91 */

#include "larith.h"

lsaddmod(x, y, n, erg)
   LONGTYPE          x, n, erg;
   int               y;
{
   LONGTYPE          h;

   h = lalloc(lstellen(n) + 2);

   lsadd(x, y, h);
```

```
   if (llabscomp(h, n) != -1)   /* |h| >= |n| ? */
      if (llcomp(h, n) != -1)   /* h >= n ? */
         llsub(h, n, erg);
      else
         lladd(h, n, erg);
   else
      llsto(h, erg);

   free(h);
   return (0);
}
```

```
/* V001
Kurze Subtraktion modulo
========================

Funktion: lssubmod(x, y, n, erg)
          LONGTYPE x, n, erg;
          int y;

erg = (x - y) mod n

Vor:
|x|, |y| < n; n > 0
Die Modulo-Reduktion wird (falls noetig) durch eine Subtraktion mit n
realisiert. Deshalb muessen x und y schon vorher reduziert
(d.h. < |n|) sein!!!

aktuell: 12.7.91 */

#include "larith.h"

lssubmod(x, y, n, erg)
   LONGTYPE          x, n, erg;
   int               y;
{
   LONGTYPE          h;

   h = lalloc(lstellen(n) + 2);

   lssub(x, y, h);
   if (llabscomp(h, n) != -1)   /* |h| >= |n| ? */
      if (llcomp(h, n) != -1)   /* h >= n ? */
         llsub(h, n, erg);
      else
         lladd(h, n, erg);
   else
      llsto(h, erg);
   free(h);

   return (0);
}
```

```
/* llcompmod - compares 2 LONGTYPES modulo
5.6.92
*/

#include "larith.h"

llcompmod(z1, z2, n)
   LONGTYPE          z1, z2, n;
{
   int               rw, ldim = lstellen(n) + 2;
   LONGTYPE          h1, h2;

   h1 = lalloc(ldim);
   h2 = lalloc(ldim);

   llsto(z1, h1);
```

```
   llsto(z2, h2);

   if (minus(h1))
      lladd(h1, n, h1);
   if (minus(h2))
      lladd(h2, n, h2);

   rw = llcomp(h1, h2);

   free(h1);
   free(h2);
   return (rw);
}




/* 001
   06.11.91
   ltrialdiv, initltrialdiv, resetltrialdiv -

   Def: void initltrialdiv(limit)
        unsigned int limit;  // upper limit for trialdivision

   Def: unsigned int ltrialdiv(n)
        LONGTYPE n;
   Res: 1, if no factor found;
        factor of n, else

   Def: void resetltrialdiv(limit)
        unsigned int limit;

   Tries to find small factors by dividing n through primes < limit. If a
   factor was found, ltrialdiv returns with this factor. Another call of
   ltrialdiv will test the primes >= factor. If you want to test from the
   smallest prime again, call resetltrialdiv(limit), which resets ltrialdiv.

   Bugs:
   Attention: befor the first call of "ltrialdiv(n)" you must call
   "initltrialdiv(limit)".
   Example:    int factor;
               initltrialdiv(1000);
               do
               {
                  factor = ltrialdiv(n);
                  printf("%d\n", factor);
               }
               while (factor != 1);
   Prints all factors of n smaller than 1000.
*/


#include <stdio.h>
#include "allg.h"
#include "larith.h"
#include "larith+.h"

static unsigned int prime, limit;
static PRIMETABLE *ptp = NULL;

unsigned int      ltrialdiv(n)
   LONGTYPE          n;
{
   unsigned int      p, factor = 1;

   if (lsmod(n, prime) == 0)
      factor = prime;
   else
   {
      while ((prime = getprimes2(ptp)) < limit && (lsmod(n, prime)) != 0);
      if (prime < limit)
         factor = prime;
   }
   return (factor);
}

void              initltrialdiv(l)
```

```
   unsigned int      l;
{
   if (l > MAXPRIM)
      limit = MAXPRIM;
   else
      limit = l;
   prime = 2;
   if (ptp != NULL)
      exitprimes(ptp);
   ptp = initprimes(NULL);
}

void              resetltrialdiv(l)
   unsigned int      l;
{
   if (l > MAXPRIM)
      limit = MAXPRIM;
   else
      limit = l;
   prime = 2;
   resetprimes(ptp);
}
```

```
/*001
  06.11.91
  lvinvert - computes the invers modulo of each element of vector of LONGTYPES

  Def: Boolean lvinvert(a, N, erg, vsize)
       Vector_of_LONGTYPES a, erg;
       LONGTYPE N;
       int vsize;
  Res: TRUE, if all elements of the vector are inverted
       FALSE, otherwise

  Computes the invers of each vector-element with a trick of Pollard.
  Es wird nur ein einziges mal der ggt mit langer Darstellung berechnet.
  Die Inversionen modulo benoetigen dann fuer jedes Element nur noch drei
  Multiplikationen modulo n. D.h.: die asymptotische Kosten fuer die
  Inversion eines Elementes ist 3 Multiplikationen modulo.
*/

#include "ecmp-types.h"
#include "larith+.h"
#include "allg.h"

Boolean lvinvert(a, N, erg, vsize)
Vector_of_LONGTYPES a, erg;
LONGTYPE N;
int vsize;
{
   Boolean success = FALSE;
   Vector_of_LONGTYPES b;
   LONGTYPE h, inv;
   int i, LDIM;

   LDIM = lstellen(N)+2;
   if (vsize < 1) goto END;   /* you can't invert less than one number */
   if (vsize == 1)   /* do a normal inversion (without tricks) */
   {
      h = lalloc(LDIM);

      llgcd1(a[0], N, h, erg[0]); /* h = erg[0]*a[0] + y * N
                                     y will not be computed
                                     if h==1 then erg[0] = 1/a[0] */
      if (test1(h))
         success = TRUE;
      free(h);
   }
   else     /* do the Pollard-inversion */
   {
      b = lvalloc(vsize, LDIM);
      h = lalloc(LDIM);
```

```
      inv = lalloc(LDIM);

      llsto(a[vsize-1], b[vsize-1]);

      /* 1) multiply the elements of a and store the results in b */
      for (i=vsize-2; i>=0; i--)
         llmulmod(a[i], b[i+1], N, b[i]);  /* b[i] = a[i]+b[i+1] mod N */

      /* 2) compute the gcd mit langer Darstellung */
      llgcd1(b[0], N, h, inv);
      if (test1(h))                        /* if h==1 then inv = 1/b[0] */
      {

         /* 3) compute the invers a */
         for (i=0; i<=vsize-2; i++)
         {
            llmulmod(inv, b[i+1], N, h);    /* h= inv*b[i+1] mod N */
            llmulmod(inv, a[i], N, inv);    /* inv= inv*a[i] mod N */
            llsto(h, erg[i]);               /* erg[i]= h */
         }
         llsto(inv, erg[i]);
         success = TRUE;
      }
      lvfree(b); free(h); free(inv);
   }
END:
   return(success);
}
```

```
/* 001
   18.10.91
   lvinv_fails (lvinvert fails)
   Search the first number which is not invertable modulo.

   Def: int lvinv_fails(VL, n, factor)
        Vector_of_LONGTYPES VL;  // this numbers sollen invertiert werden
        LONGTYPE n,              // Modul
                 factor;         // divisor of n,
   Res: -1; if a divisor of n is found
         i; i >= 0, the i-th curve is not useable, because gcd(VL[i], n) = n
                    the i-th curve has to be deactivated
*/

#include "larith+.h"


int               lvinv_fails(VL, n, factor)
   Vector_of_LONGTYPES VL;
   LONGTYPE          n, factor;
{
   int               i = 0, res;

   do
   {
      llgcd0(VL[i], n, factor);
   }
   while (VL[++i] != NULL && test1(factor));

   if (test1(factor))
      res = i;                 /* factor =1 => you can use lvinvert */
   else if (llcomp(factor, n) == 0)
      res = i - 1;             /* factor = n => deactivate the (i-1)-th
                                  curve */
   else
      res = -1;                /* factor found */

   return (res);
}
```

```
/* V001
   22.10.91
   ulpower (unsigned long power)
   Computes a^b.

   Def: unsigned long ulpower(a, b)
        unsigned long a;
        int b;      // b >= 0
   Res: a^b

   Bugs: The result must be less then ULONGMAX.
*/

unsigned long     ulpower(a, b)
   unsigned long     a;
   int               b;
{
   int               i;
   unsigned long     erg = 1;

   for (i = 0; i < b; i++)
      erg *= a;

   return (erg);
}




/*
  Funktionen zur Laufzeitmessung von Programmteilen:
  ==================================================

  int start_timing(timer)
  int timer[2];

  int get_timing(timer)
  int timer[2];

  int cont_timing(timer)
  int timer[2]

  char* strtime(format, clock)
  char *format;
  unsigned long clock;       // Zeit in Sekunden!!


  strtime ermoeglicht die komfortable Ausgabe der gemessenen Zeit.
  clock ist die gemessene Zeit in Sekunden (ACHTUNG: nicht in 1/100-
  Sekunden, wie durch die timing-Funktionen geliefert).
  format ist ein String "ydhms" mit dem das Ausgabeformat festge-
  legt wird. Die Reihenfolge ist wichtig, es koennen aber beliebige
  Teile weggelassen werden. Dabei bedeutet:
  'y': year
  'd': day
  'h': hour
  'm': minute
  's': second
  Es wird immer in die groesste angegebene Einheit umgerechnet und ein
  String erzeugt, der die berechneten Werte, der angegebenen Einheiten,
  durch den entsprechenden Buchstaben und SPACE getrennt, enthaelt.
  z.B.: strtime("ydhms", 97772710) -> "3y 36d 15h 5m 10s "
        strtime("ydh", 97772710)   -> "3y 36d 15h "
        strtime("ys", 97772710)    -> "3y 3164710s "
        strtime("dh", 97772710)    -> "1131d 15h "
        strtime("m", 97772710)     -> "1629545m "
  Das Ergebnis strtime ist ein Pointer auf diesen String.
  Achtung: Jeder Aufruf von strtime ueberschreibt den vorherigen
           Ergebnisstring, was zur Folge hat, dass man strtime z.B.
           innerhalb einer printf-Anweisung nicht mehrmals benutzen kann.

*/

#include <stdio.h>
#include <sys/time.h>
```

```
#include <sys/resource.h>

/* Dies sind Routinen von Stefan Duellmann um die Laufzeit von */
/* Programmabschnitten zu messen.                              */

static struct rusage buffer;

/***********************************************************/

int              start_timing(timer)

   int              timer[2];

{
   getrusage(0, &buffer);
   timer[0] = buffer.ru_utime.tv_sec * 100
      + buffer.ru_utime.tv_usec / 10000;
   timer[1] = buffer.ru_stime.tv_sec * 100
      + buffer.ru_stime.tv_usec / 10000;
   return (0);
}

/************************************************************/

int              get_timing(timer)

   int              timer[2];

{
   getrusage(0, &buffer);
   timer[0] = buffer.ru_utime.tv_sec * 100
      + buffer.ru_utime.tv_usec / 10000
      - timer[0];
   timer[1] = buffer.ru_stime.tv_sec * 100
      + buffer.ru_stime.tv_usec / 10000
      - timer[1];
   return (0);
}

/**************************************************************/

int              cont_timing(timer)

   int              timer[2];

{
   get_timing(timer);
   return (0);
}

/*************************************************************/

char            *strtime(format, clock)
   char            *format;
   unsigned long    clock;     /* Zeit in Sekunden */
{
   int              i = 0, ready = 0;
   static char      buf[100];
   char             s1[20], s2[20], s3[20], s4[20], s5[20];

   s1[0] = s2[0] = s3[0] = s4[0] = s5[0] = '\0';

   do
   {
      switch (format[i++])
      {
      case 'y':
         sprintf(s1, "%dy ", clock / 31536000); /* Jahre */
         clock = clock - clock / 31536000 * 31536000;
         break;
      case 'd':
         sprintf(s2, "%dd ", clock / 86400);    /* Tage */
         clock = clock - clock / 86400 * 86400;
         break;
      case 'h':
         sprintf(s3, "%dh ", clock / 3600);     /* Stunden */
         clock = clock - clock / 3600 * 3600;
         break;
```

```
      case 'm':
         sprintf(s4, "%dm ", clock / 60);       /* Minuten */
         clock -= clock / 60 * 60;
         break;
      case 's':
         sprintf(s5, "%ds ", clock);    /* Sekunden */

      case '\0':
         ready = 1;
         break;
      }
   }
   while (!ready);

   sprintf(buf, "%s%s%s%s%s", s1, s2, s3, s4, s5);
   return buf;
}




/* 001
   16.01.92
   fopena - fopen with alternativ filenames

   Def: FILE* fopena(filenames, type)
        char *filenames, *type;
   Res: NULL, if fopena can't open any of the files in filenames
        pointer on a file, else

DESCRIPTION
     fopena() opens one of the files named by filenames and associates a
     stream  with it. If the open succeeds, fopena() returns a pointer to
     be used to identify the stream in subsequent operations.

     filenames points to a character string that contains the names
     of the files to be opened. The names must be seperated by a ' ' or
     a ':'.
     Example:
          fopena("file1 file2 file3 file4", "r");

          Tries to open first 'file1', if the open not succeeds fopena tries
          to open 'file2' then 'file3' and so on.

     type is a character  string  having  one  of  the  following
     values:

          r         open for reading

          w         truncate or create for writing

          a         append: open for writing at end of  file,  or
                    create for writing

          r+        open for update (reading and writing)

          w+        truncate or create for update

          a+        append; open or create for update at EOF
*/

#include <stdio.h>
#include "allg.h"
#define TRUE 1
#define FALSE 0

FILE             *fopena(filenames, type)
   char             *filenames, *type;
{
   char              p[MAXFILENAMELEN];
   int               i = 0, j = 0, ready = FALSE;
   FILE             *fp = NULL;

   do
   {
      switch (filenames[i])
      {
```

```
      case ':':
      case ' ':
      case '\0':
         p[j] = '\0';
         if ((fp = fopen(p, type)) != NULL)
            ready = TRUE;
         j = 0;
         break;
      default:
         p[j] = filenames[i];
         ++j;
         break;
      }
   }
   while (filenames[i++] != '\0' && !ready);
   return (fp);
}




/* J.B.
   Modul ueberprueft die Existenz der Datei, die als Argument uebergeben wird.
   Dabei wird nur in dem aktuellen Ordner gesucht, falls nicht der gesamte
   Pfadname uebergebenwird
   Aufruf: fexist(datei), dabei ist datei der gesamte Pfadname
   Ergebnis: 0, falls datei nicht existiert
             1, falls datei existiert
   5.10.90, 11.3.92 FDB */

#include'stdio.h'

int             fexist(pfad)
   char           *pfad;
{
   FILE           *fopen(), *fp;

   if ((fp = fopen(pfad, "r")) == NULL)
      return (0);
   else
   {
      fclose(fp);
      return (1);
   }
}




/*DOC
******************************************************************

Mips-Funktion
===============

Funktion:  float mips()
Die Funktion fuehrt auf dem aufrufenden Rechner hostid aus und versucht
aus den beiden ersten Ziffern die MIPS-Leistunf der Maschine zu bestimmen.
Ist dies nicht moeglich so wird 1 zurueckgeliefert.

Letzte Aenderung: 19.05.92

******************************************************************
DOC*/

float           mips()
{
   int             number;

   number = gethostid();
   number = number >> 24;
   switch (number)
   {
   case 0x24:
      return (22);
```

```
      break;
   case 0x51:
      return (12.80);
      break;
   case 0x52:
      return (15.80);
      break;
   case 0x53:
      return (15.70);
      break;
   case 0x54:
      return (12.40);
      break;
   case 0x55:
      return (28.50);
      break;
   case 0x56:
      return (21.0);
      break;
   case 0x57:
      return (28.5);
      break;
   case 0x71:
      return (27);
      break;
   case 0x04:                     /* ACHTUNG: das ist nicht sauber, da nur
                                     vermutet: sol */
      return (24);
      break;
   default:
      return (1);
      break;
   }
}
```

E.3 Include-Files

```
#define STOP_ECM "stop-ecm"
#define MINCDIM 1          /* min. Anzahl von Kurven, die aktiv sein muessen */
#define MAXFILENAMELEN 100 /* max. Anzahl Character in einem Filenamen */
#define STRATEGYFILES "ecm-strategy" /*Strategie-File */
#define PARAMETERFILE "ecm-parameters /home/berger/ECM-setup/ecm-parameters" /* Parameter-File */
#define MAXSTRATEGY 50     /* max. Anzahl der Eintragspaare im Strategy-File */

/* constants for networking */
#define HOSTNAMELEN 30
#define MAXDEZSTEL 1000     /* stringsize for biggest number send with LINDA */
#define MAXJOBS 1000        /* Groesse des Jobbuffers */

#define CS_MAX 15           /* max. Anzahl simultaner Kurven (kann um 2
                               ueberschritten werden) */
#define CS_MIN 15          /* min.   "        "       "   (kann um 2
                              unterschritten werden */

/* Pfad der Primzahltafel */
# define PRIMTAFEL "KOMP-PRIMES PRIMES /glob/staff/dat/PRIMES\
                    /home1/JB/roth/PRIMES"
#define MAXPRIM 9999991    /* groesste Primzahl in Primzahltafel */
#define MAXPRIMDIF 154     /* groesster Abstand zwischen zwei aufeinander-
                              folgenden Primzahlen in der PRIMTAFEL */
```

```
/* V001 */
/* 11.10.91 */
/* structs, types, constants and functions for working with elliptic curves
   and points.
   DO NOT CHANGE THIS FILE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */

# ifndef _ecmp-types_h
# define _ecmp-types_h

# ifndef _larith_h   /* larith muss unbedingt wegen LONGTYPE included werden */
# include "larith.h"
# endif

/*************************************************************************/
/* structs and types */

typedef enum {FALSE, TRUE} Boolean;

/*-----------------------------------------------------------------------*/
/* result of the functions epvadd, epvmul:
        hopeless: gcd(denominator, n) = n => Curve is useless
        no_factor_found: go on with computing
        factor_found: great! */

typedef enum {hopeless, no_factor_found, factor_found, new_message} Status;

/*-----------------------------------------------------------------------*/
/* elliptic curve
   E: y^2 = x^3+ax+b, E = (a, b);
   I need only the 'a' in my computings. b is saved for other applications */

   struct CURVE { LONGTYPE a, b;
                  Boolean  activ;
                };
   typedef struct CURVE *Curve;

/*-----------------------------------------------------------------------*/
/* Vector of elliptic curves:
   the last element of the vector must be a NULL pointer. */

typedef Curve *Vector_of_curves;

/*-----------------------------------------------------------------------*/
/* Point on a elliptic curve */

   struct POINT { LONGTYPE x, y;
```

```
                  Boolean  O;
                  Curve C;
                };
   typedef struct POINT *Point;

/*------------------------------------------------------------------------*/
/* Vector of points:
   the last element of the vector must be a NULL pointer */

typedef Point *Vector_of_points;

/**************************************************************************/

/* P == 0 ? */
#define eptestO(P) (P->O == TRUE)

/*------------------------------------------------------------------------*/
/* P = 0   set P on zero */
#define epsetO(P) P->O = TRUE

/*------------------------------------------------------------------------*/
/* delete the zero-mark of point P */
#define epunsetO(P) P->O = FALSE;

/**************************************************************************/

Curve ecalloc();
void ecfree();
void ecwrite();
Boolean ecgen();
void ecgenset();

Vector_of_curves ecvalloc();
void ecvfree();
Status ecvfindfactor();

Point epalloc();
void epfree();
void epwrite();
void epsto();
int epnegcomp();
int epcomp();
int eponcurve();

Vector_of_points epvalloc();
void epvfree();
int epvlength();
int epvactiv_curves();
Status epvadd();
Status epvmul();

Status ecm_stdcont();
Status ecm_stdcont2();
double ecm_parameters();
Boolean new_job();
#endif
```

```
/* V001
   16.10.91
Include-File larith+.h
Einige Funktionen, die einem das Leben mit der larith leichter machen. */
# ifndef _larith+_h
# define _larith+_h

# include <stdio.h>              /* I need it for the datatype FILE */
# include "larith.h"             /* I need it for the datatype LONGTYPE */
# include "ecmp-types.h"         /* I need it for the datatype Boolean */

#define test1(x)  ( (x) && (((x)[0] & 0xffff) == 1) && ( ((x)[1] == 1) ) )
#define set1(x)   { if (x) { (x)[0] = 1; (x)[1] = 1; } }

#define PRIMETABLE FILE
```

```
typedef LONGTYPE *Vector_of_LONGTYPES;
/*---------------------------------------------------------------------*/

Vector_of_LONGTYPES lvalloc();
void lvfree();

int llmulmod();
Boolean lvinvert();
int lvinv_fails();

PRIMETABLE* initprimes();
unsigned long getprimes();
unsigned long getprimes2();
void resetprimes();
void exitprimes();

void initltrialdiv();
unsigned int ltrialdiv();
void resetltrialdiv();

unsigned long ulpower();

char* strtime();
FILE *fopena();
float mips();
#endif
```

E.4 Sonstige Dateien

E.4.1 CLIENT_CONF

```
#      EXAMPLE OF A CLIENT_CONFIGURATION FILE
#
# Parameter 1: Internet Adress ( Name of the Remote machine where programm's
#              are to be executed)
# Parameter 2: Name of client machine in extended internet notation
#              ( crypt1.cs.uni-sb.de)
# Parameter 3: LiPS - Username ( User's name on this machine )
#              In future, there'll be an own home directory for LiPS.
#              Every LiPS user will have his own working directory beneath.
#              Example: user roth will have directory ~lips/roth as his
#              working directory. See also Parameter 4
# Parameter 4: Daytime ( Time ( in hour's ) process is allowed to operate )
# Parameter 5: Day's (as before Time (in day's) week startin with sunday)
# Parameter 6: Number of User's (maximal) on this machine
#              ( more user ==> no Processing time)
# Paramater 7: Machine is Menber of this cluster of NFS-Groups
#              (Send Binaries to each first working member of a cluster)
#-----------------------------------------------------------------
#
#
#This is the path where the local lipsdata are present
/home/lips
# Gipsy's establish the next cluster
134.96.228.2  gipsy1.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.3  gipsy2.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.4  gipsy3.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.5  gipsy4.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.6  gipsy5.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.7  gipsy6.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.8  gipsy7.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.9  gipsy8.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.10 gipsy9.cs.uni-sb.de     fberger 7 21 1000001 1  5
134.96.228.11 gipsy10.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.12 gipsy11.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.13 gipsy12.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.14 gipsy13.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.15 gipsy14.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.16 gipsy15.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.17 gipsy16.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.18 gipsy17.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.19 gipsy18.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.20 gipsy19.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.21 gipsy20.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.22 gipsy21.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.23 gipsy22.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.24 gipsy23.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.25 gipsy24.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.26 gipsy25.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.27 gipsy26.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.28 gipsy27.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.29 gipsy28.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.30 gipsy29.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.31 gipsy30.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.32 gipsy31.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.33 gipsy32.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.34 gipsy33.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.35 gipsy34.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.36 gipsy35.cs.uni-sb.de    fberger 7 21 1000001 1  5
134.96.228.37 gipsy36.cs.uni-sb.de    fberger 7 21 1000001 1  5
```

Literaturverzeichnis

[Atkin/Morain] A.O.L. Atkin, F. Morain: *Finding Suitable Curves For The Elliptic Curve Method Of Factorization* / Preprint March 1991

[Brent] R.P. Brent: *Some Integer Factorization Algorithms using Elliptic Curves* / Australian Computer Science Communications 8 (1986), S. 149-163

[Bundschuh] Peter Bundschuh: *Einführung in die Zahlentheorie*

[Denny] Thomas Denny: *Faktorisieren mit dem quadratischen Sieb* / Diplomarbeit; Universität des Saarlandes 1993

[Horster] Patrik Horster: *Kryptologie* / B.I. Wissenschaftsverlag, Reihe Informatik/47

[Knuth] Donald E. Knuth: *The Art of Computer Programming Vol.2* / Addison Wesley

[Knuth/Pardo] Donald E. Knuth and Luis Trabb Pardo: *Analysis of a simple Factorization Algorithm* / Theoretical Computer Science 3 (1976), S. 321-348

[Koblitz] Neal Koblitz: *A Course in Number Theory and Cryptography* / Springer-Verlag

[Lamprecht] E. Lamprecht: *Einführung in die Algebra* / UTB Birkhäuser

[larith] Stefan Düllmann: Langzahlarithmetik

[Dixon/A. Lenstra] B. Dixon, A.K. Lenstra: *Massively parallel elliptic curve factoring* / persönliche Kommunikation

[Lenstra] H. W. Lenstra, Jr.: *Factoring integers with elliptic curves* / Annals of Mathematics, 126 (1987), S. 649-673

[LIPS] Ralf Roth, Thomas Setz: *LIPS - A System For Distributed Processing On Workstations* / Dokumentation (11.11.1992)

[Mathematica] Stephen Wolfram: *Mathematica - A System for Doing Mathematics by Computer* / Addison Wesley 19334

[Montgomery] Peter L. Montgomery: *Speeding the Pollard and Elliptic Curve Methods of Factorization* / Mathematics of Computation, Vol. 48, Number 177, Jan. 1987, pp. . 243-264

[Montgomery2] Peter L. Montgomery: *An FFT Extension of the Elliptic Curve Method of Factorization* / Dissertation; University of California 1992

[Rannacher] Rolf Rannacher: *Praktische Mathematik, Vorlesungsskriptum zum Wintersemester 1987/88* / Universität des Saarlandes

[Riesel] Hans Riesel: *Prime Numbers and Computer Methods for Factorization* / Boston; Basel; Stuttgart : Birkhäuser, 1985 (Progress in mathematics; 57)