Theorie und Implementation von parallelisierten Pseudozufallszahlengeneratoren

Forschungsbeleg von

Heiko J. Bauke

korrigierte Version vom 29. Juni 2001

Institut für Theoretische Physik
Statistische Physik und nichtlineare Dynamik

Magdeburg 2000–2001

Theorie und Implementation von parallelisierten Pseudozufallszahlengeneratoren

Inhaltsverzeichnis

Listings

Symbole und Bezeichnungen

Folgende Tabelle beschreibt im Text verwendete mathematische Symbole.

Symbol	Bedeutung
$\mathbb{N}$	Menge der natürlichen Zahlen $1, 2, 3, \ldots$
$\mathbb{N}_0$	Menge der natürlichen Zahlen und die Null $0, 1, 2, 3, \ldots$
$\mathbb{Z}$	Menge der ganzen Zahlen $0, \pm 1, \pm 2, \pm 3, \ldots$
$\mathbb{N}_P$	Menge der Primzahlen
$\mathbb{Z}/m\mathbb{Z}$	Menge der Restklassen mod m
$\mathbb{R}$	Menge der reellen Zahlen
$x = \lfloor a \rfloor$	x ist die größte ganze Zahl, für die $x \leq a$ gilt.
$\left\{ {k \atop r} \right\}$	steht für eine STIRLINGsche Zahl
$a \equiv b \mod m$	a und b lassen bei der Division durch m den selben Rest
$x \equiv \overline{a} \mod m$	äquivalent zu $1 \equiv x \cdot a \mod m$
$\gcd(a, b)$	größter gemeinsamer Teiler von a und b
ord G	die Ordnung (Anzahl der Elemente) einer Gruppe G

Kapitel 1

Einleitung

Zufallszahlen mit einer deterministischen Maschine erzeugen zu wollen, muss auf den ersten Blick absurd erscheinen. Trotzdem haben sich gewisse Verfahren zur Erzeugung von Zahlenfolgen, die sich in bestimmten Anwendungen verhalten, als wären sie das Ergebnis eines stochastischen Prozesses, als ein in der Praxis sehr wertvolles Werkzeug erwiesen.

In der Kern- und Elementarteilchenphysik, der statistischen Physik und in Operations Research sind so genannte Monte-Carlo-Methoden schon seit langem ein etabliertes Hilfsmittel, wenn sich das zu lösende Problem einer analytischen Lösung entzieht. In der Informatik können Pseudozufallszahlengeneratoren dazu genutzt werden, einen Strom von Eingabedaten zum Testen von Algorithmen zu erzeugen. Da in letzter Zeit der elektronische Zahlungsverkehr immer mehr an Bedeutung gewinnt, kommen Pseudozufallszahlengeneratoren auch verstärkt in der Kryptographie zur Anwendung[1].

Aus welcher Motivation heraus versucht man aber, „zufällige“ Zahlenfolgen algorithmisch zu erzeugen, warum versucht man nicht, Quellen echten Zufalls zu nutzen? Tatsächlich wurde und wird dies getan. So vertreibt z. B. die amerikanische Firma ComScire [ComScire 2000] eine Hardwareeinheit, die aus dem thermischen Rauschen eines Widerstandes einen Strom zufälliger Bits erzeugt, der von einem Computer eingelesen werden kann. Es sind auch ähnliche Geräte im Gebrauch, die den radioaktiven Zerfall als stochastischen Prozess nutzen. Solche Geräte haben aber Nachteile. Einer ist, so paradox es klingt, die echte Zufälligkeit der gewonnen Zahlenfolge. Das heißt, diese lässt sich nicht reproduzieren. Man hat keine Möglichkeit, mit zwei verschiedenen Algorithmen die gleiche Folge von Zufallszahlen zu verarbeiten. Dies erschwert den Vergleich dieser Algorithmen oder macht ihn ganz unmöglich. Man könnte versuchen, diesen Nachteil zu umgehen und die Zahlenfolge zu speichern. Dabei entstehen aber schnell extrem große Datenmengen, die nur schwer zu handhaben sind. MARSAGLIA [Marsaglia 1995] vertreibt eine CD-

[1] Die in dieser Arbeit vorgestellten Pseudozufallszahlengeneratoren sind für kryptographische Anwendungen nicht geeignet. Informationen über Pseudozufallszahlengeneratoren in der Kryptographie findet man z. B. in [Menezes u. a. 1996].

ROM mit Zufallszahlen. Zur Erzeugung kombinierte er algorithmische Zufallszahlengeneratoren und Quellen weißen Rauschens und, wie er es nennt, schwarzen Rauschens (amerikanische Hip-Hop-Musik). In vielen praktischen Anwendungen werden aber heute so große Zahlenfolgen benötigt, so dass die 4,8 Milliarden Zufallsbits auf der CD-ROM den Anforderungen nicht ansatzweise gerecht werden können. Ein weiterer Nachteil ist, dass das Auslesen der gespeicherten Zahlenfolgen bzw. der Hardwareeinheit unverhältnismäßig lange dauert. Der Generator von ComScire liefert gerade 10 000 Bits je Sekunde. Der Wunsch nach Reproduzierbarkeit und der Erzeugung sehr großer Zahlenfolgen in kurzer Zeit, machen es notwendig, sich mit algorithmischen Zufallszahlengeneratoren zu beschäftigen.

In dieser Arbeit sollen Methoden zur Generierung von Pseudozufallszahlenfolgen[2] dargestellt werden. Insbesondere ihre Implementierung auf Parallelrechnern ist hier von zentralem Interesse.

Die eher philosophische Frage nach dem Wesen des Zufalls soll hier nicht erläutert werden. Statt dessen wird ein eher pragmatischer Weg eingeschlagen und die Generatoren verschiedenen statistischen Tests unterzogen, um die „Zufälligkeit“ der erzeugten Zahlenfolgen zu beurteilen. Zu den philosophischen Aspekten des Zufalls sei der Leser auf [Knuth 1981a], [May 1997] oder [Gell-Mann 1996] verwiesen.

[2] Im weiteren Verlauf des Textes wird die Bezeichnung Zufallszahlen oder Zufallszahlenfolge verwendet, auch wenn die Zahlen(-folgen) algorithmischen Ursprungs sind und streng genommen als Pseudozufallszahlen(-folgen) bezeichnet werden müssten.

Mathematische Hilfsmittel; Ein wenig Zahlentheorie; GALOIS-Felder

Eins, zwei, drei, vier, fünf, sechs, sieben, acht.
Uno, due, tre, quattro.
Uno, dos, tres, cuatro.
One, two.
一二三四
Один, два, три.

KRAFTWERK,
≫Nummern≪

Die mathematischen Grundlagen der Methoden zur Erzeugung von Pseudozufallszahlen liegen in der Zahlentheorie und der Theorie der GALIOS-Felder, deshalb sollen hier einige Grundzüge dieser mathematischen Disziplinen skizziert werden. Wenn nichts anderes gesagt, sollen alle Zahlen in diesem Kapitel als aus der Menge der ganzen Zahlen $\mathbb{Z}$ stammend angenommen werden. Manchmal stammen die Zahlen auch aus der Menge der natürlichen Zahlen $\mathbb{N}$ bzw. $\mathbb{N}_0$ ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) oder der Primzahlen $\mathbb{N}_P$.

2.1 Definitionen

Zunächst werden einige Begriffe aus der Algebra benötigt, die hier zunächst definiert werden sollen.

Definition 2.1 Eine nicht leere Menge H und eine assoziative innere Verknüpfung $\circ$ auf H bilden eine *Halbgruppe* $(H, \circ)$. Eine *Gruppe* $(G, \circ)$ ist eine Halbgruppe, die bzgl. der Verknüpfung $\circ$ ein neutrales Element besitzt und in der jedes Element invertierbar ist. Das heißt, zu jedem $a \in G$ existiert ein $b \in G$, so dass $a \circ b = b \circ a = e$ ist. Wobei e das neutrale Element der (Halb-)Gruppe ist. Die Gruppe heißt kommutativ, wenn die Halbgruppe kommutativ ist. Die Anzahl der Elemente einer (Halb-)Gruppe G heißt *Ordnung* der (Halb-)Gruppe G. Sie wird mit ord G bezeichnet.

Definition 2.2 Wenn $(R, +)$ eine kommutative Gruppe und $(R, \cdot)$ eine Halbgruppe ist, so ist das Tripel $(R, +, \cdot)$ ein *Ring*, wenn zusätzlich die Distributivgesetze $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ und $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$ gelten. Der Ring heißt kommutativ, wenn $(R, \cdot)$ kommutativ ist. Das Einselement bzgl. der Operation $+$ heißt hier *Nullelement*. Die Operation $+$ wird Addition und die Operation $\cdot$ wird Multiplikation genannt.

Definition 2.3 Sei K ein Ring, so teilt $a \in K$ $n \in K$ genau dann, wenn ein $b \in K$ existiert, so dass $a \cdot b = n$. Wenn n das Einselement ist, so heißt b das *Inverse* zu a. Ein kommutativer Ring mit Einselement heißt *Körper*, wenn jedes Element außer dem Nullelement ein Inverses besitzt.

Die reellen Zahlen bilden mit der gewöhnlichen Addition und der Multiplikation einen Körper mit unendlich vielen Elementen. Körper mit endlich vielen Elementen heißen auch GALOIS-Felder $GF(q)$, wobei q der Anzahl der Körperelemente entspricht. Es lässt sich zeigen, dass q immer eine Primzahl bzw. eine Potenz einer Primzahl ist.

2.2 Teilbarkeit

Ein zentraler Begriff der Zahlentheorie ist der der Teilbarkeit.

Definition 2.4 Die ganze Zahl a *teilt* n genau dann, wenn eine ganze Zahl b existiert, so dass $n = ab$. Man schreibt kurz $a|n$ und $a \nmid n$ wenn a n nicht teilt. a ist ein *Teiler* von n.

Zum Beispiel teilt 7 56, da $7 \cdot 8 = 56$ ist, und -3 teilt 15, weil $(-3) \cdot (-5) = 15$ ist. Nach obiger Definition teilt jede Zahl a die Zahl 0, denn es gilt $a \cdot 0 = 0$ für alle a.

Satz 2.1 Für alle $a \in \mathbb{Z}$ und $b \in \mathbb{N}$ gibt es eindeutig bestimmte $p, r \in \mathbb{N}$, so dass $a = pb + r$ und $0 \leq r < b$ ist. Diese p und r sind $p = \lfloor \frac{a}{b} \rfloor$ und $r = a - bp$.

p heißt ganzzahliger Quotient und r Rest. Man schreibt kurz $r \equiv a \mod b$. $x = \lfloor a \rfloor$ heißt, x ist die größte ganze Zahl, für die $x \leq a$ gilt.

2.3 Primzahlen

Ein anderer wichtiger Begriff der Zahlentheorie, der im engen Zusammenhang zur Teilbarkeit steht, ist der der Primzahl.

Definition 2.5 Eine natürliche Zahl $p > 1$ heißt *Primzahl*, wenn sie ausschließlich 1 und sich selbst und 1 als Teiler hat. Die Menge aller Primzahlen wird mit $\mathbb{N}_P$ bezeichnet.

Satz 2.2 Jede natürliche Zahl $a > 1$ hat einen Primteiler.

Beweis: a besitzt mindestens einen Teiler > 1, sich selbst. Unter allen Teilern von a, die > 1 sind, sei p der kleinste. p muss prim sein, denn sonst besäße p einen Teiler b mit

$$1 < b < p \leq a\,,$$

was der Annahme, p sei der kleinste Teiler von a der > 1 ist, widerspricht. *q. e. d.*

Eine direkte wichtige Folge des Satzes 2.2 ist der folgende Satz:

Satz 2.3 Jede natürliche Zahl $a > 1$ kann eindeutig (bis auf die Reihenfolge der Faktoren) als Produkt von Primzahlen geschrieben werden.

Beweis: Der Beweis erfolgt durch vollständige Induktion.

- Für $a = 2$ ist nichts zu beweisen, da 2 prim.
- Falls $a > 2$ hat a nach Satz 2.2 einen Primteiler. Wenn $\frac{a}{p} = 1$, ist $a = p$ und der Satz ist bewiesen. Andernfalls ist $\frac{a}{p} > 1$ und $\frac{a}{p}$ ist nach Induktionsvoraussetzung ein Produkt von Primzahlen.

Bleibt die Eindeutigkeit zu zeigen. Angenommen es gäbe zwei verschiedene Primfaktorzerlegungen $a = p_1 \dots p_k$ und $a = q_1 \dots q_l$. Dann muss p_1 mit einem der Primfaktoren $q_1, \dots q_k$ übereinstimmen[1]. Nach einer Umnummerierung ist $p_1 = q_1$. Da nach Induktionsanfang sich auch $\frac{a}{p_1} = \frac{a}{q_1}$ eindeutig zerlegen lässt, ist $k = l$ und $p_i = q_i$ für alle $1 \leq i \leq k$. *q. e. d.*

[1] Den Beweis dieser Aussage kann man in vielen Lehrbüchern der Zahlentheorie finden , z. B. bei [Remmert und Ullrich 1995] oder auch in [Buchmann 1999], woher auch der Rest des Beweises stammt.

2.4 Kongruenzen und Restklassenringe

Definition 2.6 a ist *kongruent* zu b modulo m ($a \equiv b \mod m$), wenn m die Zahl $(a - b)$ teilt, was gleichbedeutend damit ist, dass ein $k \in \mathbb{Z}$ existiert, so dass $a = b + km$, bzw. a und b beim Teilen durch m den selben Rest lassen.

Die Kongruenz ist eine Äquivalenzrelation auf der Menge der ganzen Zahlen, denn es gelten folgende Aussagen.

- Jede Zahl ist zu sich selbst kongruent modulo m, $a \equiv a \mod m$. (Reflexivität)
- Aus $a \equiv b \mod m$ folgt $b \equiv a \mod m$. (Symmetrie)
- Aus $a \equiv b \mod m$ und $b \equiv c \mod m$ folgt $a \equiv c \mod m$. (Transitivität)

Man erhält die Äquivalenzklasse von a (oder auch Restklasse von $a \mod m$) durch Addition ganzzahliger Vielfacher von m zu a.

$$\{b : b \equiv a \mod m\} = a + m\mathbb{Z} \tag{2.1}$$

Jede Restklasse hat einen Repräsentanten a mit $0 \leq a < m$. Diese Repräsentanten bilden das so genannte Standardrestsystem. Die Menge aller Restklassen mod m wird mit $\mathbb{Z}/m\mathbb{Z}$ bezeichnet.

Satz 2.4 Für das Rechnen mit Restklassen gelten folgende Regeln: Wenn $a \equiv b \mod m$ und $c \equiv d \mod m$, so gelten auch

- $a + c \mod m \equiv (a \mod m) + (c \mod m) \mod m$,
- $a \cdot c \mod m \equiv (a \mod m) \cdot (c \mod m) \mod m$,
- $-a \equiv -b \mod m$,
- $a + c \equiv b + d \mod m$ und
- $ac \equiv bd \mod m$.

Beispiel: Seien $m = 7$, $a = 1$, $b = 15$, $c = 5$ und $d = 26$, dann gilt offenbar:

$$\begin{aligned} 1 + 5 &\equiv 6 \mod 7 \\ 15 + 26 \equiv 41 &\equiv 6 \mod 7 \\ 1 \cdot 5 &\equiv 5 \mod 7 \\ 15 \cdot 26 \equiv 390 &\equiv 5 \mod 7 \end{aligned}$$

Definition 2.7 Auf der Menge $\mathbb{Z}/m\mathbb{Z}$ lassen sich eine *Addition* und eine *Multiplikation* definieren. Die Summe der Restklassen $a + m\mathbb{Z}$ und $b + m\mathbb{Z}$ ist $(a + m\mathbb{Z}) + (b + m\mathbb{Z}) = (a + b) + m\mathbb{Z}$ und ihr Produkt $(a + m\mathbb{Z}) \cdot (b + m\mathbb{Z}) = (a \cdot b) + m\mathbb{Z}$.

Die Definition von Addition und Multiplikation der Restklassen erfolgt also über Vertreter der Restklassen, ist aber wegen Satz 2.4 von den konkreten Vertretern unabhängig. Da die Addition bzw. die Multiplikation der Restklassen assoziativ und kommutativ sind, bilden die Addition und die Multiplikation zusammen mit der Menge $\mathbb{Z}/m\mathbb{Z}$ jeweils eine kommutative Halbgruppe $(\mathbb{Z}/m\mathbb{Z}, +)$ bzw.$(\mathbb{Z}/m\mathbb{Z}, \cdot)$. Die Assoziativität und die Kommutativität der Addition bzw. der Multiplikation der Restklassen ist eine Folge der Assoziativität und der Kommutativität der Addition bzw. der Multiplikation der ganzen Zahlen.

Die Menge $\mathbb{Z}/m\mathbb{Z}$ besitzt bzgl. der Addition das neutrale Element $m\mathbb{Z}$, denn $(a + m\mathbb{Z}) + (m\mathbb{Z}) = (m\mathbb{Z}) + (a + m\mathbb{Z}) = (a) + m\mathbb{Z}$, und bzgl. der Multiplikation das neutrale Element $1 + m\mathbb{Z}$, denn $(a + m\mathbb{Z}) \cdot (1 + m\mathbb{Z}) = (1 + m\mathbb{Z}) \cdot (a + m\mathbb{Z}) = (a) + m\mathbb{Z}$.

Wenn $a \equiv b \mod m$ gilt, so folgt daraus im allgemeinen nicht, dass auch $a \equiv b \mod m'$ gilt, wenn $m \neq m'$. So lassen zwar 10 und 18 beim Teilen durch 8 den selben Rest 2, wenn man jedoch diese Zahlen durch 7 teilt, so findet man für 10 den Rest 3 und für 18 den Rest 4.

Satz 2.5 Wenn m' ein Teiler von m ist $(m'|m)$, so folgt aus $a \equiv b \mod m$ auch $a \equiv b \mod m'$.

Beweis: Aus $m'|m$ und $m|(a - b)$ folgt $m'|(a - b)$ und damit die Behauptung. *q. e. d.*

Die modulare Multiplikation ist nicht immer umkehrbar. Betrachten wir folgende Multiplikationstabelle.

	a	1	2	3	4	5	6	7	8
$3a \mod 9$		3	6	0	3	6	0	3	6

Offenbar hat die 3 in $\mathbb{Z}/9\mathbb{Z}$ kein Inverses. Wann hat die Kongruenz

$$ax \equiv 1 \mod m \tag{2.2}$$

aber eine (eindeutige) Lösung? Der folgende Satz beantwortet diese Frage.

Satz 2.6 Die Restklasse $a + m\mathbb{Z}$ ist genau dann in $\mathbb{Z}/m\mathbb{Z}$ invertierbar (Gleichung (2.2) besitzt eine Lösung.), wenn der größte gemeinsame Teiler von a und m $\gcd(a, m)$ eins ist. Das Inverse ist eindeutig bestimmt.

Beweis: Wenn $g = \gcd(a, m)$ und x eine Lösung von Gleichung (2.2), dann ist g ein Teiler von m und somit auch Teiler von $ax - 1$. Da g aber auch Teiler von a ist, muss g auch Teiler von 1 sein. Da g positiv ist, kann nur $g = 1$ sein. Wenn

umgekehrt $g = 1$ ist, so existieren Zahlen x und y, so dass $ax + my = 1$[2], d. h. $ax - 1 = -my$. Also muss x eine Lösung der Kongruenz (2.2) sein. *q. e. d.*

Z. B. gilt $7 \cdot 13 \equiv 91 \equiv 1 \mod 18$. 7 ist also das Inverse modulo 18 zu 13 und umgekehrt, $\gcd(7, 18) = \gcd(13, 18) = 1$.

Aus dem Satz 2.6 folgt unmittelbar, dass, wenn für alle k mit $1 \leq k < m$ $\gcd(k, m) = 1$ ist, $\mathbb{Z}/m\mathbb{Z}$ ein Körper ist. Diese Bedingung ist genau dann erfüllt, wenn m eine Primzahl ist. Außerdem bilden die Elemente von $\mathbb{Z}/m\mathbb{Z}$ ohne das Nullelement bzgl. der Multiplikation eine kommutative Gruppe der Ordnung $m - 1$. Besonders die Tatsache, dass alle Elemente von $\mathbb{Z}/m\mathbb{Z}$ ein eindeutiges Inverses besitzen, falls m prim ist, wird später für die Erzeugung von Pseudozufallszahlen wichtig sein.

2.5 Sätze von LAGRANGE, FERMAT und GAUSS

Definition 2.8 Besitzt eine Gruppe G eine nichtleere Teilmenge H, die selbst wieder eine Gruppe mit der Gruppenmultiplikation von G ist, so ist H eine *Untergruppe* von G.

Das Einselement von G ist Element jeder Untergruppe H. Das Einselement und G sind triviale Untergruppen von G. Im folgenden seien immer nichttriviale Untergruppen gemeint.

Für eine endliche Gruppe G der Ordnung $\operatorname{ord} G$ macht der Satz von LAGRANGE eine Aussage über die Ordnungen möglicher Untergruppen.

Satz 2.7 Die Ordnung $\operatorname{ord} H$ jeder Untergruppe H einer endlichen Gruppe G ist ein Teiler der Ordnung $\operatorname{ord} G$ der Gruppe G.

Aus dem Satz folgt, dass $\operatorname{ord} G$ ein Vielfaches von $\operatorname{ord} H$ sein muss. Den Beweis der Aussage findet man in vielen Büchern zur Gruppentheorie, z. B. in dem Buch von [Lucha und Schöbel 1993].

Definition 2.9 Eine Gruppe G der Ordnung $\operatorname{ord} G$ heißt *zyklisch*, wenn es ein Element $g_0 \in G$ gibt, so dass $G = \{g_0^1, g_0^2, \ldots, g_0^{\operatorname{ord} G}\}$ ist. g_0 heißt *erzeugendes Element* der Gruppe, $g_0^{\operatorname{ord} G}$ kann man mit dem Einselement der Gruppe identifizieren.

Gibt es ein $n < \operatorname{ord} G$ mit $g_0^n = e$, wobei e das Einselement der Gruppe G sei, so erzeugt g_0 nur eine zyklische Untergruppe von G.

[2] Den Beweis dieser Aussage findet man wieder in Lehrbüchern der Zahlentheorie, z. B. bei [Remmert und Ullrich 1995].

Satz 2.8 Ist N die Ordnung der Gruppe G und e deren Einselement, so gilt für jedes Element $x \in G$ die Beziehung $x^N = e$.

Beweis: Wenn X die durch x erzeugte zyklische Untergruppe von G, so muss es nach dem Satz von LAGRANGE ein $r \in \mathbb{N}$ geben, so dass ord $G = r$ ord X. Daraus folgt $x^{\text{ord } G} = x^{r \text{ ord } X} = (x^{\text{ord } X})^r = e^r = e$. *q. e. d.*

Aus diesem Satz folgt der (kleine) Satz von FERMAT.

Satz 2.9 Ist m eine Primzahl und a eine nicht durch m teilbare ganze Zahl, so gilt $1 \equiv a^{m-1} \mod m$.

Beweis: Der Beweis folgt unmittelbar aus Satz 2.8, wenn man a als Element der multiplikativen Gruppe $\mathbb{Z}/m\mathbb{Z}$ der Ordnung $m-1$ auffasst. *q. e. d.*

Eine wichtige Folge des Satzes von FERMAT ist, dass a^{m-2} das multiplikative Inverse modulo m zu a ist. Denn $1 \equiv a^{m-1} \equiv a \cdot a^{m-2} \mod m$.

Auf GAUSS geht der folgende Satz über die multiplikative Gruppe $\mathbb{Z}/m\mathbb{Z}$ zurück.

Satz 2.10 Sei m eine Primzahl, dann ist die multiplikative Gruppe $\mathbb{Z}/m\mathbb{Z}$ zyklisch.

Den Beweis findet man z. B. in [Forster 1996]. Das erzeugende Element der Gruppe $\mathbb{Z}/m\mathbb{Z}$ heißt hier auch Primitivwurzel. Ob ein $x \in \mathbb{Z}/m\mathbb{Z}$ eine Primitivwurzel modulo m ist, kann man natürlich leicht testen indem man alle $x^k \mod m$ $(1 < k < m-1)$ berechnet. Es geht aber effektiver.

Satz 2.11 Sei $m > 2$ eine Primzahl und besitze $m-1$ die Primfaktoren $p_1, p_2, \ldots, p_r$, so ist eine ganze Zahl x mit $m \nmid x$ genau dann eine Primitivwurzel modulo m, wenn $1 \not\equiv x^{\frac{m-1}{p_i}} \mod m$ für alle $i = 1, 2, \ldots, r$ ist.

Beweis: Dass die Bedingung notwendig ist, ist offensichtlich. Sei n die Ordnung der von x erzeugten Untergruppe von $\mathbb{Z}/m\mathbb{Z}$. Nach dem Satz 2.7 gilt $n|(m-1)$. Wäre $n < m-1$ (statt $n = m-1$), so gäbe es eine Primzahl p_i mit $p_i|(m-1)$, so dass $n|\frac{m-1}{p_i}$ gilt. Daraus würde aber $x^{\frac{m-1}{p_i}} \equiv 1 \mod m$ folgen würde, was im Widerspruch zur Voraussetzung steht. Also hat die von x erzeugte Gruppe die Ordnung $m-1$ und x ist eine Primitivwurzel. *q. e. d.*

2.6 GALOIS-Felder

Oben wurde gezeigt, dass $\mathbb{Z}/m\mathbb{Z}$ ein Körper ist, wenn m prim ist, dieser Körper werde nun mit $\mathbb{Z}/p\mathbb{Z}$ bezeichnet. Man kann aber auch Körper über Mengen konstruieren, deren Ordnung eine Potenz einer Primzahl ist. Diese endlichen Körper heißen auch GALOIS-Felder. Dazu nutzt man Polynome $K[x]$ vom Grade n in der Variablen x mit Koeffizienten aus einem Körper K.

$$f(x) = \sum_{i=0}^{n} a_i x^i \qquad g(x) = \sum_{i=0}^{l} b_i x^i \qquad a_i, b_i \in K \tag{2.3}$$

Man kann dann über diese Polynome eine Addition und eine Multiplikation definieren.

$$(f+g)(x) = \sum_{i=0}^{n} (a_i + b_i) x^i \tag{2.4}$$

$$(fg)(x) = \sum_{i=0}^{n+l} c_{n+l} x^i \qquad c_i = \sum_{k=0}^{i} a_i b_{k-i} \tag{2.5}$$

Die Addition bzw. die Multiplikation der Koeffizienten erfolgt im Körper K, wenn also $K = \mathbb{Z}/p\mathbb{Z}$ modulo p. Für zwei Polynome $a(x)$ und $b(x) \in K[x]$ existieren eindeutig bestimmte Polynome $p(x)$ und $q(x) \in K[x]$, so dass $a(x) = q(x)b(x) + r(x)$ und $r(x)$ einen kleineren Grad besitzt als $b(x)$, wenn $b(x) \neq 0$. Auf diese Weise kann man nun in Analogie zu den ganzen Zahlen eine modulare Arithmetik für Polynome und den Restklassenring $K[x]/((q(x))$ einführen. Man schreibt $a(x) \equiv b(x) \mod q(x)$, wenn beide Polynome bei der Polynomdivision durch $q(x)$ den gleichen Rest lassen.

Der Restklassenring $K[x]/((f(x))$ über dem Körper K ist genau dann ein Körper, wenn $f(x)$ irreduzibel in K ist, das heißt, dass $f(x)$ nicht invertierbar ist und dass in jeder Zerlegung $f(x) = a(x)b(x)$ ($a(x)$ und $b(x) \in K[x]$) ein Faktor invertierbar ist. Für $K = \mathbb{Z}/m\mathbb{Z}$ bedeutet das, dass keine Zerlegung $f(x) = a(x)b(x)$ ($a(x)$ und $b(x) \in \mathbb{Z}/m\mathbb{Z}[x]/((f(x)))$ existiert. In [Cigler 1995] werden diese Aussagen vertieft und bewiesen.

Wenn sich außerdem durch sukzessives Potenzieren von x alle Polynome außer dem Nullpolynom aus $K[x]/((f(x))$ aufbauen lassen, x also ein Erzeugendes der multiplikativen Gruppe $K[x]/((f(x))$ ist, dann heißt das Polynom $f(x)$ primitiv. Alle Potenzen von x zu bilden, um zu sehen, ob alle alle Elemente der Gruppe erzeugt werden, ist in der Praxis sehr umständlich. Man kann aber zeigen, ein Polynom

$$f(x) = x^k - \sum_{i=1}^{k} a_i x^{k-i} \mod m \tag{2.6}$$

ist ein primitives Polynom modulo m ($m \in \mathbb{N}_P$) genau dann, wenn mit $r = \frac{m^k-1}{m-1}$ folgende Bedingungen erfüllt sind:

- Für alle Primfaktoren q von $m-1$ gilt $1 \not\equiv ((-1)^{k+1}a_k)^{\frac{m-1}{q}} \mod m$, das heißt, $(-1)^{k+1}a_k$ ist eine Primitivwurzel modulo m.
- $(x^r \mod f(x)) \mod m = ((-1)^{k+1}a_k) \mod m$
- Das Polynom $(x^{\frac{r}{q}} \mod f(x)) \mod m$ ist vom Grade größer Null für alle Primfaktoren q von r.

Methoden zur Generierung von (Pseudo-) Zufallszahlenfolgen auf Mikrorechnern

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number — there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.
>
> JOHN VON NEUMANN,
>
> ≫Various Techniques Used in Connection with Random Digits.≪

Von John von NEUMANN stammt eine der ersten algorithmischen Verfahren zur Erzeugung von Zufallszahlenfolgen [Newman und Odell 1971], die Methode der Mittelquadrate. Man nehme eine Zahl mit n dezimalen Ziffern und quadriere sie. Die mittleren n Ziffern des $2n$-stelligen Quadrats werden dann als Zufallszahl betrachtet. Die n mittleren Ziffern des Quadrates dieser Zahl wiederum bilden das nächste Glied in der Zufallszahlenfolge und so weiter. KNUTH [Knuth 1981a] weist darauf hin, dass dieser Generator abhängig vom Startwert durchaus brauchbare Folgen liefern *kann*, aber meist in relativ kurze periodische Folgen läuft. Vorallem da dieser Generator einer theoretischen Analyse nur schwer zugänglich ist, hat er heute keine praktische Bedeutung mehr. Einen guten Überblick über aktuelle Techniken der Pseudozufallszahlengenerierung kann man z. B. bei [L'Ecuyer 1998] oder [Marsaglia 1984] erhalten.

In diesem Kapitel sollen Methoden zur Generierung von Zufallszahlenfolgen beschrieben werden und soweit möglich Aussagen über deren Qualität und Periodenlänge getroffen werden. Alle hier vorgestellten Generatoren erzeugen Zahlenfolgen $r_1, r_2, r_3, \ldots$, $r_i \in \mathbb{N}_0$, $0 \leq r_i \leq M-1$, in denen alle Werte von 0 bis $M-1$ mit gleicher Wahrscheinlichkeit auftreten. Das heißt, die Zahlenfolge r_i ist in der Menge der natürlichen Zahlen $\{0, 1, \ldots, M-1\} \subset \mathbb{N}_0$ gleichverteilt. Oft ist es in der Praxis von Vorteil, die Zahlenfolge r_i durch

$$x_i = \frac{r_i}{M}\,. \tag{3.1}$$

auf das Intervall $[0, 1)$ abzubilden. Auf diese Weise lässt sich eine Gleichverteilung im Intervall $[0, 1)$ approximieren. Eine Zufallsvariable, die wirklich in $[0, 1)$ gleichverteilt ist, könnte jeden Wert in $[0, 1)$ annehmen. x_i nimmt aber nur M diskrete Punkte im Intervall $[0, 1)$ an. Da die Fließkommaarithmetik eines Computers aber sowieso nur eine endliche Teilmenge der reellen Zahlen darstellen kann, ergeben sich daraus kaum praktische Probleme. Hier werden nur gleichverteilte Generatoren besprochen, da man aus der Gleichverteilung auch alle anderen praktisch relevanten Verteilungen ableiten kann. Näheres wird in Kapitel 5 besprochen.

Fast alle Zufallszahlengeneratoren sind über eine iterative Beziehung der Form

$$R_i = f(R_{i-1}), \qquad i \in \mathbb{N} \tag{3.2}$$

mit vorgegebenen R_0 definiert, wobei die R_i Elemente einer endlichen Menge mit M Elementen sind. Diese Menge können z. B. die natürlichen Zahlen $0, 1, \ldots, M-1$ sein oder auch die Menge der Zweiervektoren $R_i = (r_1, r_2)$ mit $r_1, r_2 \in \mathbb{N}_0$, $0 \leq r_1, r_2 < n$. Hier hat die Menge $M = n^2$ Elemente. Da die Menge, aus der die Elemente R_i stammen, endlich ist, muss sich die Folge der R_i nach spätestens M Gliedern wiederholen. Die kleinste natürliche Zahl P für die

$$R_i = R_{i+P} \quad \forall i \in \mathbb{N}_0 \tag{3.3}$$

gilt, heißt Periode des Zufallszahlengenerators.

In der Literatur findet man verschiedene Verfahren der Generierung von Zufallszahlenfolgen. Will man diese Verfahren vergleichen und bewerten, so hat dies unter folgenden Gesichtspunkten zu geschehen [James 1990]:

Verteilung Die Zahlen der Folge sollten möglichst „zufällig" verteilt sein. Um diese Zufälligkeit zu bewerten, werden die Generatoren verschiedenen statistischen Tests unterzogen, siehe Kapitel 6. Außerdem sollen die Zahlen in einem gewissen Intervall gleichverteilt sein.

Periodenlänge Die Periodenlänge sollte möglichst groß sein, damit in einer Anwendung nur eine Anzahl von Zufallszahlen genutzt wird, die nur einem Bruchteil der Periodenlänge entspricht. Generatoren mit Periodenlängen von 2^{60} oder größer sollten heute Standard sein. Mit steigender Rechenleistung wird sich die notwendige Periodenlänge wohl noch vergrößern.

Reproduzierbarkeit Um verschiedene Algorithmen, die Zufallszahlen verarbeiten, vergleichen zu können, müssen die Zufallszahlenfolgen reproduzierbar erzeugt werden können. Diesem Kriterium werden alle hier besprochenen Verfahren gerecht. Manchmal möchte man nur einen Teil einer aufwendigen Rechnung wiederholen, dann sollte man die Zahlenfolge auch erst ab einem bestimmten Glied reproduzieren können, ohne alle Vorgängerglieder bestimmen zu müssen.

Lange disjunkte Teilfolgen Um umfangreiche Berechnungen parallelisieren zu können, muss der Generator es gestatten, aus der ursprünglichen Folge verschiedene lange disjunkte Subfolgen zu erzeugen. Näheres wird in Kapitel 4 besprochen.

Portabilität In der Vergangenheit wurden, meist aus Gründen der Performance, Zufallszahlengeneratoren (in Maschinensprache) implementiert, die auf anderen Architekturen nicht oder nicht effizient umgesetzt werden konnten. Ziel sollte es sein, einen Generator zu finden, der sich portabel in einer Hochsprache auf einer breiten Hardwarebasis implementieren lässt. Leider ist es in der Praxis gar nicht einfach, diesem Anspruch gerecht zu werden. Meist müssen wenigstens über die Wortbreite bzw. die Rechengenauigkeit des Computers gewisse Annahmen getroffen werden.

Effizienz Der Rechenaufwand für die Zufallszahlenfolge sollte natürlich möglichst klein sein. Der Autor ist aber der Meinung, dass angesichts der heute verfügbaren leistungsfähigen Hardware die vorherigen Kriterien auf keinen Fall dem Drängen nach Effizienz zum Opfer fallen dürfen. Der schnellste Zufallszahlengenerator ist nutztlos, wenn die erzeugte Zahlenfolge nicht „zufällig“ genug ist.

3.1 Einfache lineare Kongruenzen

Auf LEHMER (siehe auch [James 1990], [Knuth 1981a] oder auch [Fishman 1997]) geht die Methode der linearen Kongruenzen zurück. Meist findet man sie in der Form

$$r_i \equiv a r_{i-1} + b \mod m, \tag{3.4}$$

mit $m, i \in \mathbb{N}$, $a \in \{2, 3, \ldots, m-1\} \subset \mathbb{N}$, $r_0, r_i, b \in \{0, 1, \ldots, m-1\} \subset \mathbb{N}_0$. Die Parameter a, b und m und ein Startwert r_1 müssen vorgegeben werden. LEHMER schlug ursprünglich $a = 23$, $b = 0$ und $m = 10^8 + 1$ vor [James 1990]. Insbesondere wird oft $m = 2^n$ oder $m = 10^n$ ($n \in \mathbb{N}$) gewählt. Wenn n der Wortbreite des verwendeten Computers entspricht, so lässt sich ein solcher Generator besonders effektiv implementieren. Wie unten gezeigt wird, ist es jedoch günstiger, $m \in \mathbb{N}_P$ ($\mathbb{N}_P$ bezeichnet die Menge der Primzahlen) zu wählen, siehe unten. Die Wahl $b = 0$ ist auch weit verbreitet, der Generator ist dann rein multiplikativ.

$$r_i \equiv a r_{i-1} \mod m \tag{3.5}$$

Die Qualität des Generators hängt wesentlich von der Wahl der Parameter ab.

Verlangt man, dass sich der Generator (3.4) möglichst einfach implementieren lässt, so liegt es nahe, dass man m gleich 2^e wählt, wenn e die Wortbreite

des verwendeten Computers ist. Meist ist die Integerrecheneinheit so konstruiert, dass automatisch alle Operationen mod 2^e durchgeführt werden und die sonst recht aufwendige Modulooperation entfällt. Diesen Vorteil erkauft man sich aber mit einer signifikant schlechteren Qualität des Generators. Sei m' ein Teiler von m und r'_i die Folge

$$r'_i = r_i \mod m' . \tag{3.6}$$

Dann gilt

$$\begin{aligned} r'_{i+1} &= (ar_i + c \mod m) \mod m' \qquad \text{mit Satz 2.5 folgt} \\ &= (ar_i + c) \mod m' \\ &= (a(r_i \mod m') + c) \mod m' \\ &= (ar'_i + c) \mod m' . \end{aligned} \tag{3.7}$$

Das heißt, die Folge r'_i hat nur eine Periode von maximal m'. Für den Fall $m = 2^e$ folgt daraus, dass die k niederwertigsten Bits von r_i eine Periode von 2^k aufweisen, also das niederwertigste Bit abwechselnd 1 oder 0 ist, was nun nicht gerade einer Zufallszahlenfolge entspricht. Spaltet man die Zahlenfolge in 2^k Zahlenfolgen mittels leapfrog-Methode auf (siehe auch Kapitel 4), so sind die untersten k Bits in jeder Folge sogar konstant. L'ECUYER [L'Ecuyer 1990] warnt daher auch:

> If $m = 2^e$ where e is the number of bits on the computer word, and if one can use unsigned integers without overflow checking, the products modulo m are easy to compute: just discard the overflow. This quick and simple. For that reason, MLCG with moduli of this form are used abundantly in practice, despite their serious drawbacks. Some nuclear physicists, for instance, perform simulations that use billions of random numbers on supercomputers and are quite reluctant to give up using them[...]. Usually, they also generate many substreams in parallel. In view of the above remarks, all this appears dangerous. Perhaps some people like playing with fire.

Es ist also ratsam m so zu wählen, dass m möglichst große Primfaktoren hat bzw. selbst prim ist.

Da $0 \leq r_i \leq m - 1$ gilt, muss sich die Folge (3.4) nach spätestens m Schritten wiederholen. Wann wird diese maximale Periode aber auch erreicht? Dazu lassen sich folgende Sätze formulieren:

Satz 3.1 Der Generator (3.4) erreicht die maximale Periode m genau dann, wenn

1. b relativ prim zu m ist,
2. alle Primfaktoren von m auch Primfaktoren von $a-1$ sind und
3. $a-1$ ein Vielfaches von 4 ist, wenn auch m ein Vielfaches von 4 ist.

Satz 3.2 Im Falle $m = 2^e$, $e \geq 3$ und $b = 0$ beträgt die maximale Periode des Generators (3.4) nur $P = 2^{e-2}$. Sie wird genau dann erreicht, wenn r_0 ungerade und $a \equiv \pm 3 \mod 8$.

Die Beweise dieser beiden Sätze findet man z. B. in [Knuth 1981a].

Wenn $b = 0$ kann die maximale Periode nur noch $m-1$ betragen, da $r_i = 0$ immer auf sich selbst abgebildet wird und so eine konstante Folge bildet. Wählt man $m \in \mathbb{N}_P$, gilt folgender Satz:

Satz 3.3 Der Generator $r_i \equiv a r_{i-1} \mod m$ mit $m \in \mathbb{N}_P$ hat eine volle Periode von $m-1$ genau dann, wenn a ein erzeugendes Element der Gruppe $\mathbb{Z}/m\mathbb{Z}$ ist.

Beweis: Die iterative Beziehung $r_i \equiv a r_{i-1} \mod m$ lautet explizit $r_i \equiv a^i r_0 \mod m$. Wenn a Erzeugendes der multiplikativen Gruppe $\mathbb{Z}/m\mathbb{Z}$ ist, so durchläuft a^i in $m-1$ Schritten die Zahlen 1 bis $m-1$. Diese werden dann bijektiv durch die Multiplikation mit r_0 auf die Zahlen 1 bis $m-1$ abgebildet. Ein erzeugendes Element a kann nur existieren, wenn die Multiplikation modulo m eine bijektive Abbildung ist. Dies ist genau, dann der Fall wenn $m \in \mathbb{N}_P$, denn dann ist nach Satz 2.6 die Multiplikation eindeutig umkehrbar. *q. e. d.*

Eine maximale Periode ist aber nur ein notwendiges Kriterium für einen guten Zufallszahlengenerator. Man betrachte eine Zufallszahlenfolge r_i eines Generators vom Typ (3.4) und die k-dimensionalen Vektoren

$$R_i = (r_i, r_{i+1}, \ldots, r_{i+k-1}) \,. \tag{3.8}$$

Bei einem guten Zufallszahlengenerator sollten die Vektoren R_i gleichmäßig im k-dimensionalen Hyperwürfel verteilt sein. Bei Generatoren der Form (3.4) liegen die Punkte aber typischerweise auf wenigen Ebenen im k-dimensionalen Hyperraum, siehe [Marsaglia 1968]. Abbildung 3.1 illustriert dies für $k = 2$ (Werte auf 1 normiert). Beim Multiplikator $a = 48$ gibt es parallele Ebenen, die sehr dicht liegen. Der Multiplikator $a = 58$ erzeugt Ebenen, deren minimaler Abstand größer ist und die Punkte sind gleichmäßiger auf dem Einheitsquadrat verteilt. Der Spektraltest (siehe Abschnitt 8.1) berechnet den Abstand zwischen diesen (Hyper-)Ebenen. Ein guter Multiplikator zeichnet sich dadurch

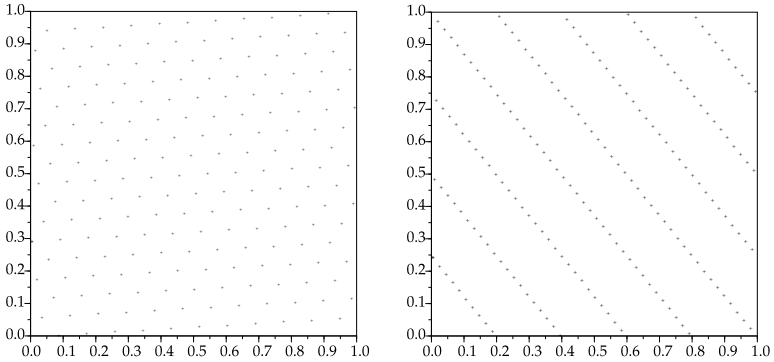

Abbildung 3.1: Gitterstrukturen zweier Zufallszahlengeneratoren; Bei ungünstig gewähltem Multiplikator liegen die Punkte (3.8) auf wenigen Ebenen. Auf der Abszisse sind die auf 1 normierten r_i und auf der Ordinate die auf 1 normierten r_{i+1} aufgetragen.

aus, dass der minimale Abstand zwischen zwei Ebenen in vielen Dimensionen möglichst groß ist.

Verschiedene Klassen neuer Generatoren erhält man, wenn man den Ansatz (3.4) zunächst verallgemeinert und dann wieder durch spezielle Wahl der Parameter einschränkt. Dazu wird (3.4) als vektorwertige Gleichung betrachtet.

$$R_i \equiv AR_{i-1} + B \mod m\,, \qquad i, m, n \in \mathbb{N}\,, A \in \mathbb{N}_0^{n\times n}\,, B \in \mathbb{N}_0^n\,. \tag{3.9}$$

mod m ist hier auf die Elemente des Vektors einzeln anzuwenden. Falls B der Nullvektor ist und $m \in \mathbb{N}_P$, dann wird die maximale Periode $m^n - 1$ des Generators genau dann erreicht, wenn das charakteristische Polynom

$$f(x) = |Ix - A| \mod m \tag{3.10}$$

ein primitives Polynom ist, siehe [L'Ecuyer 1990] oder [Knuth 1981a]. I sei hier die Einheitsmatrix.

3.2 Vektorwertige lineare Kongruenzen

Die Matrix A und der Vektor B sollen hier die Form

$$A = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_n & a_{n-1} & \cdots & a_1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{3.11}$$

haben. Gleichung (3.9) geht in

$$\begin{aligned} R_i = \begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} &\equiv \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_n & a_{n-1} & \cdots & a_1 \end{pmatrix} \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \\ &\equiv \begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ a_n r_{i-n} + a_{n-1} r_{i-(n-1)} + \cdots + a_1 r_{i-1} \end{pmatrix} \mod m \end{aligned} \tag{3.12}$$

über. Die letzten Elemente in den Vektoren R_i bilden dann die Glieder der Zufallszahlenfolge. $n = 1$ würde wieder der Gleichung (3.5) entsprechen. Es ist bekannt ([James 1990]), dass beim Übergang von $n = 1$ zu $n = 2$ die maximale Anzahl der MARSAGLIA-Hyperebenen sich auf die Anzahl der Hyperebenen im Fall $n = 1$ in Räumen der halben Dimensionalität erhöht.

Das charakteristische Polynom lautet hier

$$f(x) = x^n - \sum_{i=1}^{n} a_i x^{n-i} \mod m\,. \tag{3.13}$$

Der Zusammenhang zwischen dem charakteristischen Polynom und der iterativen Beziehung 3.12 soll an einem Beispiel erläutert werden. Sei

$$R_{i+1} \equiv \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 3 & 1 & 1 \end{pmatrix} R_i \mod 5 \qquad R_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{3.14}$$

Das Polynom $x^3 - x^2 - x - 3 \mod 5$ ist primitiv, das heißt, dass Potenzen von x alle Elemente der multiplikativen Gruppe $\mathbb{Z}/5\mathbb{Z}[x]/(x^3 - x^2 - x - 3)$ erzeugen. Berechnet man nun sukzessiv die R_i und die Elemente der multiplikativen Gruppe $\mathbb{Z}/5\mathbb{Z}[x]/(x^3 - x^2 - x - 3)$, so stellt man bemerkenswerterweise fest, dass der Koeffizient vor x^0 und die erste Zahl in R_i übereinstimmen, siehe Tabelle 3.1.

Tabelle 3.1: Über den Zusammenhang zwischen primitiven Polynom und Iterationsvorschrift

i	R_i^T nach (3.14)	$x^i \mod (x^3 - x^2 - x - 3)$
1	$x + 0$	$(0, 0, 3)$
2	$x^2 + 0$	$(0, 3, 3)$
3	$x^2 + x + 3$	$(3, 3, 1)$
4	$4x + 2x^2 + 3$	$(3, 1, 3)$
5	$x^2 + 1$	$(1, 3, 3)$
6	$x^2 + 2x + 3$	$(3, 3, 4)$
7	$3x^2 + 4x + 3$	$(3, 4, 1)$
8	$2x^2 + x + 4$	$(4, 1, 4)$
9	$3x^2 + x + 1$	$(1, 4, 2)$
10	$4x^2 + 4x + 4$	$(4, 2, 4)$
11	$3x^2 + 3x + 2$	$(2, 4, 3)$
12	$x^2 + 4$	$(4, 3, 3)$

3.3 Schieberegistergeneratoren

Bei den Schieberegistergeneratoren wählt man in Gleichung (3.9) $m = 2$, $A = (I + R^\lambda)(I + L^\mu)$ und B als Nullvektor [Marsaglia 1992]. Die $n \times n$-Matrizen R und L sind dabei wie folgt definiert:

$$L = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{3.15}$$

Der Generator arbeitet also nach der Vorschrift

$$R_i \equiv (I + R^\lambda)(I + L^\mu)R_{i-1} \mod 2\,. \tag{3.16}$$

Wendet man L bzw. R einzeln auf einen Vektor R_i an, so werden die Elemente des Vektors um eine Position nach links bzw. rechts geschoben. I ist wieder die $n \times n$-Einheitsmatrix. Den binären Vektor R_i kann man als Binärdarstellung einer ganzen Zahl und somit als Glied einer Folge von Zufallszahlen auffassen. R_0 darf nicht der Nullvektor sein.

Um eine maximale Periode zu erreichen, müssen λ und μ so gewählt werden, dass

$$f(x) = |Ix - (I + R^\lambda)(I + L^\mu)| \mod m \tag{3.17}$$

ein primitives Polynom ist, siehe Gleichung (3.10).

Marsaglia ([Marsaglia 1992]) weißt aber darauf hin, dass dieser Typ von Generatoren keine guten Eigenschaften hat. Nur für sehr große n ($n = 607$, $n = 1279$ oder gar $n = 9689$) seien diese Generatoren empfehlenswert. In der Vergangenheit wurden diese Generatoren vorallem verwendet, da sie sich sehr leicht maschinennah implementieren lassen.

3.4 Verallgemeinerte FIBONACCI-Generatoren

Die verallgemeinerten FIBONACCI-Generatoren kann man als Sonderfall der vektorwertigen linearen Kongruenzen betrachten. Hier werden $a_n = 1$ und genau ein weiteres $a_k = 1$ gesetzt, alle anderen $a_i = 0$. Die Iterationsvorschrift lässt sich dann kurz als

$$r_i = r_{i-k} + r_{i-n} \mod 2^e, \qquad 1 \leq k < n, \qquad e > 1 \tag{3.18}$$

schreiben. Neben der Addition werden hier auch andere binäre Verknüpfungen verwendet. Solche könnten z. B. Subtraktion, Multiplikation oder Exklusivoderverknüpfung sein. Die maximal mögliche Periode P hängt von der gewählten Verknüpfung ab ([Brent 1994], [Marsaglia 1992]).

- Addition: $P = 2^{e-1}(2^n - 1)$
- Subtraktion: $P = 2^{e-1}(2^n - 1)$
- Multiplikation: $P = 2^{e-3}(2^n - 1)$
- Exklusivoderverknüpfung: $P = 2^n - 1$

Im Fall der Exklusivoderverknüpfung kann man den Generator $r_i = r_{i-k} \oplus r_{i-n} \mod 2^e$ auch als e Generatoren vom Typ (3.12) mit $m = 2$ auffassen. Die maximale Periode wird erreicht, wenn das Trinom $x^n + x^k + 1 \mod 2$ primitiv ist. Eine Liste primitiver Trinome findet man u. a. in [Menezes u. a. 1996].

3.5 Add-with-carry und subtract-with-borrow

Diese beiden Generatoren sind dem verallgemeinerten FIBONACCI-Generatoren auf den ersten Blick recht ähnlich. Für add-with-carry lautet die Iterationsvorschrift

$$r_i \equiv r_{i-t} + r_{i-s} + c \mod m, \qquad t > s, \tag{3.19}$$

und für subtract-with-borrow

$$r_i \equiv r_{i-t} - r_{i-s} - c \mod m, \qquad t > s, \quad \text{bzw.} \tag{3.20}$$

$$r_i \equiv r_{i-s} - r_{i-t} - c \mod m, \qquad t > s. \tag{3.21}$$

Von besonderer Bedeutung ist hier die Größe c. Bei der Initialisierung des Generators wird diese willkürlich 0 oder 1 gewählt. Nach jeder Berechnung eines neuen Gliedes der Folge r_i wird c 1 gesetzt, wenn bei der Berechnung von r_i ein Über- bzw. Unterlauf stattfand, sonst 0. Die maximale Periode für (3.19) beträgt $m^t + m^s - 2$ und für (3.20) $m^t - m^s - 2$ bzw. $m^t - m^s$ für (3.21). Etwas ausführlicher werden diese Generatoren in [Marsaglia 1992] besprochen. Leider ist für diese Generatoren keine Methode bekannt, mit der man die Folgen effektiv spalten könnte, weshalb diese Generatoren für Parallelrechner wenig attraktiv sind, siehe auch Kapitel 4.

3.6 Inverse lineare Kongruenzen Generatoren

In [Eichenauer und Lehn. 1986] findet man eine ganz neue Klasse von Pseudozufallszahlengeneratoren.

$$r_i \equiv a\overline{r_{i-1}} + b \mod m \tag{3.22}$$

$\overline{r_{i-1}}$ steht hier für das Inverse von r_{i-1}. Das heißt, $\overline{r_{i-1}}$ erfüllt die Gleichung $1 \equiv \overline{r_{i-1}} \cdot r_{i-1} \mod m$. In diesem Abschnitt gelte $\overline{0} = 0 \mod m$.

Die Parameter sollen den Bedingungen $m \in \mathbb{N}_P$, $a, b \in \mathbb{N}_0$, $a, b < m$ und $a > 1, b > 0$ genügen. Die maximale Periode $P = m$ wird erreicht, genau dann wenn $x^2 - bx - a$ ein primitives Polynom mod m ist.

Für Anwendungen auf Parallelrechnern ist aber folgender Generator viel interessanter ([Eichenauer-Hermann 1993]).

$$r_i = \overline{ai + i_0} \mod m \tag{3.23}$$

Mit den Parametern $m \in \mathbb{N}_P$, $a, i_0 \in \mathbb{N}_0$, $a < m$, $a \geq 1, i_0 \geq 0$ erreicht dieser Generator immer die volle Periode. Seine Qualität hängt nicht signifikant von der Wahl der Parameter ab und man findet auch keine langreichweitigen Korrelationen. Leider gibt es kaum empirische Erfahrungen mit diesem Generator, was auch daran liegen könnte, dass die Berechnung des Inversen mod m sehr rechenaufwendig ist. Im Gegensatz zu Generatoren der Form (3.4) findet man bei den Generatoren (3.22) und (3.23) keinerlei Gitterstruktur, [Hellekalek].

3.7 Kombinierte Generatoren

Um Generatoren mit einer größeren Periode zu erhalten, kann man mehrere Generatoren kombinieren. Basis dafür sind die beiden folgenden Sätze. [L'Ecuyer 1988]

Satz 3.4 Seien $W_1, W_2, \ldots, W_l$ unabhängige diskrete Zufallsvariablen. Die Zufallsvariable W_1 sei zwischen 0 und $d-1$ ($d \in \mathbb{N}$) gleichverteilt, so dass für alle $n \in \mathbb{N}_0$ mit $0 \leq n \leq d-1$ die Wahrscheinlichkeit, dass W_1 den Wert n annimmt gleich

$$\Pr(W_1 = n) = \frac{1}{d} \tag{3.24}$$

ist. Dann ist die diskrete Zufallsvariable

$$V_l = \sum_{i=1}^{l} W_i \mod d \tag{3.25}$$

in den ganzen Zahlen 0 bis $d-1$ gleichverteilt. Die Zufallsvariablen $W_2, \ldots, W_l$ müssen nicht notwendigerweise gleichverteilt sein.

Beweis: Zunächst wird $l = 2$ betrachtet. W_2 nehme Werte zwischen a und b an. Für $0 \leq n \leq d-1$ gilt dann ($k \in \mathbb{N}_0$)

$$\begin{aligned} \Pr(V_2 = n) &= \sum_{k=0}^{\infty} \Pr(W_1 + W_2 = n + kd) \\ &= \sum_{i=a}^{b} \Pr(W_2 = i) \cdot \Pr(W_1 = (n-i) \mod d) \\ &= \frac{1}{d} \sum_{i=a}^{b} \Pr(W_2 = i) = \frac{1}{d}\,. \end{aligned} \tag{3.26}$$

Das heißt, $V_2 = W_1 + W_2 \mod d$ ist zwischen 0 und $d-1$ gleichverteilt. Per Induktion kann man nun zeigen, dass $V_l = W_1 + W_2 + \ldots + W_l = V_{l-1} + W_l \mod d$ auch gleichmäßig zwischen 0 und $d-1$ verteilt ist. *q. e. d.*

Satz 3.5 Die aus den l periodischen Zahlenfolgen $r_{j,i}$ mit den Perioden P_j ($j = 1, 2 \ldots l$) gebildeten geordneten l-Tupel $r_i = (r_{1,i}, r_{2,i}, \ldots r_{l,i})$ bilden eine periodische Folge der Periode P. P ist das kleinste gemeinsame Vielfache der P_j.

Beweis: Der j-te Generator hat die Periode P_j, also muss P ein Vielfaches von P_j sein, was für alle j gilt. Für jede ganze Zahl n, die ein Vielfaches aller P_j ist, gilt $r_{i+n} = r_i$, das heißt $p \leq n$. *q. e. d.*

Es liegt nun nahe, l Generatoren der Form

$$r_{j,i} \equiv a_j r_{j,i-1} \mod m_j \qquad j, i \in \mathbb{N} \qquad 1 \leq j \leq l \tag{3.27}$$

mit der Periode $m_j - 1$ zu betrachten. Aus ihnen lässt sich der Generator

$$\begin{aligned} r_i &\equiv (r_{1,i} - 1) + (r_{2,i} + 1) + \sum_{j=3}^{l} r_{j,i} \mod (m_1 - 1) \\ &\equiv \sum_{j=1}^{l} r_{j,i} \mod (m_1 - 1) \end{aligned} \tag{3.28}$$

ableiten. Der Term $r_{1,i} - 1$ entspricht hierbei dem W_1 im Satz 3.4. Wenn die Module m_j prim sind, lässt sich die Periode P dieses Generators nach oben abschätzen. Da $m_j - 1$ gerade ist, folgt

$$P \leq \frac{\prod_{j=1}^{l}(m_j - 1)}{2^{j-1}} . \tag{3.29}$$

Die m_j sollten natürlich möglichst so gewählt werden, dass das Gleichheitszeichen gilt.

Neben der großen Periode hat der Generator (3.28) noch den großen Vorteil, dass er keinerlei Gitterstruktur aufweist, siehe Abbildung 3.3.

Etwas ausführlicher wird dieser Typ von Generator in [L'Ecuyer 1988] und [James 1990] besprochen. In [L'Ecuyer 1999] wird der Ansatz (3.28) verallgemeinert. Bei [Marsaglia und Zaman 1994] findet man einige Implementationen von kombinierten Generatoren, die sich aus Lineare-Kongruenzen-Generatoren, verallgemeinerten FIBONACCI-Generatoren, Add-with-carry- und subtract-with-borrow-Generatoren zusammensetzen.

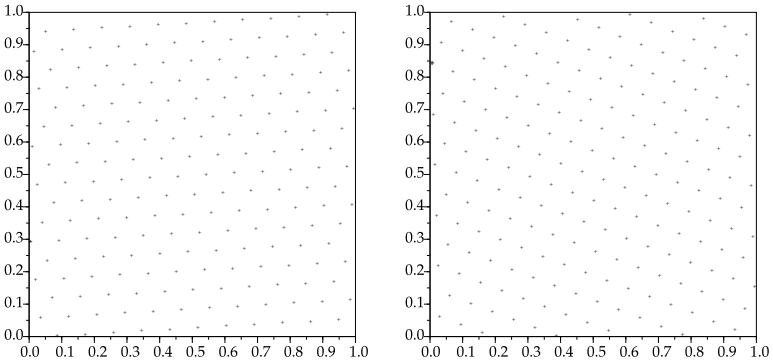

Abbildung 3.2: Gitterstruktur zweier Zufallszahlengeneratoren mit reinmultiplikativen Kongruenzen; Auf der Abszisse sind die auf 1 normierten r_i und auf der Ordinate die auf 1 normierten r_{i+1} aufgetragen.

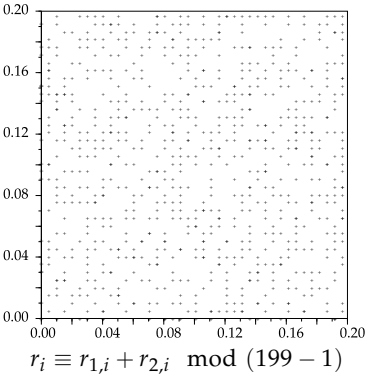

Abbildung 3.3: Die mit Gleichung (3.28) kombinierten Generatoren aus Abbildung 3.2 zeigen keinerlei Gitterstruktur; Auf der Abszisse sind die auf 1 normierten r_i und auf der Ordinate die auf 1 normierten r_{i+1} aufgetragen.

Techniken zur Parallelisierung der Erzeugung von (Pseudo-) Zufallszahlenfolgen

Für sehr rechenintensive Aufgaben soll nun die Erzeugung der Zufallszahlen parallelisiert werden. Die Informatik unterscheidet verschiedene Arten der Parallelität. Die hier gemachten Aussagen beziehen sich in erster Linie auf SPMD- und MIMD[1]-Architekturen, bei anderen Architekturen, wie z. B. Vektorrechnern sind gegebenenfalls noch Besonderheiten zu beachten, siehe auch [Anderson 1990] oder [Brent 1992].

Im wesentlichen gibt es zwei Methoden aus einem sequentiellen Zufallszahlengenerator, der die Folge r_i liefert, weitere Folgen abzuleiten, die dann parallel auf verschiedenen CPUs berechnet werden können. Erstens kann man die Folge r_i in j Folgen sich nicht überlappender Teilfolgen $s_{j,i}$ spalten.

$$\begin{aligned} s_{0,i} &= r_i \\ s_{1,i} &= r_{i+k} \\ &\dots \\ s_{j-1,i} &= r_{i+(j-1)k} \end{aligned} \qquad i = 0 \dots k-1 \tag{4.1}$$

Damit dieses Verfahren anwendbar ist, muss man aus r_i ohne großen Aufwand r_{i+k} berechnen können, ohne die dazwischen liegenden Gliedern berechnen zu müssen. Der zweite Ansatz erzeugt k Folgen $t_{j,i}$ der Art:

$$\begin{aligned} t_{0,i} &= r_{ki} \\ t_{1,i} &= r_{ki+1} \\ &\dots \\ t_{k-1,i} &= r_{ki+(k-1)} \end{aligned} \qquad i = 0 \dots k-1 \tag{4.2}$$

[1] Single Program Multiple Data bzw. Multiple Instruction/Multiple Data, Beowulf-Cluster oder cluster of workstations gehören z. B. in diese Kategorie.

Jeweils k Glieder der Folge r_i bilden je ein Glied der k Folgen $t_{j,i}$, deshalb ist in den $t_{j,i}$ nur jedes k-te Glied von r_i enthalten. In der englischsprachigen Literatur wird diese Methode leapfrog method genannt. Leapfrog heißt soviel wie Bocksprung. In Abbildung 4.1 und Abbildung 4.2 soll der Unterschied zwischen den Methoden nochmal graphisch verdeutlicht werden.

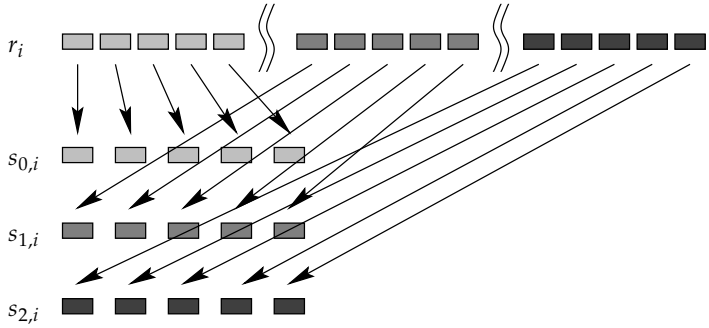

Abbildung 4.1: Parallelisierung eines Zufallszahlengenerators durch Sequenzspaltung

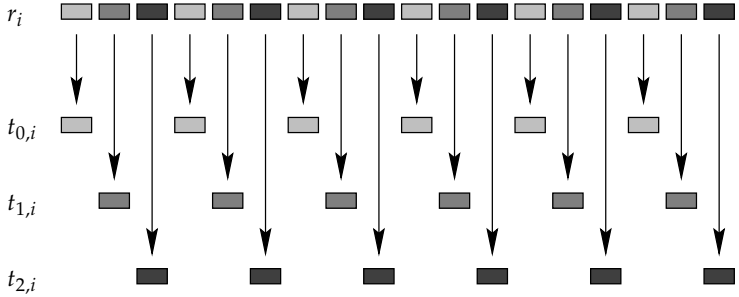

Abbildung 4.2: Parallelisierung eines Zufallszahlengenerators mit Leapfrog-Methode

In [Coddington 1997] wird noch eine dritte Methode vorgeschlagen. Wählt man den Anfangszustand eines Generators mit sehr großer Periode zufällig, so ist die Wahrscheinlichkeit, dass gleiche Generatoren mit verschiedenen Anfangszuständen sich überlappende Folgen liefern, sehr gering. Wählt man die Anfangszustände aber wahrlos, so sind die Wechselwirkungen zwischen den Generatoren nur schwer zu kontrollieren. Daher soll diese Möglichkeit hier nicht weiter betrachtet werden.

Wie setzt man diese Konzepte nun für die in Abschnitt 3 vorgestellten Generatoren konkret um? Bei den einfachen linearen Kongruenzen kann man das iterative Bildungsgesetz in ein explizites umwandeln.

$$\begin{aligned} r_i &\equiv a r_{i-1} + b \equiv a^i r_0 + \left(1 + a^1 + a^2 + \cdots + a^{i-1}\right) b \\ &\equiv a^i r_0 + \frac{a^i - 1}{a - 1} b \mod m \end{aligned} \tag{4.3}$$

Die Formel (4.3) lässt sich z. B. durch vollständige Induktion beweisen. Sie gestattet es, Bildungsgesetze für die Folgen $s_{j,i}$ und $t_{j,i}$ anzugeben.

$$s_{j,i} \equiv a s_{j,i-1} + b \mod m\,, \qquad s_{j,0} \equiv a^{jk} r_0 + \frac{a^{jk} - 1}{a - 1} b \mod m \tag{4.4}$$

$$t_{j,i} \equiv a^k t_{j,i-1} + \frac{a^k - 1}{a - 1} b \mod m\,, \qquad t_{j,0} \equiv r_j \tag{4.5}$$

Man erhält also wieder einfache Generatoren vom Typ (3.4), jedoch mit anderen Parametern. Für $b = 0$ vereinfachen sich die Gleichungen (4.4) und (4.5) zu

$$s_{j,i} \equiv a s_{j,i-1} \mod m\,, \qquad s_{j,0} \equiv a^{jk} r_0 \mod m \tag{4.6}$$

und

$$t_{j,i} \equiv a^k t_{j,i-1} \mod m\,, \qquad t_{j,0} \equiv r_j\,. \tag{4.7}$$

Auf die vektorwertigen Generatoren der Form

$$R_i \equiv A R_{i-1} \mod m\,, \qquad i, m, n \in \mathbb{N}\,, A \in \mathbb{N}^{n \times n}\,, \tag{4.8}$$

die in den Abschnitten 3.2, 3.3 und 3.4 besprochen wurden, kann man nun obige Überlegungen übertragen. An die Stelle der Potenzen des Skalars a treten die der Matrix A.

$$S_{j,i} \equiv A S_{j,i-1} \mod m\,, \qquad S_{j,0} \equiv A^{jk} R_0 \mod m \tag{4.9}$$

$$T_{j,i} \equiv A^k T_{j,i-1} \mod m\,, \qquad T_{j,0} \equiv R_j \tag{4.10}$$

$S_{j,i}$ und $T_{j,i}$ sind natürlich auch Vektoren. Für Generatoren der Form (3.12) beschreibt [L'Ecuyer 1990] diese Vorgehensweise genauer. Diese hat aber einen signifikanten Nachteil. Die Matrix A ist schwach besetzt, was die Matrixmultiplikation $A^k T_{j,i-1} \mod m$ enorm erleichtert. Hat A die Größe $n \times n$, so müssen nur n Multiplikationen berechnet werden. Die Potenzen von A sind aber im Allgemeinen nicht mehr schwach besetzt, es müssen n^2 Multiplikationen vorgenommen werden. Dies kann den Rechenaufwand für die Folge $T_{j,i}$ enorm erhöhen.

Um diesem Problem zu entgehen, schauen wir uns die Folge $T_{0,i} \equiv A^k T_{0,i-1} \mod m$, $T_{0,0} \equiv R_0$ etwas genauer an.

$$\begin{pmatrix} r_{i-n+k} \\ \vdots \\ r_{i-2+k} \\ r_{i-1+k} \end{pmatrix} \equiv \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_n & a_{n-1} & \cdots & a_1 \end{pmatrix}^k \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \mod m \tag{4.11}$$

Für die Zufallszahlenfolge sind nur die jeweils letzten Glieder des Vektors R_i relevant. Wirklich interessant ist also nur jedes k. Glied der Folge r_i. Was geschieht, wenn sich die Vektoren R_i nur aus solchen Gliedern zusammensetzten? Die Generator transformiert sich dann auf die Form

$$\begin{pmatrix} r_{i+k} \\ \vdots \\ r_{i+(n-1)k} \\ r_{i+nk} \end{pmatrix} \equiv \underbrace{\begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ b_n & b_{n-1} & \cdots & b_1 \end{pmatrix}}_{B} \begin{pmatrix} r_i \\ \vdots \\ r_{i+(n-2)k} \\ r_{i+(n-1)k} \end{pmatrix} \mod m \tag{4.12}$$

Dass die Matrix B genau diese Form hat, soll nun gezeigt werden.

Beweis: Mit der Matrix OE_j, deren Elemente überall 0 sind, außer an der Position (j, j), wo der Matrixeintrag 1 ist, gilt folgende Beziehung

$$OE_j \cdot \underbrace{\begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_n & a_{n-1} & \cdots & a_1 \end{pmatrix}}_{A}^{1+q+(n-j)} \underbrace{\begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix}}_{R} = \underbrace{\begin{pmatrix} 0 \\ \vdots \\ r_{i+q} \\ \vdots \\ 0 \end{pmatrix}}_{r_{i+q} \text{ in Zeile } j} \tag{4.13}$$

Die Gleichung (4.12) kann nun mit noch zunächst unbekannter Matrix B auf folgende Weise umgeschrieben werden:

$$B \sum_{j=1}^{n} OE_j \cdot A^{1+(j-1)k+(n-j)} \cdot R \equiv \sum_{j=1}^{n} OE_j \cdot A^{1+jk+(n-j)} \cdot R \mod m\,. \tag{4.14}$$

Die Gleichung muss natürlich von R unabhängig sein.

$$B \sum_{j=0}^{n-1} OE_{j+1} \cdot A^{jk+n-j} \equiv \sum_{j=1}^{n} OE_j \cdot A^{1+jk+n-j} \mod m \tag{4.15}$$

$$B\Big(OE_1 \cdot A^n + \overbrace{\sum_{j=1}^{n-1} OE_{j+1} \cdot A^{jk+n-j}}^{D}\Big) \equiv OE_n \cdot A^{1+kn} + \underbrace{\sum_{j=1}^{n-1} OE_j \cdot A^{1+jk+n-j}}_{D'} \mod m \tag{4.16}$$

Die $n \times n$-Matrix A hat folgende Eigenschaft: Die Zeilen $1, \ldots, n-1$ des Produkts $A \cdot C$ (C sei auch eine $n \times n$-Matrix) stimmen mit den Zeilen $2, \ldots n$ von C überein. Die Multiplikation mit A schiebt also die Zeilen von C um eins nach oben, die letzte Zeile enthält neue Einträge. Dies hat zur Folge, dass die Zeilen $2, \ldots n$ von D mit den Zeilen $1, \ldots, n-1$ von D' übereinstimmen. Wegen der speziellen Form von OE_j, verschwinden alle Matrixeinträge der ersten Zeile von D und alle der letzten Zeile von D'.

$$\left(B - \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{pmatrix} \right) (OE_1 \cdot A^n + D) \equiv \tag{4.17}$$

$$\left(B - \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{pmatrix} \right) (OE_1 \cdot A^n + D) + D' \equiv OE_n \cdot A^{1+kn} + D' \mod m$$

$$\left(B - \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{pmatrix} \right) (OE_1 \cdot A^n + D) \equiv OE_n \cdot A^{1+kn} \mod m \tag{4.18}$$

In der Matrix $OE_n \cdot A^{1+kn}$ sind außer in der letzten Zeile alle Einträge Null. Die Matrix $OE_1 \cdot A^n + D$ wird im allgemeinen überall von Null verschiedene Einträge haben. Daher müssen in

$$\left(B - \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \end{pmatrix} \right)$$

in allen Zeilen außer der letzten, die Einträge verschwinden. *q. e. d.*

Man könnte nun die unbekannten Elemente der Matrix B aus der Gleichung (4.18) berechnen. Genauso gut kann man auch die r_i, r_{i+k} bis r_{i+2nk} berechnen (aus der ursprünglichen Folge) und dann das lineare Gleichungssystem (4.12) lösen.

Ein anderer Ansatz, das Aufspalten der Zahlenfolge in Teilfolgen zu ermöglichen, besteht darin, die iterative Vorschrift (3.12) in eine explizite umzuwandeln. Dies ist prinzipiell auch möglich. Die Vorgehenweise ist der beim Lösen von homogenen linearen gewöhnlichen Differentialgleichnungen ähnlich[Lueker 1980]. Dabei muss man die Nullstellen eines Polynom vom Grade der Anzahl der Summanden in der Iterationsvorschrift lösen. Selbst im einfachsten Fall zweier Summanden ist die explizite Formel recht unhandlich, da im allgemeinen irrationale Faktoren auftreten..

Bei den kombinierten Zufallszahlengeneratoren nach Abschnitt 3.7 werden die Generatoren einzeln wie in (4.4) und (4.5) behandelt.

Ableitung weiterer Verteilungen aus der Normalverteilung

Bisher wurden nur im Intervall $[0, 1)$ gleichverteilte Zufallszahlenfolgen betrachtet. Oftmals werden aber auch andere Verteilungen benötigt, die man aus Gleichverteilung ableiten kann. Dabei kann man sich zweier wichtiger Methoden bedienen ([Knuth 1981a], [Vesely 1993], [Press u. a. 1997]), der Transformationsmethode und der Zurückweisungs- oder Verwerfungsmethode.

5.1 Grundlagen und Definitionen

Definition 5.1 (Verteilungsfunktion) Sei x eine reellwertige Zufallsvariable, dann bezeichnet die Verteilungsfunktion $P(x_0)$ die Wahrscheinlichkeit, dass die Zufallsvariable x einen Wert $x < x_0$ annimmt.

Die bisher betrachten im Intervall $[0, 1)$ gleichverteilten Zufallszahlenfolgen haben die Verteilungsfunktion

$$P(x) = \begin{cases} 0 & \text{falls } x < 0 \\ x & \text{falls } 0 \leq x \leq 1 \\ 1 & \text{falls } 1 < x \end{cases} . \tag{5.1}$$

Definition 5.2 (Wahrscheinlichkeitsdichte) Die Wahrscheinlichkeitsdichte $p(x)$ wird dadurch definiert, dass $p(x_0)\,\mathrm{d}x$ die Wahrscheinlichkeit dafür beschreibt, dass die reellwertige Zufallsvariable x Werte aus dem Intervall $[x_0, x_0 + \mathrm{d}x]$ annimmt.

Aus diesen beiden Definitionen folgt:

$$p(x) = \frac{\mathrm{d}P(x)}{\mathrm{d}x}, \qquad P(x_0) = \int_{-\infty}^{x_0} p(x)\,\mathrm{d}x \tag{5.2}$$

5.2 Transformationsmethode

Wie verhält sich eine reellwertige Zufallsvariable x mit der Verteilungsfunktion $P(x)$ unter der bijektiven Variablentransformation $y = f(x)$? Aus der Erhaltung der Wahrscheinlichkeit folgt unmittelbar

$$|\mathrm{d}P(y)| = |\mathrm{d}P(x)|\,, \tag{5.3}$$

und mit Gleichung (5.2) folgt

$$|p(y)\,\mathrm{d}y| = |p(x)\,\mathrm{d}x| \qquad \text{bzw.} \qquad p(y) = p(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right|\,. \tag{5.4}$$

Die Beziehung $y = f(x)$ ist bijektiv, es existiert also eine inverse Funktion $x = \bar{f}(y)$.

$$p(y) = p(\bar{f}(y))\left|\frac{\mathrm{d}(\bar{f}(y))}{\mathrm{d}y}\right| \tag{5.5}$$

Wenn y eine im Intervall $[0, 1)$ gleichverteilte reellwertige Zufallsvariable ist, so ist $p(y) = 1$ und somit

$$p(x) = \left|\frac{\mathrm{d}y}{\mathrm{d}x}\right| = \left|\frac{\mathrm{d}(f(x))}{\mathrm{d}x}\right|\,. \tag{5.6}$$

Aus Gleichung (5.2) folgt sofort

$$y = f(x) = P(x) \qquad \text{bzw.} \qquad x = \bar{P}(y)\,. \tag{5.7}$$

Man muss also nur die inverse Funktion ($\bar{P}(y)$ der Verteilungsfunktion $P(x)$ finden. Diese Methode lässt sich auch auf reellwertige Zufallsvektoren $\vec{x}$ und $\vec{y}$ verallgemeinern. Dann findet man mit der JACOBIdeterminante $\left|\frac{\partial \vec{x}}{\partial \vec{y}}\right|$ der inversen Transformation $\vec{x} = \bar{f}(\vec{y})$ die Beziehung

$$p(\vec{y}) = p(\vec{x})\left|\frac{\partial \vec{x}}{\partial \vec{y}}\right|\,. \tag{5.8}$$

Ist $\vec{y}$ im n-Dimensionalen Hyperwürfel $[0, 1)^n$ gleichverteilt, so ist $p(\vec{y}) = 1$ und

$$p(\vec{x}) = \left|\frac{\partial \vec{y}}{\partial \vec{x}}\right|\,. \tag{5.9}$$

5.3 Zurückweisungsmethode

Die Transformationsmethode ist nur praktikabel, wenn man einen analytischen Ausdruck für die Umkehrfunktion $x = \bar{P}(y)$ hat. Die Zurückweisungsmethode kommt auch ohne deren Kenntnis aus. Die Wahrscheinlichkeitsdichte der Zufallsvariable x sei nur im Intervall $[a, b]$ verschieden von Null und nehme dort den Maximalwert $p_{\max}$ an. Der Algorithmus für die Zurückweisungsmethode für eine Zufallsvariable mit der Wahrscheinlichkeitsdichte $p(x)$ sieht dann wie folgt aus:

1. Wähle eine gleichverteillte Zufallszahl x aus dem Intervall $[a, b]$ und eine zweite y aus dem Intervall $[0, p_{\max}]$.

2. Wenn $y \leq p(x)$, ist x die nächste Zufallszahl, ansonsten wird wieder bei 1. begonnen.

Abbildung 5.1 macht die Methode noch einmal graphisch deutlich. Dort wo die Wahrscheinlichkeitsdichte $p(x)$ groß ist, ist auch die Wahrscheinlichkeit, dass x als Zufallswert akzeptiert wird, groß.

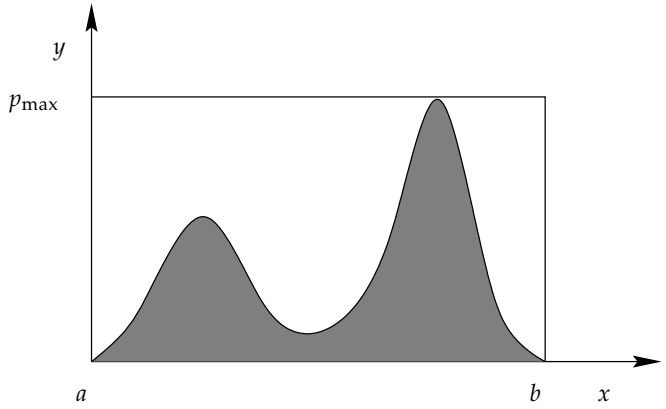

Abbildung 5.1: Bei der Zurückweisungsmethode wählt man einen Punkt aus dem Rechteck $[a, b] \times [0, p_{\max}]$, nur wenn der Punkt unter dem Graphen der Wahrscheinlichkeitsdichte liegt, wird der Wert der x-Koordinate als Zufallswert akzeptiert.

Ist die Fläche des Rechtecks $[a, b] \times [0, p_{\max}]$ viel größer als die Fläche unter dem Graphen der Wahrscheinlichkeitsdichte, so werden Werte sehr oft zurückgewiesen und die Methode wird sehr ineffektiv. Dann kann man aber die Zurückweisungsmethode mit der Transformationsmethode kombinieren. Dazu wählt man eine Vergleichsfunktion $f(x)$ mit $p(x) \leq f(x)$ für alle $x \in$

$[a, b]$. $f(x)$ sei integrierbar und habe die (umkehrbare) Stammfunktion $F(x)$. Der Algorithmus lautet dann:

1. Wähle eine Zufallszahl x aus dem Intervall $[a, b]$ mit der Verteilung

$$\hat{p}(x) = \frac{f(x)}{F(b) - F(a)}$$

   und eine zweite gleichverteilte Zufallszahl y aus dem Intervall $[0, f(x)]$.

2. Wenn $y \leq p(x)$, ist x die nächste Zufallszahl, ansonsten wird wieder bei 1. begonnen.

Methoden zum Testen (Pseudo-) Zufallszahlengeneratoren

Um die Qualität eines Zufallszahlengenerators beurteilen zu können, kann man ihn einer Reihe von empirischen Tests unterziehen. [Knuth 1981a] schlägt zehn statistische Tests vor, die hier kurz beschrieben werden sollen. Die Tests werden auch auf von den Generatoren gebildete Subfolgen angewandt. Außerdem werden auf Random-Walks basierende Tests, eine Monte-Carlo-Simulation des ISING-Modells und eine Monte-Carlo-Integration vorgestellt.

Den meisten Tests liegt eine erwartete diskrete Wahrscheinlichkeitsverteilung p_i zugrunde. Wenn im Experiment k verschiedene Ereignisse auftreten können, läuft der Index i von 1 bis k. Mit der Anzahl der wirklich beobachteten Ereignisse x_i lässt sich der χ^2-Test durchführen. Wenn das Ereignis i mit einer Wahrscheinlichkeit p_i eintritt, sollte man im Mittel np_i Ereignisse der Klasse i beobachten, wenn n Summe der beobachteten Ereignisse ist, $n = \sum_{i=1}^{k} x_i$. Die Testgröße

$$c^2 = \sum_{i=1}^{k} \frac{(x_i - np_i)^2}{np_i} \tag{6.1}$$

ist ein Maß dafür, wie stark die empirische Verteilung von der theoretischen Verteilung abweicht. Dazu wird über die quadratische Abweichung der beobachteten von der erwarteten Anzahl der eingetretenen Ereignisse i summiert. Die Summanden werden mit dem reziproken der erwarteten Anzahl gewichtet, das heißt, unwahrscheinliche Ereignisse werden stärker gewichtet. Beim χ^2-Test wird also die Hypothese getestet, dass die Ereignisse einer vorgegebenen Verteilung gehorchen.

Die χ^2-Wahrscheinlichkeitsfunktion $Q(c^2, \nu)$ gibt Auskunft darüber, ob der Test bestanden wurde. Ist $Q\left(\frac{c^2}{2}, \frac{\nu}{2}\right) < 0{,}01$ oder $Q\left(\frac{c^2}{2}, \frac{\nu}{2}\right) > 0{,}99$, muss der Test als nicht bestanden angesehen werden. ν ist die Anzahl der Freiheitsgrade $\nu = k - 1$. Die χ^2-Wahrscheinlichkeitsfunktion lautet

$$Q(a, x) = \frac{1}{\Gamma(a)} \int_x^\infty \mathrm{e}^{-t} t^{a-1} \,\mathrm{d}t \qquad a > 0\,, \tag{6.2}$$

siehe auch [Press u. a. 1997]. Die Philosophie hinter dieser Vorgehensweise ist die folgende. Die Funktion $Q\left(\frac{c^2}{2}, \frac{\nu}{2}\right)$ gibt die Wahrscheinlichkeit an, dass man, wenn die x_i der durch p_i gegebenen diskreten Verteilungsfunktion genügen, ein c^2 findet, das größer gleich c^2 ist. Wenn also $Q\left(\frac{c^2}{2}, \frac{\nu}{2}\right)$ unterhalb einer vorgegeben Schranke liegt, ist der Test nicht bestanden. Wenn andererseits $Q\left(\frac{c^2}{2}, \frac{\nu}{2}\right)$ nahe 1 liegt, wurde der Test auch nicht bestanden, denn dann weicht die empirische Verteilung nur sehr wenig von der theoretischen Verteilung ab, was darauf hinweist, dass die beobachten Ergebnisse nicht zufällig sind. Meist wird der χ^2-Test mehrfach durchgeführt.

Angenommen man würfle gleichzeitig mit zwei nicht unterscheidbaren Würfeln. Die Summe der Augen der beiden Würfel kann Zahlen von 2 bis 12 annehmen. Diese treten bei fairen Würfeln mit den folgenden Wahrscheinlichkeiten auf.

Augenzahl	2	3	4	5	6	7	8	9	10	11	12
Wahrscheinlichkeit	$\frac{1}{36}$	$\frac{1}{18}$	$\frac{1}{12}$	$\frac{1}{9}$	$\frac{5}{36}$	$\frac{1}{6}$	$\frac{5}{36}$	$\frac{1}{9}$	$\frac{1}{12}$	$\frac{1}{18}$	$\frac{1}{36}$

Nach 360 Würfen wurden in einem Experiment die Augenzahlen 2 bis 12 registriert.

Augenzahl	2	3	4	5	6	7	8	9	10	11	12
erwartet	10	20	30	40	50	60	50	40	30	20	10
beobachtet	13	26	28	36	45	55	49	42	35	19	12

Man berechnet leicht die Größen $c^2 = 5{,}55$ und $Q\left(\frac{5{,}55}{2}, \frac{11}{2}\right) = 0{,}851$. Die Würfel sind offensichtlich fair. Ein zweites Experiment mit anderen Würfeln ergab:

Augenzahl	2	3	4	5	6	7	8	9	10	11	12
erwartet	10	20	30	40	50	60	50	40	30	20	10
beobachtet	4	20	32	50	30	60	53	39	35	17	20

Hier findet man $c^2 = 25{,}72$ und $Q\left(\frac{25{,}72}{2}, \frac{11}{2}\right) = 0{,}004$. Der Test wurde diesesmal nicht bestanden.

Die Programme für die hier vorgestellten Testprozeduren befinden sich auf der beiliegenden CD-ROM.

6.1 Empirische Tests nach KNUTH

6.1.1 Equidistribution test

Im equidistribution test (Gleichverteilungstest) werden mit gleicher Wahrscheinlichkeit Zahlen aus dem Intervall $[0, 1)$ gewählt. Teilt man das Intervall in m gleich große disjunkte Teilintervalle, so fällt die gewählte Zahl in genau eines dieser Teilintervalle. Die Wahrscheinlichkeit, dass ein bestimmtes Teilintervall gewählt wird, beträgt $1/m$.

Das Testprogramm equidisttest arbeitet genau nach diesem Prinzip. Ein Beispiel:

```
user@random:~ > equidisttest 5 128 750 12
```

Hier wird das Intervall $[0, 1)$ in 12 Teilintervalle zerlegt, nach dem 750 Zufallszahlen ausgewählt wurden, wird ein χ^2-Test durchgeführt und der Wert der Wahrscheinlichkeitsfunktion ausgegeben. Der Test wird je 5-mal durchgeführt. Um auch Subfolgen zu testen, wird die Folge nach der leapfrog-Methode schrittweise in 1 bis 128 Subfolgen gespalten, wobei jeweils die erste der Teilfolgen dem Test unterzogen wird. Wenn nicht anders angegeben wird, haben die ersten beiden Parameter bei den folgenden Programmen die gleiche Bedeutung wie hier.

bitequidisttest testet die Gleichverteilung der einzelnen Bits. Dazu wird immer nur ein bestimmtes Bit der vom Zufallszahlengenerator gebildeten 31-Bit-Integerzahl betrachtet. Dazu wieder ein Beispiel:

```
user@random:~ > bitequidisttest 5 128 500
```

Dies entspricht genau der gleichen Testprozedur wie oben nur auf einzelne Bits angewandt. Natürlich kann die Anzahl der Teilintervalle hier nicht frei gewählt werden, sie beträgt immer zwei.

Das Programm bitequidisttest untersucht immer alle implementierten Generatoren. Will man das Testprogramm für weitere Generatoren erweitern, so muss man in der Funktion main nur eine Zeile mit dem Funktionsaufruf für die Testprozedur und eine Zeile für die Deklaration des Pseudozufallszahlengenerators hinzu fügen. Das gilt auch für alle weiteren Testprogramme. Die dazu notwendigen Quelltexte befinden sich auf der der Arbeit beiliegenden CD-ROM.

6.1.2 Serial test

Der serial test untersucht Paare (r_{2j}, r_{2j+1}) aufeinander folgender ganzer Zufallszahlen. Liegen die Zufallszahlen r_i im Bereich $0 \leq r_i < m$, so tritt jedes der m^2 möglichen Paare mit einer Wahrscheinlichkeit $1/m^2$ auf. Das Programm serialtest kann wie folgt aufgerufen werden:

```
user@random:~ > serialtest 5 128 5000 12
```

Die Bedeutung der Parameter ist die gleiche wie bei dem Programm equidisttest, nur dass hier nicht aus 12 Teilintervallen ausgewählt wird sondern aus 12^2 Paaren. Nachdem 5000 Paare gewählt wurden, wird ein χ^2-Test durchgeführt.

6.1.3 Gap test

Bei diesem Test werden Folgen $r_i, r_{i+1}, \ldots r_{i+q}$ gesucht bei denen außer r_{i+q} keines der Glieder im Intervall $[a, b]$ liegt $(0 \leq a, b, r_i < 1)$. Der Test zählt nun wie oft solche Folgen der Länge $q+1$ gefunden werden. Die Wahrscheinlichkeit eine Folge der Länge $q+1$ zu finden beträgt

$$p_q = (b-a)(1-(b-a))^q \,. \tag{6.3}$$

Diese Wahrscheinlichkeitsverteilung ist wieder Ausgangspunkt für einen χ^2-Test. Um z. B. den Gap test mit dem Intervall $[0, 0{,}5]$ durchzuführen, wird das Programm gaptest aufgerufen.

```
user@random:~ > gaptest 5 128 0 0.5 8 5000
```

Hierbei werden für den χ^2-Test alle Folgen der Läge acht oder länger zu einer Klasse zusammengefasst, 5000 Folgen werden untersucht.

6.1.4 Partition test

Wenn man aus der ganzzahlwertigen Folge r_i $(0 \leq r_i < d)$ k-Tupel bildet, so kann man untersuchen mit welcher Wahrscheinlichkeit man in solch einem k-Tupel $k, k-1, \ldots$ verschiedene Werte findet bzw. gar alle gleich groß sind. Die Wahrscheinlichkeit in einem solchen k-Tupel r verschiedene Werte zu finden beträgt

$$p_r = \frac{d(d-1)\ldots(d-r+1)}{d^k} \left\{ {k \atop r} \right\} . \tag{6.4}$$

$\left\{ {k \atop r} \right\}$ steht hier für eine STIRLINGsche Zahl zweiter Art, siehe [Knuth 1981b]. Um den Test mit 5-Tupeln und $d = 52$ durchzuführen, muss man nur

```
user@random:~ > partitiontest 5 128 52 5 5000
```

aufrufen, wobei 5000 5-Tupel untersucht werden. Da für kleine r die Wahrscheinlichkeiten schnell sehr klein werden, werden für den χ^2-Test einige Klassen mit kleinem r zu einer großen zusammengefasst.

6.1.5 Coupon collector's test

Auch hier wird eine ganzzahlwertige Folge r_i ($0 \leq r_i < d$) benutzt. Mit welcher Wahrscheinlichkeit muss man r aufeinanderfolgende Glieder einer solchen Folge betrachten, bis sämtliche Zahlen 0 bis $d-1$ beobachtet wurden? Man findet

$$p_r = \frac{d!}{d^r} \left\{ \begin{matrix} r-1 \\ d-1 \end{matrix} \right\} \qquad d \leq r < t\,. \tag{6.5}$$

Die Wahrscheinlichkeit dafür, dass man t oder mehr als t solcher Glieder betrachten muss, beträgt

$$p_t = 1 - \frac{d!}{d^{t-1}} \left\{ \begin{matrix} t-1 \\ d \end{matrix} \right\}\,. \tag{6.6}$$

Die Formel ist nützlich, wenn man, wegen der kleinen Wahrscheinlichkeiten für große r, alle Folgen mit einem $r \geq t$ für den χ^2-Test zusammenfasst. Wieder ein Beispiel:

```
user@random:~ > couponcollectorstest 5 128 2500 7 35
```

Für einen χ^2-Test werden hier die Längen von 2500 Folgen untersucht, 7 Objekte sind zu sammeln, $t = 35$.

6.1.6 Permutation test

Generiert man eine Folge von n Zufallszahlen r_i, so können diese auf $n!$ Arten angeordnet sein. Für $n = 3$ findet man z. B. folgende Anordnungen:

$$\begin{array}{ll} r_1{<}r_2{<}r_3 & r_1{<}r_3{<}r_2 \\ r_2{<}r_1{<}r_3 & r_2{<}r_3{<}r_1 \\ r_3{<}r_1{<}r_2 & r_3{<}r_2{<}r_1 \end{array} \tag{6.7}$$

Jede dieser Ordnungen sollte man mit der gleichen Wahrscheinlichkeit $\frac{1}{n!}$ finden. Der Fall, dass mehrere r_i gleich groß sind, wird hier nicht weiter berücksichtigt, was kein großer Fehler ist, wenn der Wertebereich aus dem die r_i Werte annehmen können, nur groß genug ist (hier ca. 2^{31}). Das Programm permutationtest untersucht die relative Ordnung solcher Folgen.

```
user@random:~ > permutationtest 5 128 5 3000
```

Dieser Aufruf untersucht 3000 5-Tupel und wendet auf die gefundene Verteilung den χ^2-Test an.

6.1.7 Run test

Der Run test sucht nach Subfolgen die monoton sind. Der Run test hier sucht nach Subfolgen die monoton nach oben laufen. In folgender Auflistung ist der Anfang einer solchen Folge jeweils durch ein | gekennzeichnet.

$$11\,|\,1\;\;13\;\;16\,|\,0\;\;4\;\;10\;\;12\,|\,8\,|\,7\;\;13\;\;15\,|\,7\;\;12\;\;19 \tag{6.8}$$

Da solche Folgen aber nicht statistisch voneinander unabhängig sind, kann man auf die Häufigkeitsverteilung der auftretenden Längen den χ^2-Test nicht anwenden. Statistische Unabhängigkeit kann man aber erhalten, indem man nach einer monotonen Subfolge einen Wert „weg wirft". Man findet dann eine monotone Subfolge der Länge r mit der Wahrscheinlichkeit

$$p_r = \frac{1}{r!} - \frac{1}{(1+r)!}\,. \tag{6.9}$$

Den Run test wird mit

```
user@random:~ > runtest 5 128 6 100000
```

gestartet. Hierbei werden 100000 monotone Subfolgen untersucht. Subfolgen länger gleich 6 werden statistisch für den χ^2-Test zusammengefasst.

6.1.8 Maximum-of-t test

Der maximum-of-t test untersucht die Verteilung der Maxima $M(t)$ der Werte eines t-Tupels. Wenn die Werte dieser t-Tupel aus dem Intervall $[0,1)$ stammen, so sind die $M(t)^t$ in $[0,1)$ gleichverteilt. Das Programm max_of_t_test testet diese $M(t)^t$ auf Gleichverteilung. Wenn man das Programm max_of_t_test mit

```
user@random:~ > max_of_t_test 5 128 8 600 25
```

startet, wird der Test mit $t = 8$ 600-mal durchgeführt. Für den Gleichverteilungstest werden 25 Teilintervalle verwendet.

6.1.9 Collision test

Beim collision test kann der χ^2-Test nicht verwendet werden. Angenommen man verteilt 2^n Kugeln auf 2^m Urnen ($m > n$). Dann werden in einige Urnen keine Kugeln landen, in manchen genau eine und in einigen wenigen mehrere. Der collision test zählt nun, wie oft eine Kugel in eine Urne geworfen wird, in der bereits eine Kugel liegt. Um zu entscheiden, in welche Urne gerade eine Kugel geworfen wird, werden aus den i-ten Bits von m ganzzahligen Zufallszahlen ein Zahl zwischen 0 und $2^m - 1$ gebildet. i läuft dabei von 0 bis 30.

```
user@random:~ > collisiontest 5 14 20
```

Das Programm collisiontest berechnet zunächst eine Tabelle, die angibt, mit welcher Wahrscheinlichkeit man weniger Kollisionen erwarten kann als eine angegebene Schranke. Wenn weniger Kollisionen auftreten als man mit einer Warscheinlichkeit von 0,01 oder mehr als man mit einer Wahrscheinlichkeit 0,99erwarten kann, ist der Test nicht bestanden. In obigen Beispiel ist $m = 20$ und $n = 14$. Der Test wird jeweils 5-mal durchgeführt.

6.1.10 Serial correlation test

Der Korrelationskoeffizient zwischen zwei Zahlenfolgen r_i, p_i der Länge n ($n > 2$) berechnet man nach der Formel

$$C = \frac{n\sum_{i=1}^{n}(r_i p_i) - \left(\sum_{i=1}^{n} r_i\right)\left(\sum_{i=1}^{n} p_i\right)}{\sqrt{\left(n\sum_{i=1}^{n} r_i^2 - \left(\sum_{i=1}^{n} r_i\right)^2\right)\left(n\sum_{i=1}^{n} p_i^2 - \left(\sum_{i=1}^{n} p_i\right)^2\right)}}. \tag{6.10}$$

Wenn die Zahlenfolgen r_i, p_i unkorreliert sind, sollte der Korrelationskoeffizient im Intervall $[\mu_n - 2\sigma_n, \mu_n + 2\sigma_n]$ liegen ($\mu_n = \frac{-1}{n-1}$, $\sigma_n = \frac{1}{n-1}\sqrt{\frac{n(n-3)}{n+1}}$).

```
user@random:~ > serialcorrelationtest 5 128 3000
```

Durch dieses Kommando werden die Korrelationskoeffizienten zwischen je zwei verschiedenen Zufallszahlenfolgen gemessen. Dazu wird zunächst die ursprüngliche Folge mit der leapfrog-Methode in 2 bis 128 Teilfolgen zerlegt. Zwischen jeweils den ersten beiden wird der Korrelationskoeffizient fünfmal berechnet. In einem zweiten Schritt wird der Korrelationskoeffizient zwischen der ursprünglichen Folge und einer Folge, die man erhält indem man diese ursprünglichen Folge um 2^i Schritte vorstellt, berechnet. i läuft dabei von 1 bis 128. Bei jeder der fünf Berechnungen des Korrelationskoeffizienten werden $n = 3000$ Folgenglieder berücksichtigt.

6.2 Random walks

In vielen physikalischen Simulationen werden random walkers eingesetzt, so z. B. zur Beschreibung von Diffusionsprozessen oder Kristallwachstum. Die beiden folgenden Tests nutzen solche random walkers.

6.2.1 Random walk in zwei Dimensionen

Hier soll sich ein random walker auf einem quadratischen Gitter bewegen. Er startet bei den Koordinaten $p(0) = (p_x(0), p_y(0)) = (0,0)$. Um die Richtung seines nächsten Schritts zu bestimmen wird eine in $[0,1)$ gleichverteilte Zufallsvariable r gezogen. Die neue Position ist dann

$$(p_x(t+1), p_y(t+1)) = \begin{cases} (p_x(t)+1, p_y(t)) & \text{falls } 0 \leq r < \frac{1}{8} \\ (p_x(t)+1, p_y(t)+1) & \text{falls } \frac{1}{8} \leq r < \frac{2}{8} \\ (p_x(t), p_y(t)+1) & \text{falls } \frac{2}{8} \leq r < \frac{3}{8} \\ (p_x(t)-1, p_y(t)+1) & \text{falls } \frac{3}{8} \leq r < \frac{4}{8} \\ (p_x(t)-1, p_y(t)) & \text{falls } \frac{4}{8} \leq r < \frac{5}{8} \\ (p_x(t)-1, p_y(t)-1) & \text{falls } \frac{5}{8} \leq r < \frac{6}{8} \\ (p_x(t), p_y(t)-1) & \text{falls } \frac{6}{8} \leq r < \frac{7}{8} \\ (p_x(t)+1, p_y(t)-1) & \text{falls } \frac{7}{8} \leq r < 1 \end{cases} . \quad (6.11)$$

randomwalktest ist ein MPI-Programm. Alle Instanzen von randomwalktest simulieren zusammen mindestens $25 \cdot 8^7$ random walkers. Der Strom der Zufallszahlen eines Zufallszahlengenerators, der die Bewegung der einzelnen random walkers bestimmt, wird mit der leapfrog-Methode auf die einzelnen Instanzen von randomwalktest verteilt. Auf die Verteilung der Endpositionen der random walkers wird ein χ^2-Test angewandt. Mit (MPI-Implementation LAM)

```
user@random:~ > mpirun -np 24 randomwalktest 5
```

werden 24 Instanzen von randomwalktest gestartet. Der Test wird jeweils 5-mal durchgeführt. Bei anderen MPI-Implementationen als LAM kann der Aufruf etwas anders erfolgen. Der Leser konsultiere dazu bitte die Dokumentation, der von ihm verwendeten MPI-Implementation. In [Vattulainen 1995] bzw. [Vattulainen u. a. 1995] wird ein ähnlicher Test vorgeschlagen, bei dem aber nur berücksichtigt wird, in welchem Quadranten der random walker seine Wanderung beendet.

6.2.2 S_N-Test

Von [Vattulainen 1999] stammt der folgende Test. N random walkers starten bei $p_i(t) = 0$ bewegen sich gleichzeitig auf einem eindimensionalen Gitter. $S_{N,t}$, die Anzahl der zum Zeitpunkt t besuchten Gitterplätze, folgt dem Potenzgesetz $S_{N,t} \sim \sqrt{\ln N}\, t^{\gamma}$. Bewegen sich die random walkers wirklich zufällig, findet man $\gamma = \frac{1}{2}$.

```
user@random:~ > sitetest 2 2500 1000000
```

Mit diesem Aufruf startet man den S_N-Test. Hier bewegen sich gleichzeitig je zwei random walkers dem eindimensionalen Gitter und wandern je 2500 Schritte. Insgesamt werden 1 000 000 random walks vollführt. Das Programm gibt den Exponenten als Funktion der Zeit aus. Die Zufallszahlen werden durch die leapfrog-Methode und durch Vorstellen auf die je N random walkers aufgeteilt.

6.3 Monte-Carlo-Integration

Monte-Carlo-Methoden werden auch genutzt, um hochdimensionale Integrale zu berechnen. Wenn man eine Monte-Carlo-Integration durchführen will, um einen Pseudozufallszahlengenerator zu bewerten, so ist die Auswahl der zu integrierenden Testfunktion von besonderer Bedeutung. Damit die Integration nummerisch möglichst stabil ist, sollte das Integral über die Funktion Null sein. [Entacher u. a. 1998] schlägt die Funktion

$$f(x_1, x_2, \ldots, x_s) = \prod_{i=1}^{s} g(x_i, i^2) - 1 \tag{6.12}$$

mit

$$g(x, a) = \frac{|4x - 2|}{1 + a} \tag{6.13}$$

vor. Das s-dimensionale Integral dieser Funktion über $[0, 1]^s$ ist Null. Um das Integral zu berechnen, wählt man n Punkte aus $[0, 1]^s$ und berechnet

$$I = \frac{1}{n} \sum_{i=1}^{n} f(x_1, x_2, \ldots, x_s)\,, \tag{6.14}$$

was zugleich auch die Abweichung vom exakten Wert Null ist. Das Testprogramm könnte folgendermaßen gestartet werden:

```
user@random:~ > integral 32 8 16 1000000
```

Damit werden je Test 32 Integrationen durchgeführt, die Zufallszahlenfolge wird dabei in acht Teilfolgen zerlegt. Für jede Teilfolge wird eine Monte-Carlo-Integration durchgeführt und das Gesamtergebnis, das sich aus den acht Integrationen ergibt ausgegeben. Nach den 32 Integrationen wird für die Ergebnisse der Mittelwert und die Standardabweichung σ berechnet. Bei einem Vertrauensbereich von 99 % kann man das Ergebnis der Monte-Carlo-Integration im Intervall $[-0{,}485\sigma, 0{,}485\sigma]$ erwarten.

6.4 Parallelisierter WOLFF-Algorithmus am ISING-Modell

Hier sollen die Pseudozufallszahlengeneratoren an einem konkreten physikalischen System getestet werden. Betrachtet wird das zweidimensionale ISING-Modell mit Spins auf einem quadratischen Gitter, welches auf einem Torus aufgewickelt ist. Die HAMILTON-Funktion für das System lautet (kein Magnetfeld)

$$H = -J \sum_{\langle i,j \rangle} s_i s_j \qquad s_i \in \{-1, 1\}\,. \tag{6.15}$$

($\langle i, j \rangle$ heißt, dass nur über nächste Nachbarn summiert wird.) Für dieses System kann die Energie und die spezifische Wärme exakt berechnet werden. Die entsprechenden Ergebnisse findet man bei [Ferdinand und Fisher 1969]. Das unendlich große System besitzt bei $k_B T_c = \frac{2J}{\ln(1+\sqrt{2})}$ einen Phasenübergang.

Der wichtigste Algorithmus für Monte-Carlo-Simulationen stammt von METROPOLIS und ULAM [Metropolis und Ulam 1949]. In der Nähe des Phasenübergangs versagt die Methode und es dauert sehr lange, bis das simulierte System in das thermische Gleichgewicht übergeht. Beim METROPOLIS-Algorithmus werden in jedem Simulationschritt nur einzelne Spins verändert. Von WOLF [Wolff 1989] stammt ein Algorithmus, bei dem ganze Gebiete von Spin umgeklappt werden. Dieser Algorithmus liefert auch in der Nähe des Phasenübergangs gute Ergebnisse.

Es ist bekannt, dass einige Pseudozufallszahlengeneratoren, die sich in der Vergangenheit gut bewährt hatten, bei der Verwendung dieses Algorithmus drastisch falsche Ergebnisse liefern, [Ferrenberg und Landau 1992].

Beim WOLFF-Algorithmus handelt es sich um einen Clusteralgorithmus. Bei der Implementierung des Algorithmus muss man eine Datenstruktur verwalten, die die Nachbarspins des Spins, der gerade dem Cluster angelagert wird, enthält. Prinzipiell bieten sich hier eine Queue oder ein Stack an. Die Wahl der Datenstruktur beeinflusst die Art und Weise auf die der Cluster wächst [Newman und Barkema 1999]. Man kann erwarten, dass bei einem ungenügenden Pseudozufallszahlengenerator die Ergebnisse der Monte-Carlo-Simulation von der Wahl der Datenstruktur abhängen.

Die Testprogramme berechnen die Energie und die spezifische Wärme je Spin für ein 12×12-Gitter mit zyklischen Randbedingungen bei der kritischen Temperatur des unendlich großen Systems. Die Parallelisierung der Berechnung geschah hier auf die triviale Art, das heißt, jede CPU berechnet ein anderes System. Die Berechnung wird jeweils zehnmal mit 1 000 000 Monte-Carlo-Schritten durchgeführt. Daraus werden Mittelwert und Standardabweichung der Energie und der spezifischen Wärme je Spin errechnet und die auf die

Standardabweichung normierte Abweichung vom exakten Wert ausgegeben. Mit

```
user@random:~ > mpirun -np 6 ising2dwolff_queue
```

bzw.

```
user@random:~ > mpirun -np 6 ising2dwolff_stack
```

werden z. B. sechs parallele Berechnungen gestartet. Zum Schluss wird noch einmal über die sechs Rechnungen gemittelt.

6.5 Weitere Tests

Auf der der Arbeit beiliegenden CD-ROM befinden sich zwei weitere Testprogramme. Das erste, profile, misst die Zeit, die der Computer benötigt, um eine gewisse Zahl von Pseudozufallszahlen zu generieren. Das zweite dürfte besonders nützlich sein, wenn man die Bibliothek um weitere Genreatoren erweitern will. plausibility prüft ob die einzelnen Methoden zum Erzeugen der Teilfolgen usw. korrekt implementiert wurden. Der Test kann aber nicht beweisen, dass ein Generator korrekt implementiert wurde, er kann aber helfen, grobe Programmierfehler aufzudecken. Beide Programme benötigen keine Parameter.

Außerdem befindet sich auf der CD-ROM ein Programm mit dem gute Multiplikatoren für reinmultiplikative Lineare-Kongruenzen-Generatoren gesucht werden können. Das Programm spectral führt den Spektraltest in den Dimensionen zwei bis acht für prime Module kleiner 2^{31} durch. Es werden alle Multiplikatoren ausgegeben, die eine größtmögliche Periode garantieren und für die die normierten Resultate der Spektralteste in den Dimensionen zwei bis acht über einer angegebenen Schranke liegen. Die Art der Normierung wird in Abschnitt 8.1 näher beschrieben.

Implementierung

> Am Heimcomputer sitz' ich hier
> und programmier' die Zukunft mir.
>
> KRAFTWERK,
> ≫Heimcomputer≪

Die Implementation der Zufallszahlengeneratoren erfolgt in C++. Der objektorientierte Ansatz dieser Programmiersprache erlaubt dem Nutzer das komfortable Einbinden der Generatoren in eigene Applikationen. C++ ist auf einer breiten Hardwarebasis verfügbar und wird von internationalen Gremien standardisiert, was ein hohes Maß an Portabilität verspricht.

Bei der Implementierung wurde davon ausgegangen, dass der Compiler einen Ganzzahltyp long mit einer Genauigkeit von mindestens 32 Bit und einen Typ long long mit einer Genauigkeit von mindestens 64 Bit zur Verfügung stellt.

Um Namenskonflikte mit anderen Applikationen zu vermeiden, sind alle Bezeichner durch den Namensbereich TRNG gekapselt.

7.1 Allgemeine Unterprogramme

7.1.1 Versionsangabe und Fehlerbehandlung

Falls die hier beschriebene Bibliothek mit Routinen für Pseudozufallszahlengeneratoren im Zuge einer Weiterentwicklung erhebliche Änderungen erfahren sollte, ist es für den Anwender von Nutzten, wenn er leicht erkennen kann, mit welcher Version er arbeitet. Dies gestattet die Funktion const char * version(void), die einen Zeiger auf einen konstanten String mit der Versionsnummer zurück gibt.

Listing 7.1: Versionsangabe

```
const char * TRNG::version(void) {
  return("TRNG␣V1.0");
}
```

Die Funktion void warn(const char *s) gibt eine Fehlermeldung auf der Standardfehlerausgabe aus. Werden in den Unterprogrammen zur Erzeugung der Zufallszahlen Laufzeitfehler entdeckt, die eine ordnungsgemäße Weiterausführung des Programms unmöglich machen, so gibt void die(const char *s) eine Fehlermeldung aus und beendet das Programm.

Listing 7.2: Unterprogramm zur Fehlerausgabe

```
void TRNG::warn(const char *s) {
  // string to stderr
  cerr << "TRNG: " << s << endl;
}
```

Listing 7.3: Unterprogramm zur Fehlerausgabe mit Programmabbruch

```
void TRNG::die(const char *s) {
  // string to stderr and exit
  warn(s);
  exit(EXIT_FAILURE);
}
```

7.1.2 Division in primen Restklassenringen

In primen Restklassenringen kann die Kongruenz

$$ax \equiv 1 \mod p, \qquad p \in \mathbb{N}_P \tag{7.1}$$

durch Exponentiation gelöst werden.

$$x \equiv a^{p-2} \mod p \tag{7.2}$$

Für $a = 0$ wird $x = 0$ definiert. Die Funktion long trnd_modulo_invers(long a, long m) führt diese Exponentiation durch. Dazu wird die binäre Darstellung des Exponenten ausgenutzt. Eine andere Methode, das Inverse zu berechnen, ist der erweiterte EUKLIDische Algorithmus. Er findet auch das Inverse in nichtprimen Restklassenringen, so es existiert. Eine ausführliche Analyse des EUKLIDischen Algorithmus' findet man bei [Buchmann 1999].

Listing 7.4: Funktion zur Division in primen Restklassenringen

```
inline long TRNG::modulo_invers(long a, long m) {
  // a modulus invers for prime modulus m
  // 1/a mod m = a^(m-2)
  long long p, al;
  long e;
  if (a==0)
    return(0);
  else {
    al=static_cast<long long>(a);
    e=m-2;
    p=1;
```

```
    while (e>0) {
      if ((e&1)==1) {
        p*=al;
        p%=m;
      }
      al*=al;
      al%=m;
      e>>=1;
    }
    return(static_cast<long>(p));
  }
}
```

7.1.3 GAUSSscher Algorithmus

Der GAUSSsche Algorithmus ist ein Standardverfahren zum Lösen von linearen Gleichungssystemen der Form

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \equiv \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \mod m\,. \qquad (7.3)$$

Die Modulo-Funktion soll hierbei auf die einzelnen Elemente angewandt werden.

Zunächst wird die Koeffizientenmatrix diagonalisiert und dann das diagonale Gleichungssystem schrittweise gelöst. Auf die i-te Gleichung wird über das Feld $p_i = i$ zugegriffen. Ein eventuell notwendiges Austauschen der i-ten und j-ten Gleichung kann einfach durch Vertauschen von p_i und p_j erfolgen. Dies kann notwendig sein, wenn ein Diagonalelement $a_{i,i}$ der Koeffizientenmatrix beim Diagonalisieren im i-ten Umformungsschritt verschwindet. Sind alle $a_{j,i} = 0$ $(i \leq j \leq n)$, so ist die Matrix singulär und das Programm bricht ab.

Durch das Vertauschen der Gleichungen, liegen die Elemente im Lösungsvektor in falscher Reihenfolge vor und müssen noch nachträglich sortiert werden.

Die Funktion void trnd_gauss(std::vector⟨long⟩ &a, std::vector⟨long⟩ &b, long m) gibt die Lösung des Gleichungssystems im Vector b zurück, übergeben werden die Koeffizientenmatrix in a und die inhomogene rechte Seite in b. m ist der Modul.

Listing 7.5: GAUSSscher Algorithmus in modularer Arithmetik

```
void TRNG::gauss(vector<long> &a, vector<long> &b, long m) {
  // Gauss algorithm in modular aithmetic
  // a[0..n^2-1] : matrix
  // b[0..n-1]   : vector
  // m           : modulus
  // returns in b the solution
```

```
long i, j, k, t, n;
// initialize indices
n=b.size();
vector<long> p(n);
for (i=0; i<n; ++i)
  p[i]=i;
// make matrix diagonal
for (i=0; i<n-1; ++i) {
  if (a[n*p[i]+i]==0) {
    // swap
    j=i+1;
    while (j<n && a[n*p[j]+i]==0)
      j++;
    if (j==n)
      die("singular matrix in TRNG::gauss");
    t=p[i];  p[i]=p[j];  p[j]=t;
  }
  t=modulo_invers(a[n*p[i]+i], m);
  for (j=i; j<n; j++)
    a[n*p[i]+j]=static_cast<long>
      ((static_cast<long long>(a[n*p[i]+j])*
        static_cast<long long>(t))%m);
  b[p[i]]=static_cast<long>
    ((static_cast<long long>(b[p[i]])*
      static_cast<long long>(t))%m);
  for (j=i+1; j<n; j++) {
    if (a[n*p[j]+i]!=0) {
      t=modulo_invers(a[n*p[j]+i], m);
      for (k=i; k<n; k++) {
        a[n*p[j]+k]=
          static_cast<long>
          ((static_cast<long long>(a[n*p[j]+k])*
            static_cast<long long>(t))%m);
        a[n*p[j]+k]-=a[n*p[i]+k];
        if (a[n*p[j]+k]<0)
          a[n*p[j]+k]+=m;
      }
      b[p[j]]=static_cast<long>
        ((static_cast<long long>(b[p[j]])*
          static_cast<long long>(t))%m);
      b[p[j]]-=b[p[i]];
      if (b[p[j]]<0)
        b[p[j]]+=m;
    }
  }
}
// solve diagonal system
t=modulo_invers(a[n*p[n-1]+n-1], m);
a[n*p[n-1]+n-1]=1;
b[p[n-1]]=static_cast<long>
  ((static_cast<long long>(b[p[n-1]])*
    static_cast<long long>(t))%m);
for (i=n-2; i>=0; i--)
  for (j=i+1; j<n; j++) {
    b[p[i]]-=static_cast<long>
      ((static_cast<long long>(a[n*p[i]+j])*
        static_cast<long long>(b[p[j]]))%m);
    if (b[p[i]]<0)
      b[p[i]]+=m;
  }
// sort
```

```
  for (i=0; i<n; ++i)
    p[i]=b[p[i]];
  for (i=0; i<n; ++i)
    b[i]=p[i];
}
```

7.1.4 Matrix-Matrix-Multiplikation und Matrix-Vektor-Multiplikation

Mit void trnd_matrix_mult(std::vector⟨long⟩ &a, std::vector⟨long⟩ &b, std::vector⟨long⟩ &c, long m) und void trnd_matrix_vec_mult(std::vector⟨long⟩ &a, std::vector⟨long⟩ &b, std::vector⟨long⟩ &c, long m) stehen zwei Funktionen zur Matrix-Matrix- bzw. zur Matrix-Vektor-Multiplikation zur Verfügung. Die Berechnungen erfolgen wieder in modularer Arithmetik (Modulus m). Es können $n \times n$-Matrizen und $n \times n$-Matrizen bzw. $n \times n$-Matrizen und Vektoren mit n Elementen miteinander multipliziert werden. Es werden a und b miteinander multipliziert, das Ergebnis wird in c gespeichert.

Listing 7.6: Matrix-Matrix-Multiplikation in modularer Arithmetik

```
void TRNG::matrix_mult(vector<long> &a, vector<long> &b,
                       vector<long> &c, long m) {
  // multiply two matrices a and b
  // c=a*b
  // n : dimension of a and b
  // m : modulus
  long i, j, k, n;
  long long t;
  n=a.size()<b.size() ? a.size() : b.size();
  if (a.size()!=b.size())
    warn("different sized vectors in TRNG::matrix_mult");
  n=static_cast<long>(sqrt(static_cast<double>(n)));
  for (i=0; i<n; ++i)
    for (j=0; j<n; j++) {
      t=0;
      for (k=0; k<n; k++) {
        t+=(static_cast<long long>(a[j*n+k])*
            static_cast<long long>(b[k*n+i]))%m;
        if (t>=m)
          t-=m;
      }
      c[j*n+i]=static_cast<long>(t);
    }
}
```

Listing 7.7: Matrix-Vektor-Multiplikation in modularer Arithmetik

```
void TRNG::matrix_vec_mult(vector<long> &a, vector<long> &b,
                           vector<long> &c, long m) {
  // multiply a matrix a and a vector b
  // c=a*b
  // n : dimension of a and b
  // m : modulus
  long j, k, n;
```

```
    long long t;
    n=b.size();
    if (a.size()!=b.size()*b.size())
      warn("different_sized_vectors_in_TRNG::matrix_vec_mult");
    for (j=0; j<n; j++) {
      t=0;
      for (k=0; k<n; k++) {
        t+=(static_cast<long long>(a[j*n+k])*
            static_cast<long long>(b[k]))%m;
        if (t>=m)
          t-=m;
      }
      c[j]=static_cast<long>(t);
    }
  }
```

7.2 Die Zufallszahlengeneratoren

7.2.1 Die Elternklasse

Jeder Zufallszahlengenerator wird als eigene Klasse implementiert. Diese Klassen werden von einer übergeordneten Klasse class RNG abgeleitet. In ihr sind alle für einen Zufallszahlengenerator typischen Methoden implementiert. Generatorspezifische Methoden sind zunächst leer und als virtuell deklariert.

Listing 7.8: Klasse RNG von der später alle Klassen für die Zufallszahlengeneratoren abgeleitet werden

```
  class RNG {
    // generic class for random number generators
  protected:
    long max_val;
    long max_val2;
  public:
    static const RNG_type type=RNG_t;
    virtual const char * name(void);
    virtual void reset(void);
    virtual void seed(long);
    virtual long rand(void);
    bool boolean(void);
    bool boolean(double);
    double uniform(void);
    double uniformco(void);
    double uniformcc(void);
    double uniformoo(void);
    double uniformoc(void);

    double exp_dist(double=1.0);
    double normal_dist(double=1.0, double=0.0);
    double Gamma_dist(double);
    double Beta_dist(double, double);
    double chi_square_dist(double);
    double Student_t_dist(double);
    long binomial_dist(long, double=0.5);
    long poisson_dist(double=1.0);
```

```
    double rejection(double (*)(double), double, double, double);
    vector2d spherical2d(void);
    vector3d spherical3d(void);
    vector4d spherical4d(void);

    long max(void);
    virtual RNG *copy(void);
    virtual void split(long, long);
    virtual void jump(long);
    virtual void save_status(std::vector<long> &);
    virtual void load_status(std::vector<long> &);
    RNG();
};
```

Die Funktion const char * RNG::name(void) gibt einen Zeiger auf einen konstanten String mit dem Namen des verwendeten Generators zurück. Das Attribut type hat einen Wert des Aufzählungstyps enum RNG_type.

Listing 7.9: der Aufzählungstyp enum RNG_type

```
enum RNG_type {RNG_t, generic_MLCG_t, ParkMiller_t, RAND_t, LCG64_t,
               LMC2_t, LMC3_t, LMC4_t, CLCG2_t, CLCG3_t, CLCG4_t,
               EINV_t, EINVLCG64_t, user1_t, user2_t, user3_t};
```

Mit diesem Attribut kann der gerade verwendete Generator leicht identifiziert werden. Die Werte user1_t, user2_t und user3_t sind für Erweiterungen anderer Programmierer gedacht, die anderen korrespondieren mit den später beschriebenen Pseudozufallszahlengeneratoren.

void RNG::reset(void) versetzt den Generator in einen definierten Ausgangszustand, die Methode void RNG::seed(long s=0l) legt nur einen neuen Startwert fest. Das Argument dieser Funktion muss nichtnegativ sein. Kernstück eines jeden Generators ist eine Methode long RNG::rand(void), die Zahlen vom Typ long zwischen 0 und einer oberen Schranke, die mit long RNG::max(void) abgefragt werden kann, berechnet.

Um verschiedene gleichverteilte Zufallszahlen aus verschiedenen Intervallen zu erzeugen, besitzt die Klasse RNG die Methoden

- double RNG::uniform(void) für gleichverteilte Zahlen aus $[0,1)$,
- double RNG::uniformco(void) für gleichverteilte Zahlen aus $[0,1)$,
- double RNG::uniformcc(void) für gleichverteilte Zahlen aus $[0,1]$,
- double RNG::uniformoo(void) für gleichverteilte Zahlen aus $(0,1)$ und
- double RNG::uniformoc(void) für gleichverteilte Zahlen aus $(0,1]$.

Diese Methoden wandeln den Rückgabewert von long RNG::rand(void) in eine Fließkommazahl aus einem offenen oder geschlossenen Einheitsintervall. bool RNG::boolean(void) gibt mit einer Wahrscheinlichkeit von 0,5 einen Wert true zurück, bei bool RNG::boolean(double p) kann die Wahrscheinlichkeit dafür, dass der Rückgabewert true ist, explizit durch das Argument übergeben werden.

Außerdem besitzt die Klasse Methoden zur Generierung nichtgleichverteilter Zufallszahlen. Folgende Verteilungen wurden implementiert:

Exponentialverteilung: Die Exponentialverteilung ist durch die Verteilungsfunktion

$$P(x) = 1 - \mathrm{e}^{-\frac{x}{\mu}} \qquad x, \mu \in \mathbb{R} \quad x, \mu \geq 0 \tag{7.4}$$

gekennzeichnet und kann mit der Methode void RNG::exp_dist(double mu=1.0), erzeugt werden. Exponentialverteilte Zufallszahlen x erhält man einfach aus der Transformation

$$x = -\mu \ln y\,, \tag{7.5}$$

wobei y in $(0, 1)$ gleichverteilt ist.

Normalverteilung: Die Normalverteilung besitzt die Verteilungsfunktion

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{e}^{-\frac{(t-\mu)^2}{2\sigma^2}}\,\mathrm{d}t \qquad x, \mu, \sigma \in \mathbb{R} \quad \sigma > 0 \tag{7.6}$$

mit der Standardabweichung σ und dem Mittelwert μ. Normalverteilte Zufallszahlen erhält man mit der Methode void RNG::normal_dist(double sigma=1.0, double mean=0.0). Die Berechnung erfolgt aus gleichverteilten Zufallszahlen nach der oben beschiebenen Transformationsmethode, die hier auch BOX-MÜLLER-Methode heißt. Die Verteilungsdichte zweier voneinander unabhängiger normalverteilter Zufallsvariablen x_1 und x_2 (mit der Standardabweichung $\sigma = 1$ und dem Mittelwert $\mu = 0$) lautet

$$p(x_1, x_2) = \frac{1}{2\pi} \mathrm{e}^{-\frac{x_1^2+x_2^2}{2}}\,. \tag{7.7}$$

Die Wahrscheinlichkeit, dass der Punkt (x_1, x_2) in $[x_1, x_1 + \mathrm{d}x_1] \times [x_2, x_2 + \mathrm{d}x_2]$ liegt, beträgt dann

$$p(x_1, x_2)\,\mathrm{d}x_1\,\mathrm{d}x_2 = \frac{1}{2\pi} \mathrm{e}^{-\frac{x_1^2+x_2^2}{2}}\,. \tag{7.8}$$

und in Polarkoordinaten

$$p(r, \varphi)\,\mathrm{d}r\,\mathrm{d}\varphi = \underbrace{\frac{1}{2\pi}}_{p(\varphi)} \underbrace{r\mathrm{e}^{-\frac{r^2}{2}}}_{p(r)} \mathrm{d}r\,\mathrm{d}\varphi\,. \tag{7.9}$$

$$y_1 = P(r) = \int_0^r p(t)\mathrm{d}t = 1 - \mathrm{e}^{-\frac{r^2}{2}} \tag{7.10}$$

$$y_2 = P(\varphi) = \int_0^{\varphi} p(\varphi)\mathrm{d}t = \frac{1}{2\pi}\varphi \tag{7.11}$$

Wenn y_1 gleichverteilt ist, so ist dies auch $1 - y_1$. Somit lautet die Transformation zwischen den auf der Einheitskreisscheibe gleichverteilten x_1 und x_2 und den normalverteilten y_1 und y_2

$$\begin{pmatrix} \mathrm{e}^{-\frac{x_1^2+x_2^2}{2}} \\ \frac{1}{2\pi}\arctan\frac{x_2}{x_1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{7.12}$$

und deren Inverse

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \sqrt{-2\ln y_1}\cos 2\pi y_2 \\ \sqrt{-2\ln y_1}\sin 2\pi y_2 \end{pmatrix}. \tag{7.13}$$

Die dazu nötigen Zufallspunkte auf der Einheitskreisfläche werden nach der Verwerfungsmethode berechnet, wozu erst Punkte aus dem Quadrat $[-1,1]\times[-1,1]$ gewählt werden, siehe auch [Knuth 1981a], [Press u. a. 1997] oder [Vesely 1993]. Normalverteilte Zufallszahlen z mit der Standardabweichung σ und dem Mittelwert μ erhält man durch die Transformation $z_{1,2} = \sigma x_{1,2} + \mu$.

Γ-Verteilung: Hier lautet die Verteilungsfunktion

$$P(x) = \frac{1}{\Gamma(b)}\int_0^x t^{b-1}\mathrm{e}^{-t}\,\mathrm{d}t \qquad x, b, \in \mathbb{R} \quad x \le 0 \quad b > 0\,. \tag{7.14}$$

Der implementierte Algorithmus stammt aus [Knuth 1981a] und ist für Verteilungen mit $b > 1$ korrekt. Der Fall $b < 1$ wird von einem aus [Barnett et al. 1996] stammenden Algorithmus behandelt. Für $b = 1$ entartet die Γ-Verteilung zur Exponentialverteilung. Die Methode zur Generierung von Pseudozufallszahlen heißt double RNG::Gamma_dist(double b).

***B*-Verteilung** Die Betaverteilung wird durch die Verteilungsfunktion

$$P(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\int_0^x t^{a-1}(1-t)^{b-1}\,\mathrm{d}t \qquad 0 \le x \le 1 \quad a > 0 \quad b > 0 \tag{7.15}$$

definiert. Die Methode double Beta_dist(double a, double b) erzeugt Zufallszahlen, die der B-Verteilung genügen.

χ^2-Verteilung Für die χ^2-Verteilung mit ν ($\nu \in \mathbb{N}$) Freiheitsgraden gilt die Verteilungsfunktion

$$P(x) = \frac{1}{2^{\frac{\nu}{2}}\Gamma(\frac{\nu}{2})}\int_0^x t^{\frac{\nu}{2}-1}\mathrm{e}^{-\frac{x}{2}}\,\mathrm{d}t\,. \tag{7.16}$$

Die Methode double RNG::chi_square_dist(double nu) berechnet aus Γ-verteilten Zufallszahlen Zufallszahlen mit χ^2-Verteilung, denn die χ^2-Verteilung mit ν Freiheitsgraden entspricht einer Γ-Verteilung mit dem Parameter $b = \frac{\nu}{2}$, wenn man zusätzlich noch $x \to 2x$ transformiert.

STUDENTsche-t-Verteilung: Die Methode double RNG::Student_t_dist(double nu) berechnet Zufallszahlen, die der Verteilungsfunktion

$$P(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{2\nu}\,\Gamma(\frac{\nu}{2})} \int_{-\infty}^{x} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}} \mathrm{d}t \tag{7.17}$$

genügt. Diese Verteilungsfunktion heißt STUDENTsche-t-Verteilung. Der Algorithmus stammt aus [Barnett et al. 1996].

Binomialverteilung: Die Binomialverteilung ist eine diskrete Verteilungsfunktion, die durch die Parameter p ($p \in (0,1)$) und t ($t \in \mathbb{N}$) charakterisiert wird. Sie beschreibt wie oft ein Ereignis nach t unabhängigen Versuchen eingetreten ist, wenn die Wahrscheinlichkeit für das Eintreten des Ereignisses p beträgt. Die Verteilungsfunktion, die die Wahrscheinlichkeit dafür abgibt, dass eine binomialverteilte Zufallsvariable einen Wert kleiner gleich n annimmt, lautet

$$P(n) = \sum_{i=0}^{n} \binom{t}{i} p^i (1-p)^{t-i}\,. \tag{7.18}$$

Die Methode long RNG::binomial_dist(long t, double p=0.5) erzeugt t Pseudozufallszahlen aus $[0,1)$, zählt wieviele kleiner als p sind und gibt diese Zahl zurück.

Poissonverteilung: Lässt man in der Binomialverteilung p gegen 0 und t so gegen ∞ gehen, dass das Produkt $\mu = pt$ konstant bleibt, so erhält man die Poissonverteilung, deren Verteilungsfunktion

$$P(n) = \sum_{i=0}^{n} \frac{\mu^i \mathrm{e}^{-\mu}}{i!} \tag{7.19}$$

lautet. Die Methode long RNG::poisson_dist(double mu=1.0) generiert mit einem Algorithmus nach [Barnett et al. 1996] poissonverteilte Zufallszahlen mit dem Mittelwert μ.

Mit der Methode double RNG::rejection(double (*f)(double), double a1, double a2, double p_max) wurde die Zurückweisungsmethode implementiert. Die Funktion f beschreibt die Wahrscheinlichkeitsdichte über dem Intervall $[a_1, a_2]$. Das Maximum $p_{\max}$ der Funktion f über dem Intervall $[a_1, a_2]$ muss an die Methoden übergeben werden. Das Argument $p_{\max}$ kann auch eine Zahl

größer diesem Maximum sein, dann wird der Algorithmus aber schnell ineffektiv. Die Methode gibt Zufallszahlen, die der durch f beschriebenen Verteilung entsprechen zurück.

Oft benötigt man in Anwendungen Einheitsvektoren aus $\mathbb{R}^n$ mit zufällig gewählter Richtung. Diese berechnet man für $n \in \{2, 3, 4\}$ leicht mit den Methoden

- vector2d spherical2d(void),
- vector3d spherical3d(void) und
- vector4d spherical4d(void).

Die Algorithmen zur Berechnung dieser Einheitsvektoren stammen aus [Vesely 1993] bzw. [Barnett et al. 1996]. Diese drei Methoden geben Strukturen zurück, die die Koordinaten aufnehmen.

Listing 7.10: Definition der Vektor-Strukturen

```
typedef struct {
  double x1, x2;
} vector2d;

typedef struct {
  double x1, x2, x3;
} vector3d;

typedef struct {
  double x1, x2, x3, x4;
} vector4d;
```

Manchmal möchte man eine Instanz eines Generators vervielfältigen, dies ist mit der Methode RNG *copy(void) möglich. Nach dem man sich zunächst mehrere identische Generatoren erzeugt hat, kann man diese nun so manipulieren, dass sie garantiert disjunkte Subfolgen liefern. Dies geschieht mit den Methoden void RNG::split(long s, long n) und RNG::jump(long s).

void RNG::split(long s, long n) erlaubt es, den Generator so zu verändern, dass er von nun an nur noch jedes s-te Glied der Folge von Zufallszahlen zurück gibt. Vorher springt er jedoch noch n Schritte weiter. Die Methode void RNG::split(long s, long n) macht also aus einem Generator, der die Folge r_i liefert, einen neuen mit der Folge $t_{n,i}$, vgl. Formel (4.2). RNG::jump(long s) stellt den Generator um 2^s Schritte vor. Sie wird für die Parallelisierung nach Formel (4.1) benötigt.

Den Status eines jeden Zufallszahlengenerators kann man mit der Methode void RNG::savestatus(vector⟨long⟩ &a) abspeichern. Diese Methode verändert die Länge und die Werte des Vectors. Der Status eines Generators kann mit void RNG::loadstatus(vector⟨long⟩ &a) später wieder hergestellt werden. Das erste Element des Vectors enthält eine Prüfzahl, die sichern soll, dass ein zuvor mit void RNG::savestatus(vector⟨long⟩ &a) gespeicherter Status nur

einem Generator vom gleichen Typ zugewiesen werden kann. Prinzipiell stellen diese beiden Funktionen auch eine Schnittstelle da, mit der man die Parameter der Generatoren beliebige Werte zuweisen kann. Davon sollte der Benutzer nur Gebrauch machen, wenn er genau weiß, was er da tut.

Listing 7.11: Implementation der Klasse RNG von der später alle Klassen für die Zufallszahlengeneratoren abgeleitet werden

```
inline const char * TRNG::RNG::name(void) {
  return("RNG");
}

inline void TRNG::RNG::reset(void) {
}

inline void TRNG::RNG::seed(long s=0l) {
}

inline long TRNG::RNG::rand(void) {
  return(0l);
}

inline bool TRNG::RNG::boolean(void) {
  return((rand()<max_val2) ? true : false);
}

inline bool TRNG::RNG::boolean(double p) {
  return((rand()<static_cast<long>(p*max())+1.0) ? true : false);
}

inline double TRNG::RNG::uniform(void) {
  return(uniformco());
}

inline double TRNG::RNG::uniformco(void) {
  return(static_cast<double>(rand())/
         (static_cast<double>(max())+1.0));
}

inline double TRNG::RNG::uniformcc(void) {
  return(static_cast<double>(rand())/
         static_cast<double>(max()));
}

inline double TRNG::RNG::uniformoo(void) {
  return((static_cast<double>(rand())+1.0)/
         (static_cast<double>(max())+2.0));
}

inline double TRNG::RNG::uniformoc(void) {
  return((static_cast<double>(rand())+1.0)/
         (static_cast<double>(max())+1.0));
}

double TRNG::RNG::exp_dist(double mu) {
  double t;
  if (mu<=0.0)
    die("negative or zero parameter in TRNG::RNG::exp_dist");
  t=uniformoc();
  return(-mu*log(t));
```

```
}

double TRNG::RNG::normal_dist(double sigma, double mean) {
  static double s, t1, t2;
  static bool calc=true;
  if (sigma<=0.0)
    die("negative or zero standard deviation TRNG::RNG::normal_dist");
  if (calc) {
    calc=false;
    do {
      t1=2.0*uniformco()-1.0;
      t2=2.0*uniformco()-1.0;
      s=t1*t1+t2*t2;
    } while (s>=1.0);
    s=sqrt(-2.0*log(s)/s);
    return(t1*s*sigma+mean);
  } else {
    calc=true;
    return(t2*s*sigma+mean);
  }
}

double TRNG::RNG::Gamma_dist(double p1) {
  const double e=2.71828182845904524;
  const double pi=3.1415926535897932;
  double t1, t2, v, x, y, u1, u2, v1, v2;
  if (p1<=0.0)
    die("parameter less or equal 0.0 in TRNG::RNG::Gamma_dist");
  if (p1<1.0) {
    v1=(e+p1)/e;
    while (true) {
      u1=uniformco();
      u2=uniformco();
      v2=v1*u1;
      if (v2<=1.0) {
        x=pow(v2, 1.0/p1);
        if (u2<=exp(-x))
          return(x);
      } else {
        x=-log((v1-v2)/p1);
        if (u2<=pow(x, p1-1.0))
          return(x);
      }
    }
  }
  if (p1>1.0) {
    t1=sqrt(2.0*p1-1.0);
    t2=p1-1.0;
    do {
      do {
        y=tan(pi*uniformco());
        x=t1*y+t2;
      } while (x<=0.0);
      v=uniformco();
    } while (v>(1.0+y*y)*exp(t2*log(x/t2)-t1*y));
    return(x);
  }
  return(exp_dist(1.0));
}

double TRNG::RNG::Beta_dist(double a, double b) {
```

```
  double t1, t2;
  if (a<=0.0 || b<=0.0)
    die("negative or zero parameter in TRNG::RNG::Beta_dist");
  if (a>1.0 || b>1.0) {
    t1=Gamma_dist(a);
    t2=Gamma_dist(b);
  } else {
    a=1.0/a;
    b=1.0/b;
    do {
      t1=pow(uniformco(), a);
      t2=pow(uniformco(), b);
    } while (t1+t2>1.0);
  }
  return(t1/(t1+t2));
}

double TRNG::RNG::chi_square_dist(double nu) {
  if (nu<1.0)
    die("parameter less than one in TRNG::RNG::chi_square_dist");
  return(2.0*Gamma_dist(0.5*nu));
}

double TRNG::RNG::Student_t_dist(double nu) {
  if (nu<=0.0)
    die("parameter less than zero in TRNG::RNG::Student_t_dist");
  return(normal_dist(1.0, 0.0)*sqrt(2.0*nu/Gamma_dist(0.5*nu)));
}

long TRNG::RNG::binomial_dist(long t, double p) {
  long k, j;
  if (p<=0.0 || p>=1.0)
    die("probability <=0.0 or >=1.0 in TRNG::RNG::binomial_dista");
  if (t<=0l)
    die("less than one trail TRNG::RNG::binomial_dista");
  k=0l;
  for (j=0l; j<t; ++j)
    if (boolean(p))
      ++k;
  return(k);
}

long TRNG::RNG::poisson_dist(double mu) {
  long k;
  double A;
  if (mu<=0)
    die("parameter less than 0 in TRNG::RNG::poisson_dista");
  if (mu<10l) {
    mu=exp(-mu);
    k=-1l;
    A=1.0;
    do {
      ++k;
      A*=uniformco();
    } while (A>mu);
    return(k);
  } else {
    k=static_cast<long>(rint(normal_dist(sqrt(mu), mu)));
    return(k>=0l ? k : 0l);
  }
}
```

```
double TRNG::RNG::rejection(double (*f)(double), double a1, double a2,
                            double p_max) {
  double t1, t2;
  do {
    t1=uniformco()*(a2-a1)+a1;
    t2=uniformco()*p_max;
  } while ((*f)(t1)<t2);
  return(t1);
}

vector2d TRNG::RNG::spherical2d(void) {
  double r2, t1, t2, t12, t22;
  vector2d vec;
  do {
    t1=2.0*uniformco()-1.0;
    t2=uniformco();
    t12=t1*t1;
    t22=t2*t2;
    r2=t12+t22;
  } while (r2>1.0);
  vec.x1=2.0*t1*t2/r2;
  vec.x2=(t12-t22)/r2;
  return(vec);
}

vector3d TRNG::RNG::spherical3d(void) {
  double q, r2, t1, t2;
  vector3d vec;
  do {
    t1=2.0*uniformco()-1.0;
    t2=2.0*uniformco()-1.0;
    r2=t1*t1+t2*t2;
  } while (r2>1.0);
  q=2.0*sqrt(1.0-r2);
  vec.x1=t1*q;
  vec.x2=t2*q;
  vec.x3=1-2*r2;
  return(vec);
}

vector4d TRNG::RNG::spherical4d(void) {
  double q, r21, r22, t1, t2, t3, t4;
  vector4d vec;
  do {
    t1=2.0*uniformco()-1.0;
    t2=2.0*uniformco()-1.0;
    r21=t1*t1+t2*t2;
  } while (r21>1.0);
  do {
    t3=2.0*uniformco()-1.0;
    t4=2.0*uniformco()-1.0;
    r22=t3*t3+t4*t4;
  } while (r22>1.0);
  q=sqrt((1.0-r21)/r22);
  vec.x1=t1;
  vec.x2=t2;
  vec.x3=t3*q;
  vec.x4=t4*q;
  return(vec);
}
```

```
inline long TRNG::RNG::max(void) {
  return(max_val);
}

RNG *TRNG::RNG::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new RNG);
  save_status(s);
  R->load_status(s);
  return(R);
}

inline void TRNG::RNG::split(long s, long n) {
  warn("TRNG::RNG::split not implemented");
}

inline void TRNG::RNG::jump(long s) {
  warn("TRNG::RNG::jump not implemented");
}

void TRNG::RNG::save_status(vector<long> &s) {
  warn("TRNG::RNG::save_status not implemented");
  s.resize(1, 0);
  s[0]=type;
}

void TRNG::RNG::load_status(vector<long> &s) {
  warn("TRNG::RNG::load_status not implemented");
  if ((s.size()!=1) || (s[0]!=type))
    warn("TRNG::RNG::load_status wrong parameter");
}

inline TRNG::RNG::RNG() {
}
```

7.2.2 Generischer rein multiplikativer Lineare-Kongruenzen-Generator

Einige der weiteren Generatoren bauen auf verschiedenen rein multiplikativen Lineare-Kongruenzen-Generatoren auf. Deshalb ist es sinnvoll zunächst eine allgemeine Klasse für diesen Typ von Generatoren zu schreiben. Diese Klasse wird öffentlich von der Klasse RNG abgeleitet. Der Konstruktor erwartet den Multiplikator, den primen Modulus und den Startwert als Parameter. Die private Methode void generic_MLCG::backward(void) stellt den Generator um einen Schritt zurück. Man beachte, dass die Methode long generic_MLCG::vall(void) nie Null zurück gibt.

Listing 7.12: Klassendefinition für generischen rein multiplikativen lineare Kongruenzengenerator

```
  class generic_MLCG : public RNG {
    // generic class for multiplicative linear conguential
    // random number generators
  private:
```

```
    long a;
    long a_save;
    long r;
    long r_save;
    long modulus;
    long modulus_save;
    void backward(void);
  public:
    static const RNG_type type=generic_MLCG_t;
    const char * name(void);
    void reset(void);
    void seed(long);
    long rand(void);
    RNG *copy(void);
    void split(long, long);
    void jump(long);
    void save_status(std::vector<long> &);
    void load_status(std::vector<long> &);
    generic_MLCG(long, long, long);
  };
```

Listing 7.13: Implementation für generischen rein multiplikativen lineare Kongruenzengenerator

```
inline const char * TRNG::generic_MLCG::name(void) {
  return("generic_MLCG");
}

void TRNG::generic_MLCG::backward(void) {
  r=static_cast<long>
    ((static_cast<long long>(TRNG::modulo_invers(a, modulus))*
      static_cast<long long>(r))%modulus);
}

inline void TRNG::generic_MLCG::reset(void) {
  a=a_save;
  r=r_save;
  modulus=modulus_save;
  max_val=modulus-1l;;
  max_val2=max_val/2l;
}

inline void TRNG::generic_MLCG::seed(long s=0l) {
  s++;
  if ((s%modulus)<=0l)
    warn("bad_seed_for_TRNG::generic_MLCG::seed");
  r=s%modulus;
}

inline long TRNG::generic_MLCG::rand(void) {
  r=static_cast<long>
    ((static_cast<long long>(a)*static_cast<long long>(r))%modulus);
  return(r);
}

RNG *TRNG::generic_MLCG::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new generic_MLCG(1l, 1l, 1l));
  save_status(s);
  R->load_status(s);
```

```
  return(R);
}

inline void TRNG::generic_MLCG::split(long s, long n) {
  long b, c, i;
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::generic_MLCG::split");
  if (s>1l) {
    b=1l;
    c=a;
    for (i=0l; i<=n; ++i)
      rand();
    while (s>0) {
      if (s&1l==1l)
        b=static_cast<long>((static_cast<long long>(c)*
                             static_cast<long long>(b))%modulus);
      s>>=1l;
      c=static_cast<long>((static_cast<long long>(c)*
                           static_cast<long long>(c))%modulus);
    }
    a=b;
    backward();
  }
}

inline void TRNG::generic_MLCG::jump(long s) {
  long i, t;
  if (s<0l)
    warn("invalid arguments for TRNG::generic_MLCG::jump");
  else {
    t=a;
    for (i=0; i<s; ++i)
      a=static_cast<long>((static_cast<long long>(a)*
                           static_cast<long long>(a))%modulus);
    rand();
    a=t;
  }
}

void TRNG::generic_MLCG::save_status(vector<long> &s) {
  s.resize(4);
  s[0]=type;
  s[1]=a;
  s[2]=r;
  s[3]=modulus;
}

void TRNG::generic_MLCG::load_status(vector<long> &s) {
  if ((s.size()!=4) || (s[0]!=type))
    warn("TRNG::generic_MLCG::load_status wrong parameter");
  else {
    a=s[1];
    r=s[2];
    modulus=s[3];
    max_val=modulus-1l;;
    max_val2=max_val/2l;
  }
}

TRNG::generic_MLCG::generic_MLCG(long a0=16807l, long m=2147483647l,
                                 long r0=1l) {
```

```
    modulus=m;
    a=a0;
    max_val=modulus-1l;
    max_val2=max_val/2l;
    r_save=r0;
    a_save=a0;
    modulus_save=m;
  }
```

7.2.3 Generator nach PARK und MILLER

PARK und MILLER schlagen in [Park und Miller 1988] einen Minimalstandard für Zufallszahlengeneratoren vor.[1] Dieser ist in der Klasse class ParkMiller realisiert. Dabei handelt es sich um einen Generator vom Typ der linearen Kongruenzen mit

$$r_i \equiv 16807 \cdot r_{i-1} \mod (2^{31} - 1). \tag{7.20}$$

Dieser Generator hat eine Periode von $2^{31} - 2$. Der Modulus $2^{31} - 1$ ist prim. Die vorliegende Implementation gibt jedoch $r_n - 1$ zurück, so dass die erzeugten Ganzzahlwerte einschließlich von 0 bis $2^{31} - 3$ reichen. Die Klasse class ParkMiller wird von class RNG abgeleitet und ist mit class GenericMLCG implementiert. Da die Periode des Generators recht kurz ist, soll er eher als Vergleichsmaßstab für die folgenden Generatoren dienen als als praktisches Werkzeug.

Listing 7.14: Klassendefinition für Zufallszahlengenerator nach PARK und MILLER

```
class ParkMiller : public RNG {
  // modified random number generator by Park and Miller
  // see Communication of the ACM, Oct. 1988, Vol. 31, Nr. 10,
  // pp. 1192-1201
private:
  generic_MLCG R;
public:
  static const RNG_type type=ParkMiller_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  ParkMiller();
};
```

[1] Laut [James 1990] wurde dieser Generator unter der Bezeichnung SURAND schon 1968 von IBM verwendet.

Listing 7.15: Implementation für Zufallszahlengenerator nach PARK und MILLER

```
inline const char * TRNG::ParkMiller::name(void) {
  return("ParkMiller");
}

inline void TRNG::ParkMiller::reset(void) {
  R.reset();
  max_val=R.max()-1l;
  max_val2=max_val/2l;
}

inline void TRNG::ParkMiller::seed(long s=0l) {
  R.seed(s);
}

inline long TRNG::ParkMiller::rand(void) {
  return(R.rand()-1l);
}

RNG *TRNG::ParkMiller::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new ParkMiller);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::ParkMiller::split(long s, long n) {
  R.split(s, n);
}

void TRNG::ParkMiller::jump(long s) {
  if (s<0l)
    warn("invalid arguments for TRNG::ParkMiller::jump");
  else
    R.jump(s);
}

void TRNG::ParkMiller::save_status(vector<long> &s) {
  R.save_status(s);
  s[0]=type;
}

void TRNG::ParkMiller::load_status(vector<long> &s) {
  if ((s.size()!=4) || (s[0]!=type))
    warn("TRNG::ParkMiller::load_status wrong parameter");
  else {
    s[0]=generic_MLCG_t;
    R.load_status(s);
    s[0]=type;
  }
}

inline TRNG::ParkMiller::ParkMiller() : R(16807l, 2147483647l, 1l) {
  reset();
}
```

7.2.4 RAND

In der Literatur (z. B. [James 1990]) findet man immer wieder einen Lineare-Kongruenzen-Generator mit dem Multiplikator 69069

$$r_i \equiv 69069 r_{i-1} + 1 \mod 2^{32} \tag{7.21}$$

Dieser Generator erreicht eine Periode von 2^{32}. Er ist zwar sehr schnell, für einige Anwendungen ist er aber wenig geeignet. Z. B. wiederholen sich sie untersten Bits in kurzen Zyklen. [James 1990] meint sogar zum Multiplikator 69069:

> Its best property is that it is easy to remember.

Die Methode long RAND::vall(void) gibt nicht r_i sondern $\left\lfloor \frac{r_i}{2} \right\rfloor$ zurück. Bei der Implementation der Methode zum Vorstellen dieses Generators sind einige Besonderheiten zu nennen, die aber auch für den den nächsten Generator gelten und deshalb dort erläutert werden.

Listing 7.16: Klassendefinition für Zufallszahlengenerator RAND

```
class RAND : public RNG {
private:
  unsigned long a, b, r;
  void backward(void);
public:
  static const RNG_type type=RAND_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  RAND();
};
```

Listing 7.17: Implementation für Zufallszahlengenerator RAND

```
inline const char * TRNG::RAND::name(void) {
  return("RAND");
}

inline void TRNG::RAND::backward(void) {
  long i;
  for (i=0l; i<32l; ++i)
    jump(i);
}

inline void TRNG::RAND::reset(void) {
  a=69069l;
  b=1l;
  r=1l;
```

```
}

inline void TRNG::RAND::seed(long s=0l) {
  if (s<0l)
    warn("bad seed for TRNG::RAND::seed");
  r=s;
}

inline long TRNG::RAND::rand(void) {
  r=a*r+b;
  return(r/2l);
}

RNG *TRNG::RAND::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new RAND);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::RAND::split(long s, long n) {
  long i;
  unsigned long t1, t2;
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::RAND::split");
  if (s>1l) {
    t1=1ul;
    t2=0ul;
    for (i=0l; i<=n; ++i)
      rand();
    for (i=0l; i<s; ++i) {
      t2+=t1;
      t1*=a;
    }
    b*=t2;
    a=t1;
    backward();
  }
}

void TRNG::RAND::jump(long s) {
  long i;
  unsigned long t1, t2, t3;
  if (s<0l)
    warn("invalid arguments for TRNG::RAND::jump");
  else {
    t1=a;
    for (i=0l; i<s; ++i)
      t1*=t1;
    t2=1ul;
    t3=a;
    while (s>0l) {
      t2*=(1ul+t3);
      t3*=t3;
      --s;
    }
    r=r*t1+t2*b;
  }
}
```

```
void TRNG::RAND::save_status(vector<long> &s) {
  s.resize(4);
  s[0]=type;
  s[1]=a;  s[2]=b;  s[3]=r;
}

void TRNG::RAND::load_status(vector<long> &s) {
  if ((s.size()!=4) || (s[0]!=type))
    warn("TRNG::RAND::load_status wrong parameter");
  else {
    a=s[1];
    b=s[2];
    r=s[3];
  }
}

inline TRNG::RAND::RAND() {
  reset();
  max_val=2147483647l;
  max_val2=max_val/2l;
}
```

7.2.5 Lineare-Kongruenzen-Generator mit Modulus 2^{64}

Lineare-Kongruenzen-Generatoren sind sehr schnell, sollten aber wegen ihrer oben beschriebenen Defizite nur eingesetzt werden, wenn diese für die konkrete Anwendung wirklich unproblematisch sind und es besonders auf Geschwindigkeit ankommt, siehe Abschnitt 8. Die Klasse class LCG64 stellt einen solchen Generator mit der Iterationsvorschrift

$$r_i \equiv 1814546000247786 6997 \cdot r_{i-1} + 1 \mod 2^{64} \tag{7.22}$$

zur Verfügung. Die Periode beträgt 2^{64}. Da die Methode long LCG64::val(void) einen Rückgabewert vom Typ long hat, wird bei der Rückgabe r_i durch 2^{33} geteilt.

Listing 7.18: Klassendefinition für Zufallszahlengenerator als Lineare-Kongruenzen-Generator mit Modulus 2^{64}

```
class LCG64 : public RNG {
  // linear conguential random number generator
  // modulo 2^64
private:
  unsigned long long a;
  unsigned long long b;
  unsigned long long r;
  void backward(void);
public:
  static const RNG_type type=LCG64_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
```

```
    void jump(long);
    void save_status(std::vector<long> &);
    void load_status(std::vector<long> &);
    LCG64();
  };
```

Listing 7.19: Implementation Klasse für Zufallszahlengenerator als Lineare-Kongruenzen-Generator mit Modulus 2^{64}

```
void TRNG::LCG64::backward(void) {
  long i;
  for (i=0l; i<64l; ++i)
    jump(i);
}

inline const char * TRNG::LCG64::name(void) {
  return("LCG64");
}

inline void TRNG::LCG64::reset(void) {
  a=18145460002477866997ull;
  b=1ull;
  r=1ull;
}

inline void TRNG::LCG64::seed(long s=0l) {
  r=(static_cast<unsigned long long>(s)<<32)+1ull;
}

inline long TRNG::LCG64::rand(void) {
  r=a*r+b;
  return(static_cast<long>(r>>33));
}

RNG *TRNG::LCG64::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new LCG64);
  save_status(s);
  R->load_status(s);
  return(R);
}

inline void TRNG::LCG64::split(long s, long n) {
  long i;
  unsigned long long t1, t2;
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::LCG64::split");
  if (s>1l) {
    t1=1ull;
    t2=0ull;
    for (i=0l; i<=n; ++i)
      rand();
    for (i=0l; i<s; ++i) {
      t2+=t1;
      t1*=a;
    }
    b*=t2;
    a=t1;
    backward();
  }
```

```
}

inline void TRNG::LCG64::jump(long s) {
  long i;
  unsigned long long t1, t2, t3;
  if (s<0l)
    warn("invalid arguments for TRNG::LCG64::jump");
  else {
    t1=a;
    for (i=0l; i<s; ++i)
      t1*=t1;
    t2=1ull;
    t3=a;
    while (s>0l) {
      t2*=(1ull+t3);
      t3*=t3;
      --s;
    }
    r=r*t1+t2*b;
  }
}

void TRNG::LCG64::save_status(vector<long> &s) {
  s.resize(7);
  s[0]=type;
  s[1]=static_cast<long>(a>>32);
  s[2]=static_cast<long>(a&0xffffffffull);
  s[3]=static_cast<long>(b>>32);
  s[4]=static_cast<long>(b&0xffffffffull);
  s[5]=static_cast<long>(r>>32);
  s[6]=static_cast<long>(r&0xffffffffull);
}

void TRNG::LCG64::load_status(vector<long> &s) {
  if ((s.size()!=7) || (s[0]!=type))
    warn("TRNG::LCG64::load_status wrong parameter");
  else {
    a=(static_cast<unsigned long long>(s[1])<<32)|
      (static_cast<unsigned long long>
       (static_cast<unsigned long>(s[2])));
    b=(static_cast<unsigned long long>(s[3])<<32)|
      (static_cast<unsigned long long>
       (static_cast<unsigned long>(s[4])));
    r=(static_cast<unsigned long long>(s[5])<<32)|
      (static_cast<unsigned long long>
       (static_cast<unsigned long>(s[6])));
  }
}

TRNG::LCG64::LCG64() {
  reset();
  max_val=0x7fffffffl;
  max_val2=max_val/2l;
}
```

In der Methode void LCG64::jump(long s) muss eine Summe der Form

$$1 + a^1 + a^2 + \cdots + a^{2^s-1} \tag{7.23}$$

berechnet werden, siehe Formel (4.3). Auf den ersten Blick sieht es so aus, als müsse man eine Summe mit 2^s Summanden berechnen. Man kann diese Summe aber vorteilhaft umformen:

$$\begin{aligned} & 1 + a^1 + a^2 + \cdots + a^{2^s-1} \\ &= (1 + a^2 + a^4 + \cdots + a^{2^s-2}) + (a^1 + a^3 + \cdots + a^{2^s-1}) \\ &= (1 + a^2 + a^4 + \cdots + a^{2^s-2}) + a(1 + a^2 + a^4 + \cdots + a^{2^s-2}) \\ &= (1 + a)(1 + a^2 + a^4 + \cdots + a^{2^s-2}) \\ &= (1 + a)(1 + (a^2) + (a^2)^2 + \cdots + (a^2)^{2^{(s-1)}-1}) . \end{aligned} \tag{7.24}$$

Wendet man nun diese Umformung auf den zweiten Faktor immer wieder an, so erhält man schließlich:

$$1 + a^1 + a^2 + \cdots + a^{2^s-1} = (1 + a)(1 + a^2)(1 + a^4) \ldots (1 + a^{2^{s-1}})) . \tag{7.25}$$

Aus einer Summe mit 2^s Summanden wurde ein Produkt aus s Faktoren.

Die Klasse class LCG64 besitzt eine Methode void LCG64::backward(void), die den Generator um einen Schritt zurückstellt. Da der Generator eine Periode von 2^{64} hat, entspricht dies einem Vorstellen um $2^{64} - 1 = \sum_{i=0}^{63} 2^i$ Schritte.

7.2.6 Verallgemeinerte lineare Kongruenzen

Die Klasse class LMC2 stellt eine Zufallszahlengenerator in Form der verallgemeinerten linearen Kongruenzen mit

$$r_i \equiv 523007613 \cdot r_{i-1} + 756894486 \cdot r_{i-2} \mod (2^{31} - 1) \tag{7.26}$$

dar. Die Periode beträgt $(2^{31} - 1)^2 - 1 \approx 2^{62} \approx 4{,}6 \cdot 10^{18}$.

Listing 7.20: Klassendefinition für Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit zwei Faktoren

```
class LMC2 : public RNG {
  // multiplicative linear conguential random number generator
  // with 2 multipliers
private:
  std::vector<long> a;
  std::vector<long> r;
  long modulus;
  void backward(void);
public:
  static const RNG_type type=LMC2_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
```

```
    void jump(long);
    void save_status(std::vector<long> &);
    void load_status(std::vector<long> &);
    LMC2();
    virtual ~LMC2() {}
  };
```

Listing 7.21: Implementation einer Klasse für eine Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit zwei Faktoren

```
inline const char * TRNG::LMC2::name(void) {
  return("LMC2");
}

void TRNG::LMC2::backward(void) {
  long t;
  t=r[0];
  t-=static_cast<long>((static_cast<long long>(a[0])*
                        static_cast<long long>(r[1]))%modulus);
  if (t<0l)
    t+=modulus;
  t=static_cast<long>((static_cast<long long>(t)*
                       static_cast<long long>
                       (modulo_invers(a[1], modulus)))%modulus);
  r[0]=r[1];  r[1]=t;
}

inline void TRNG::LMC2::reset(void) {
  vector<long> s(6);
  s[0]=type;
  s[1]=523007613l;  s[2]=756894486l;  // multiplier
  s[3]=0l;          s[4]=1l;          // seed
  s[5]=2147483647l;                   // modulus
  load_status(s);
}

inline void TRNG::LMC2::seed(long s=0l) {
  if (s<0l)
    warn("invalid argument for LMC2::seed");
  else {
    r[0]=s;  r[1]=1l;
  }
}

inline long TRNG::LMC2::rand(void) {
  long t;
  t=static_cast<long>
    (((static_cast<long long>(a[0])*
       static_cast<long long>(r[0]))%modulus+
      (static_cast<long long>(a[1])*
       static_cast<long long>(r[1]))%modulus)
     %modulus);
  r[1]=r[0];  r[0]=t;
  return(r[0]);
}

RNG *TRNG::LMC2::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new LMC2);
  save_status(s);
```

```
  R->load_status(s);
  return(R);
}

void TRNG::LMC2::split(long s, long n) {
  long i, q0, q1, q2, q3;
  vector<long> b(4);
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::LMC2::split");
  else
    if (s>1l) {
      for (i=0l; i<=n; ++i)
        q0=rand();
      for (i=0l; i<s; ++i)
        q1=rand();
      for (i=0l; i<s; ++i)
        q2=rand();
      for (i=0l; i<s; ++i)
        q3=rand();
      b[0]=q1;  b[1]=q0;
      b[2]=q2;  b[3]=q1;
      a[0]=q2;  a[1]=q3;
      gauss(b, a, modulus);
      r[0]=q1;  r[1]=q0;
      backward();
      backward();
    }
}

void TRNG::LMC2::jump(long s) {
  long i, t1, t2;
  vector<long> b(4), c(4), d(2);
  if (s<0l)
    warn("invalid arguments for TRNG::LMC2::jump");
  else {
    t1=a[0];    t2=a[1];
    b[0]=a[0];  b[1]=a[1];
    b[2]=1l;    b[3]=0l;
    for (i=0l; i<s; ++i)
      if ((i&1l)==0l)
        matrix_mult(b, b, c, modulus);
      else
        matrix_mult(c, c, b, modulus);
    if ((s&1l)==0l)
      matrix_vec_mult(b, r, d, modulus);
    else
      matrix_vec_mult(c, r, d, modulus);
    r[0]=d[0];  r[1]=d[1];
    a[0]=t1;    a[1]=t2;
  }
}

void TRNG::LMC2::save_status(vector<long> &s) {
  s.resize(6);
  s[0]=type;
  s[1]=a[0];  s[2]=a[1];
  s[3]=r[0];  s[4]=r[1];
  s[5]=modulus;
}

void TRNG::LMC2::load_status(vector<long> &s) {
```

```
  if ((s.size()!=6) || (s[0]!=type))
    warn("TRNG::LMC2::load_status_wrong_parameter");
  else {
    a[0]=s[1];  a[1]=s[2];
    r[0]=s[3];  r[1]=s[4];
    modulus=s[5];
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::LMC2::LMC2() : a(2), r(2) {
  reset();
}
```

Analog zu class LMC2 sind die Klassen class LMC3 und class LMC4 aufgebaut. Bei class LMC3 lautet die Iterationsvorschrift

$$r_i \equiv 523007613 \cdot r_{i-1} + 756894484 \cdot r_{i-2} + 935294381 \cdot r_{i-3} \mod (2^{31}-1) \tag{7.27}$$

und bei class LMC4

$$r_i \equiv 523007613 \cdot r_{i-1} + 210014925 \cdot r_{i-2} + 1033683247 \cdot r_{i-3} + 935294388 \cdot r_{i-4} \mod (2^{31}-1)\,. \tag{7.28}$$

Die Periode des Generators class LMC3 beträgt $(2^{31}-1)^3 - 1 \approx 2^{93} \approx 9{,}9 \cdot 10^{27}$, die von class LMC3 beträgt $(2^{31}-1)^4 - 1 \approx 2^{124} \approx 2{,}1 \cdot 10^{37}$.

Listing 7.22: Klassendefinition für Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit drei Faktoren

```
class LMC3 : public RNG {
  // multiplicative linear conguential random number generator
  // with 3 multipliers
private:
  std::vector<long> a;
  std::vector<long> r;
  long modulus;
  void backward(void);
public:
  static const RNG_type type=LMC3_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  LMC3();
  virtual ~LMC3() {}
};
```

Listing 7.23: Implementation einer Klasse für eine Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit drei Faktoren

```
inline const char * TRNG::LMC3::name(void) {
  return("LMC3");
}

void TRNG::LMC3::backward(void) {
  long t;
  t=r[0];
  t-=static_cast<long>((static_cast<long long>(a[0])*
                        static_cast<long long>(r[1]))%modulus);
  if (t<0l)
    t+=modulus;
  t-=static_cast<long>((static_cast<long long>(a[1])*
                        static_cast<long long>(r[2]))%modulus);
  if (t<0l)
    t+=modulus;
  t=static_cast<long>((static_cast<long long>(t)*
                       static_cast<long long>
                       (modulo_invers(a[2], modulus)))%modulus);
  r[0]=r[1];  r[1]=r[2];  r[2]=t;
}

inline void TRNG::LMC3::reset(void) {
  vector<long> s(8);
  s[0]=type;
  s[1]=523007613l;  s[2]=756894484l;  // multiplier
  s[3]=935294381l;
  s[4]=0l;          s[5]=1l;          // seed
  s[6]=1l;
  s[7]=2147483647l;                   // modulus
  load_status(s);
}

inline void TRNG::LMC3::seed(long s=0l) {
  if (s<0l)
    warn("invalid argument for LMC3::seed");
  else {
    r[0]=s;  r[1]=1l;  r[2]=1l;
  }
}

inline long TRNG::LMC3::rand(void) {
  long t;
  t=static_cast<long>
    (((static_cast<long long>(a[0])*
       static_cast<long long>(r[0]))%modulus+
      (static_cast<long long>(a[1])*
       static_cast<long long>(r[1]))%modulus+
      (static_cast<long long>(a[2])*
       static_cast<long long>(r[2]))%modulus)
     %modulus);
  r[2]=r[1];  r[1]=r[0];  r[0]=t;
  return(r[0]);
}

RNG *TRNG::LMC3::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new LMC3);
  save_status(s);
```

```
  R->load_status(s);
  return(R);
}

void TRNG::LMC3::split(long s, long n) {
  long i, q0, q1, q2, q3, q4, q5;
  vector<long> b(9);
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::LMC3::split");
  else
    if (s>1l) {
      for (i=0l; i<=n; ++i)
        q0=rand();
      for (i=0l; i<s; ++i)
        q1=rand();
      for (i=0l; i<s; ++i)
        q2=rand();
      for (i=0l; i<s; ++i)
        q3=rand();
      for (i=0l; i<s; ++i)
        q4=rand();
      for (i=0l; i<s; ++i)
        q5=rand();
      b[0]=q2;  b[1]=q1;  b[2]=q0;
      b[3]=q3;  b[4]=q2;  b[5]=q1;
      b[6]=q4;  b[7]=q3;  b[8]=q2;
      a[0]=q3;  a[1]=q4;  a[2]=q5;
      gauss(b, a, modulus);
      r[0]=q2;  r[1]=q1;  r[2]=q0;
      backward();
      backward();
      backward();
    }
}

void TRNG::LMC3::jump(long s) {
  long i, t1, t2, t3;
  vector<long> b(9), c(9), d(3);
  if (s<0l)
    warn("invalid arguments for TRNG::LMC3::jump");
  else {
    t1=a[0];    t2=a[1];    t3=a[2];
    b[0]=a[0];  b[1]=a[1];  b[2]=a[2];
    b[3]=1l;    b[4]=0l;    b[5]=0l;
    b[6]=0l;    b[7]=1l;    b[8]=0l;
    for (i=0l; i<s; ++i)
      if ((i&1l)==0l)
        matrix_mult(b, b, c, modulus);
      else
        matrix_mult(c, c, b, modulus);
    if ((s&1l)==0l)
      matrix_vec_mult(b, r, d, modulus);
    else
      matrix_vec_mult(c, r, d, modulus);
    r[0]=d[0];  r[1]=d[1];  r[2]=d[2];
    a[0]=t1;    a[1]=t2;    a[2]=t3;
  }
}

void TRNG::LMC3::save_status(vector<long> &s) {
  s.resize(8);
```

```
  s[0]=type;
  s[1]=a[0];  s[2]=a[1];  s[3]=a[2];
  s[4]=r[0];  s[5]=r[1];  s[6]=r[2];
  s[7]=modulus;
}

void TRNG::LMC3::load_status(vector<long> &s) {
  if ((s.size()!=8) || (s[0]!=type))
    warn("TRNG::LMC3::load_status wrong parameter");
  else {
    a[0]=s[1];  a[1]=s[2];  a[2]=s[3];
    r[0]=s[4];  r[1]=s[5];  r[2]=s[6];
    modulus=s[7];
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::LMC3::LMC3() : a(3), r(3) {
  reset();
}
```

Listing 7.24: Klassendefinition für Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit vier Faktoren

```
  class LMC4 : public RNG {
    // multiplicative linear conguential random number generator
    // with 4 multipliers
  private:
    std::vector<long> a;
    std::vector<long> r;
    long modulus;
    void backward(void);
  public:
    static const RNG_type type=LMC4_t;
    const char * name(void);
    void reset(void);
    void seed(long);
    long rand(void);
    RNG *copy(void);
    void split(long, long);
    void jump(long);
    void save_status(std::vector<long> &);
    void load_status(std::vector<long> &);
    LMC4();
    virtual ~LMC4() {}
  };
```

Listing 7.25: Implementation einer Klasse für eine Zufallszahlengenerator mit verallgemeinerten linearen Kongruenzen mit vier Faktoren

```
inline const char * TRNG::LMC4::name(void) {
  return("LMC4");
}

void TRNG::LMC4::backward(void) {
  long t;
  t=r[0];
```

```
  t-=static_cast<long>((static_cast<long long>(a[0])*
                        static_cast<long long>(r[1]))%modulus);
  if (t<0l)
    t+=modulus;
  t-=static_cast<long>((static_cast<long long>(a[1])*
                        static_cast<long long>(r[2]))%modulus);
  if (t<0l)
    t+=modulus;
  t-=static_cast<long>((static_cast<long long>(a[2])*
                        static_cast<long long>(r[3]))%modulus);
  if (t<0l)
    t+=modulus;
  t=static_cast<long>((static_cast<long long>(t)*
                       static_cast<long long>
                       (modulo_invers(a[3], modulus)))%modulus);
  r[0]=r[1];  r[1]=r[2];  r[2]=r[3];  r[3]=t;
}

inline void TRNG::LMC4::reset(void) {
  vector<long> s(10);
  s[0]=type;
  s[1]=523007613l;  s[2]=210014925l;  // multiplier
  s[3]=1033683247l; s[4]=935294388l;
  s[5]=0l;          s[6]=1l;          // seed
  s[7]=1l;          s[8]=1l;
  s[9]=2147483647l;                   // modulus
  load_status(s);
}

inline void TRNG::LMC4::seed(long s=0l) {
  if (s<0l)
    warn("invalid argument for LMC4::seed");
  else {
    r[0]=s;  r[1]=1l;  r[2]=1l;  r[3]=1l;
  }
}

inline long TRNG::LMC4::rand(void) {
  long t;
  t=static_cast<long>
    (((static_cast<long long>(a[0])*
       static_cast<long long>(r[0]))%modulus+
      (static_cast<long long>(a[1])*
       static_cast<long long>(r[1]))%modulus+
      (static_cast<long long>(a[2])*
       static_cast<long long>(r[2]))%modulus+
      (static_cast<long long>(a[3])*
       static_cast<long long>(r[3]))%modulus)
     %modulus);
  r[3]=r[2];  r[2]=r[1];  r[1]=r[0];  r[0]=t;
  return(r[0]);
}

RNG *TRNG::LMC4::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new LMC4);
  save_status(s);
  R->load_status(s);
  return(R);
}
```

```
void TRNG::LMC4::split(long s, long n) {
  long i, q0, q1, q2, q3, q4, q5, q6, q7;
  vector<long> b(16);
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::LMC4::split");
  else
    if (s>1l) {
      for (i=0l; i<=n; ++i)
        q0=rand();
      for (i=0l; i<s; ++i)
        q1=rand();
      for (i=0l; i<s; ++i)
        q2=rand();
      for (i=0l; i<s; ++i)
        q3=rand();
      for (i=0l; i<s; ++i)
        q4=rand();
      for (i=0l; i<s; ++i)
        q5=rand();
      for (i=0l; i<s; ++i)
        q6=rand();
      for (i=0l; i<s; ++i)
        q7=rand();
      b[ 0]=q3;  b[ 1]=q2;  b[ 2]=q1;  b[ 3]=q0;
      b[ 4]=q4;  b[ 5]=q3;  b[ 6]=q2;  b[ 7]=q1;
      b[ 8]=q5;  b[ 9]=q4;  b[10]=q3;  b[11]=q2;
      b[12]=q6;  b[13]=q5;  b[14]=q4;  b[15]=q3;
      a[ 0]=q4;  a[ 1]=q5;  a[ 2]=q6;  a[ 3]=q7;
      gauss(b, a, modulus);
      r[0]=q3;   r[1]=q2;   r[2]=q1;   r[3]=q0;
      backward();
      backward();
      backward();
      backward();
    }
}

void TRNG::LMC4::jump(long s) {
  long i, t1, t2, t3, t4;
  vector<long> b(16), c(16), d(4);
  if (s<0l)
    warn("invalid arguments for TRNG::LMC4::jump");
  else {
    t1=a[0];     t2=a[1];     t3=a[2];     t4=a[3];
    b[ 0]=a[0];  b[ 1]=a[1];  b[ 2]=a[2];  b[ 3]=a[3];
    b[ 4]=1l;    b[ 5]=0l;    b[ 6]=0l;    b[ 7]=0l;
    b[ 8]=0l;    b[ 9]=1l;    b[10]=0l;    b[11]=0l;
    b[12]=0l;    b[13]=0l;    b[14]=1l;    b[15]=0l;
    for (i=0l; i<s; ++i)
      if ((i&1l)==0l)
        matrix_mult(b, b, c, modulus);
      else
        matrix_mult(c, c, b, modulus);
    if ((s&1l)==0l)
      matrix_vec_mult(b, r, d, modulus);
    else
      matrix_vec_mult(c, r, d, modulus);
    r[0]=d[0];  r[1]=d[1];  r[2]=d[2];  r[3]=d[3];
    a[0]=t1;    a[1]=t2;    a[2]=t3;    a[3]=t4;
  }
}
```

```
void TRNG::LMC4::save_status(vector<long> &s) {
  s.resize(10);
  s[0]=type;
  s[1]=a[0];  s[2]=a[1];  s[3]=a[2];  s[4]=a[3];
  s[5]=r[0];  s[6]=r[1];  s[7]=r[2];  s[8]=r[3];
  s[9]=modulus;
}

void TRNG::LMC4::load_status(vector<long> &s) {
  if ((s.size()!=10) || (s[0]!=type))
    warn("TRNG::LMC4::load_status wrong parameter");
  else {
    a[0]=s[1];  a[1]=s[2];  a[2]=s[3];  a[3]=s[4];
    r[0]=s[5];  r[1]=s[6];  r[2]=s[7];  r[3]=s[8];
    modulus=s[9];
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::LMC4::LMC4() : a(4), r(4) {
  reset();
}
```

7.2.7 Kombinierte Generatoren

Mit den Klassen class CLCG2, class CLCG3 und class CLCG4 wurden drei kombinierte Generatoren implementiert, wie sie in Abschnitt 3.7 beschrieben wurden. die Iterationsvorschrift für class CLCG2 lautet

$$\begin{aligned} r_{1,i} &\equiv 376555083 \cdot r_{1,i-1} \mod 2147482951 \\ r_{2,i} &\equiv 1028879659 \cdot r_{2,i-1} \mod 2147482949 \\ r_i &\equiv \sum_{j=1}^{2} r_{j,i} \mod (2147482951 - 1), \end{aligned} \tag{7.29}$$

die für class CLCG3

$$\begin{aligned} r_{1,i} &\equiv 376555083 \cdot r_{1,i-1} \mod 2147482951 \\ r_{2,i} &\equiv 1028879659 \cdot r_{2,i-1} \mod 2147482949 \\ r_{3,i} &\equiv 225802979 \cdot r_{3,i-1} \mod 2147482943 \\ r_i &\equiv \sum_{j=1}^{3} r_{j,i} \mod (2147482951 - 1) \end{aligned} \tag{7.30}$$

und die für class CLCG4

$$
\begin{aligned}
r_{1,i} &\equiv 376555083 \cdot r_{1,i-1} \mod 2147482951 \\
r_{2,i} &\equiv 1028879659 \cdot r_{2,i-1} \mod 2147482949 \\
r_{3,i} &\equiv 225802979 \cdot r_{3,i-1} \mod 2147482943 \\
r_{4,i} &\equiv 2028073966 \cdot r_{2,i-1} \mod 2147482859 \\
r_i &\equiv \sum_{j=1}^{4} r_{j,i} \mod (2147482951 - 1) .
\end{aligned}
\tag{7.31}
$$

Diese Generatoren erreichen die für Generatoren diesen Typs maximale Periode von ca. $2^{61} \approx 2{,}3 \cdot 10^{18}$, $2^{91} \approx 2{,}5 \cdot 10^{27}$ bzw. ca. $2^{121} \approx 2{,}7 \cdot 10^{36}$.

Listing 7.26: Definition der Klasse für aus zwei rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
class CLCG2 : public RNG {
  // combined linear conguential random number generator
  // with two generators
private:
  generic_MLCG R1;
  generic_MLCG R2;
  static const long a1= 376555083l, m1=2147482951l;
  static const long a2=1028879659l, m2=2147482949l;
  long modulus;
public:
  static const RNG_type type=CLCG2_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  CLCG2();
};
```

Listing 7.27: Implementation Klasse für aus zwei rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
inline const char * TRNG::CLCG2::name(void) {
  return("CLCG2");
}

inline void TRNG::CLCG2::reset(void) {
  R1.reset();
  R2.reset();
  modulus=R1.max();
  max_val=modulus-1l;
  max_val2=max_val/2l;
}

inline void TRNG::CLCG2::seed(long s=0l) {
```

```
  if (s<0l)
    warn("bad_seed_for_TRNG::CLCG2::seed");
  R1.seed(s);
  R2.seed(s);
}

inline long TRNG::CLCG2::rand(void) {
  return(static_cast<long>((static_cast<long long>(R1.rand())+
                            static_cast<long long>(R2.rand()))%
                           modulus));
}

RNG *TRNG::CLCG2::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new CLCG2);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::CLCG2::split(long s, long n) {
  R1.split(s, n);
  R2.split(s, n);
}

void TRNG::CLCG2::jump(long s) {
  if (s<0l)
    warn("invalid_arguments_for_TRNG::CLCG2::jump");
  else {
    R1.jump(s);
    R2.jump(s);
  }
}

void TRNG::CLCG2::save_status(vector<long> &s) {
  vector<long> t;
  s.resize(7);
  R1.save_status(t);
  s[0]=type;
  s[1]=t[1];  s[2]=t[2];  s[3]=t[3];
  R2.save_status(t);
  s[4]=t[1];  s[5]=t[2];  s[6]=t[3];
}

void TRNG::CLCG2::load_status(vector<long> &s) {
  vector<long> t(4);
  if ((s.size()!=7) || (s[0]!=type))
    warn("TRNG::CLCG2::load_status_wrong_parameter");
  else {
    t[0]=generic_MLCG_t;
    t[1]=s[1];  t[2]=s[2];  t[3]=s[3];
    R1.load_status(t);
    t[1]=s[4];  t[2]=s[5];  t[3]=s[6];
    R2.load_status(t);
    modulus=R1.max();
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::CLCG2::CLCG2() :
```

```
  R1(a1, m1, 1l), R2(a2, m2, 1l) {
  reset();
}
```

Listing 7.28: Definition der Klasse für aus drei rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
class CLCG3 : public RNG {
  // combined linear conguential random number generator
  // with three generators
private:
  generic_MLCG R1;
  generic_MLCG R2;
  generic_MLCG R3;
  static const long a1= 376555083l, m1=2147482951l;
  static const long a2=1028879659l, m2=2147482949l;
  static const long a3= 225802979l, m3=2147482943l;
  long modulus;
public:
  static const RNG_type type=CLCG3_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  CLCG3();
};
```

Listing 7.29: Implementation Klasse für aus drei rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
inline const char * TRNG::CLCG3::name(void) {
  return("CLCG3");
}

inline void TRNG::CLCG3::reset(void) {
  R1.reset();
  R2.reset();
  R3.reset();
  modulus=R1.max();
  max_val=modulus-1l;
  max_val2=max_val/2l;
}

inline void TRNG::CLCG3::seed(long s=0l) {
  if (s<0l)
    warn("bad_seed_for_TRNG::CLCG3::seed");
  R1.seed(s);
  R2.seed(s);
  R3.seed(s);
}

inline long TRNG::CLCG3::rand(void) {
  return(static_cast<long>((static_cast<long long>(R1.rand())+
```

```
                                static_cast<long long>(R2.rand())+
                                static_cast<long long>(R3.rand()))%
                               modulus));
}

RNG *TRNG::CLCG3::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new CLCG3);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::CLCG3::split(long s, long n) {
  R1.split(s, n);
  R2.split(s, n);
  R3.split(s, n);
}

void TRNG::CLCG3::jump(long s) {
  if (s<0l)
    warn("invalid arguments for TRNG::CLCG3::jump");
  else {
    R1.jump(s);
    R2.jump(s);
    R3.jump(s);
  }
}

void TRNG::CLCG3::save_status(vector<long> &s) {
  vector<long> t;
  s.resize(10);
  R1.save_status(t);
  s[0]=type;
  s[1]=t[1];  s[2]=t[2]; s[3]=t[3];
  R2.save_status(t);
  s[4]=t[1];  s[5]=t[2]; s[6]=t[3];
  R3.save_status(t);
  s[7]=t[1];  s[8]=t[2]; s[9]=t[3];
}

void TRNG::CLCG3::load_status(vector<long> &s) {
  vector<long> t(4);
  if ((s.size()!=10) || (s[0]!=type))
    warn("TRNG::CLCG3::load_status wrong parameter");
  else {
    t[0]=generic_MLCG_t;
    t[1]=s[1];  t[2]=s[2];  t[3]=s[3];
    R1.load_status(t);
    t[1]=s[4];  t[2]=s[5];  t[3]=s[6];
    R2.load_status(t);
    t[1]=s[7];  t[2]=s[8];  t[3]=s[9];
    R3.load_status(t);
    modulus=R1.max();
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::CLCG3::CLCG3() :
  R1(a1, m1, 1l), R2(a2, m2, 1l), R3(a3, m3, 1l) {
```

```
  reset();
}
```

Listing 7.30: Definition der Klasse für aus vier rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
class CLCG4 : public RNG {
  // combined linear conguential random number generator
  // with four generators
private:
  generic_MLCG R1;
  generic_MLCG R2;
  generic_MLCG R3;
  generic_MLCG R4;
  static const long a1= 376555083l, m1=2147482951l;
  static const long a2=1028879659l, m2=2147482949l;
  static const long a3= 225802979l, m3=2147482943l;
  static const long a4=2028073966l, m4=2147482859l;
  long modulus;
public:
  static const RNG_type type=CLCG4_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  CLCG4();
};
```

Listing 7.31: Implementation Klasse für aus vier rein multiplikativen Generatoren kombinierten Zufallszahlengenerator

```
inline const char * TRNG::CLCG4::name(void) {
  return("CLCG4");
}

inline void TRNG::CLCG4::reset(void) {
  R1.reset();
  R2.reset();
  R3.reset();
  R4.reset();
  modulus=R1.max();
  max_val=modulus-1l;
  max_val2=max_val/2l;
}

inline void TRNG::CLCG4::seed(long s=0l) {
  if (s<0l)
    warn("bad_seed_for_TRNG::CLCG4::seed");
  R1.seed(s);
  R2.seed(s);
  R3.seed(s);
  R4.seed(s);
}
```

```
inline long TRNG::CLCG4::rand(void) {
  return(static_cast<long>((static_cast<long long>(R1.rand())+
                            static_cast<long long>(R2.rand())+
                            static_cast<long long>(R3.rand())+
                            static_cast<long long>(R4.rand()))%
                           modulus));
}

RNG *TRNG::CLCG4::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new CLCG4);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::CLCG4::split(long s, long n) {
  R1.split(s, n);
  R2.split(s, n);
  R3.split(s, n);
  R4.split(s, n);
}

void TRNG::CLCG4::jump(long s) {
  if (s<0l)
    warn("invalid arguments for TRNG::CLCG4::jump");
  else {
    R1.jump(s);
    R2.jump(s);
    R3.jump(s);
    R4.jump(s);
  }
}

void TRNG::CLCG4::save_status(vector<long> &s) {
  vector<long> t;
  s.resize(13);
  R1.save_status(t);
  s[0]=type;
  s[1]=t[1];  s[2]=t[2];  s[3]=t[3];
  R2.save_status(t);
  s[4]=t[1];  s[5]=t[2];  s[6]=t[3];
  R3.save_status(t);
  s[7]=t[1];  s[8]=t[2];  s[9]=t[3];
  R4.save_status(t);
  s[10]=t[1];  s[11]=t[2];  s[12]=t[3];
}

void TRNG::CLCG4::load_status(vector<long> &s) {
  vector<long> t(4);
  if ((s.size()!=13) || (s[0]!=type))
    warn("TRNG::CLCG4::load_status wrong parameter");
  else {
    t[0]=generic_MLCG_t;
    t[1]=s[1];  t[2]=s[2];  t[3]=s[3];
    R1.load_status(t);
    t[1]=s[4];  t[2]=s[5];  t[3]=s[6];
    R2.load_status(t);
    t[1]=s[7];  t[2]=s[8];  t[3]=s[9];
    R3.load_status(t);
```

```
    t[1]=s[10];  t[2]=s[11];  t[3]=s[12];
    R4.load_status(t);
    modulus=R1.max();
    max_val=modulus-1l;
    max_val2=max_val/2l;
  }
}

inline TRNG::CLCG4::CLCG4() :
  R1(a1, m1, 1l), R2(a2, m2, 1l), R3(a3, m3, 1l), R4(a4, m4, 1l) {
  reset();
}
```

7.2.8 Explizit inverser Generator

Der hier implementierte explizit inverse Generator arbeitet noch der Iterationsvorschrift

$$r_i = \overline{1073741831i} \mod (2^{30} + 2^{28} + 3). \tag{7.32}$$

Das Erzeugen der Teilfolgen ist trivial. Aus Geschwindigkeitsgründen besitzt die Klasse eine eigene speziell auf den primen Modulus $2^{30} + 2^{28} + 3$ abgestimmte Methode zur Berechnung des Inversen. Trotzdem ist der Generator noch immer vergleichsweise langsam. Gegenüber ParkMiller benötigt er ca. 30-mal mehr Zeit, um eine Pseudozufallszahl zu erzeugen. In Abschnitt 8.6 wird das Laufzeitverhalten der Generatoren näher untersucht. Seine Periode beträgt $2^{30} + 2^{28} + 3$.

Listing 7.32: Definition der Klasse für einen explizit inversen Generator

```
class EINV : public RNG {
  // explicit inversive congruential generator
  // Eichenauer-Hermann, J. "Statistical independence of a new class
  // of inversive congruential pseudorandom numbers"
  // Math. Comp. 60:375-384, 1993
private:
  long a, i, di, modulus;
  long modulo_invers(long);
public:
  static const RNG_type type=EINV_t;
  const char * name(void);
  void reset(void);
  void seed(long);
  long rand(void);
  RNG *copy(void);
  void split(long, long);
  void jump(long);
  void save_status(std::vector<long> &);
  void load_status(std::vector<long> &);
  EINV();
};
```

Listing 7.33: Implementation Klasse für einen explizit inversen Generator

```
inline const char * TRNG::EINV::name(void) {
  return("EINV");
}

inline long TRNG::EINV::modulo_invers(long a) {
  long long t1, t2, t3;
  // special modulus invers function for 2^30+2^28+3
  t1=t2=a;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;  t2=(t2*t2)%modulus;
  t2=(t2*t2)%modulus;
  t3=t2;
  t3=(t3*t3)%modulus;  t3=(t3*t3)%modulus;
  return(static_cast<long>((((t1*t2)%modulus)*t3)%modulus));
}

inline void TRNG::EINV::reset(void) {
  i=0l;
  di=1l;
  a=1073741831l;
  modulus=1342177283l;  // 2^30+2^28+3
  max_val=modulus-1l;
  max_val2=max_val/2l;
}

inline void TRNG::EINV::seed(long s=0l) {
  if ((s%modulus)<=0l)
    warn("bad seed for TRNG::EINV::seed");
  i=s%modulus;
}

inline long TRNG::EINV::rand(void) {
  i=static_cast<long>
    ((static_cast<long long>(i)+
      static_cast<long long>(di))%modulus);
  return(modulo_invers(static_cast<long>
                       ((static_cast<long long>(a)*
                         static_cast<long long>(i))%modulus)));
}

RNG *TRNG::EINV::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new EINV);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::EINV::split(long s, long n) {
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::EINV::split");
  if (s>1l) {
```

```
    di=static_cast<long>
      ((static_cast<long long>(s)*
        static_cast<long long>(di))%modulus);
    i=static_cast<long>
      ((static_cast<long long>(i)-
        static_cast<long long>(di)+
        static_cast<long long>(n)+1ll)%modulus);
  }
}

void TRNG::EINV::jump(long s) {
  long long t;
  long j;
  if (s<0l)
    warn("invalid arguments for TRNG::EINV::jump");
  else {
    t=1ll;
    for (j=0l; j<s; ++j)
      t=(t+t)%modulus;
    t=t*di%modulus;
    i=static_cast<long>((t+static_cast<long long>(i))%modulus);
  }
}

void TRNG::EINV::save_status(vector<long> &s) {
  s.resize(4);
  s[0]=type;
  s[1]=a;  s[2]=i;  s[3]=di;
}

void TRNG::EINV::load_status(vector<long> &s) {
  if ((s.size()!=4) || (s[0]!=type))
    warn("TRNG::EINV::load_status wrong parameter");
  else {
    a=s[1];  i=s[2];  di=s[3];
  }
}

inline TRNG::EINV::EINV() {
  reset();
}
```

7.2.9 Ein weiterer kombinierter Generator

Für der Generator der Klasse EINVLCG64 werden die der Klassen LCG64 und EINV einfach mod 2^{31} addiert. Dadurch erreicht man eine viel größere Periode als bei der Klasse EINV allein, nämlich $2^{64}(2^{30}+2^{28}+3)$. Dabei ist der Generator nur geringfügig langsamer als EINV.

Listing 7.34: Definition der Klasse für einen explizit inversen Generator

```
class EINVLCG64 : public RNG {
  // combined generator
  // explicit inversive congruential generator and
  // linear conguential random number generator modulo 2^64
private:
  LCG64 R1;
```

```
    EINV R2;
  public:
    static const RNG_type type=EINVLCG64_t;
    const char * name(void);
    void reset(void);
    void seed(long);
    long rand(void);
    RNG *copy(void);
    void split(long, long);
    void jump(long);
    void save_status(std::vector<long> &);
    void load_status(std::vector<long> &);
    EINVLCG64();
  };
```

Listing 7.35: Implementation Klasse für einen explizit inversen Generator

```
inline const char * TRNG::EINVLCG64::name(void) {
  return("EINVLCG64");
}

inline void TRNG::EINVLCG64::reset(void) {
  R1.reset();
  R2.reset();
  max_val=0x7fffffff;
  max_val2=max_val/2l;
}

inline void TRNG::EINVLCG64::seed(long s=0l) {
  R1.seed(s);
  R2.seed();
}

inline long TRNG::EINVLCG64::rand(void) {
  return((R1.rand()+R2.rand())&0x7fffffffl);
}

RNG *TRNG::EINVLCG64::copy(void) {
  vector<long> s;
  RNG *R=static_cast<RNG *>(new EINVLCG64);
  save_status(s);
  R->load_status(s);
  return(R);
}

void TRNG::EINVLCG64::split(long s, long n) {
  if (s<1l || n>s || n<0l)
    warn("invalid arguments for TRNG::EINVLCG64::split");
  if (s>1l) {
    R1.split(s, n);
    R2.split(s, n);
  }
}

void TRNG::EINVLCG64::jump(long s) {
  R1.jump(s);
  R2.jump(s);
}

void TRNG::EINVLCG64::save_status(vector<long> &s) {
```

```
  vector<long> t;
  s.resize(10);
  R1.save_status(t);
  s[0]=type;
  s[1]=t[1];  s[2]=t[2];  s[3]=t[3];
  s[4]=t[4];  s[5]=t[5];  s[6]=t[6];
  R2.save_status(t);
  s[7]=t[1];  s[8]=t[2];  s[9]=t[3];
}

void TRNG::EINVLCG64::load_status(vector<long> &s) {
  vector<long> t;
  if ((s.size()!=10) || (s[0]!=type))
    warn("TRNG::EINVLCG64::load_status_wrong_parameter");
  else {
    t.resize(7);
    t[0]=LCG64_t;
    t[1]=s[1];  t[2]=s[2];  t[3]=s[3];
    t[4]=s[4];  t[5]=s[5];  t[6]=s[6];
    R1.load_status(t);
    t.resize(4);
    t[0]=EINV_t;
    t[1]=s[7];  t[2]=s[8];  t[3]=s[9];
    R2.load_status(t);
  }
}

inline TRNG::EINVLCG64::EINVLCG64() {
  reset();
}
```

Die implementierten (Pseudo-) Zufallszahlengeneratoren im Test

> Indeed, a random number generator is much like sex: when its good its wonderful, and when its bad its still pretty good.
>
> GEORGE MARSAGLIA,
> ≫A current view of random number generators≪

8.1 Spektraltest

KNUTH beschreibt ausführlich in [Knuth 1981a] einen Algorithmus, um die Gitterstruktur von Generatoren der Form (3.4) zu untersuchen. Er berechnet den Anstand der Hyperflächen auf denen die Punkte (3.8) liegen. Er skaliert mit $O(3^n)$, wobei n hier für die Dimension des Raums, in dem die Hyperflächen liegen, steht.

Auf der dieser Arbeit beiliegenden CD-ROM findet man ein Programm, mit dem man erschöpfend nach guten (im Sinne des Spektraltests) Multiplikatoren suchen kann. Diese Implementation führt den Spektraltest bis in acht Dimensionen für m maximal $2^{31} - 1$ durch. Das Programm erwartet den Modulus und einen Suchbereich als Argumente. Diese Implementation benutzt fast ausschließlich Integerarithmetik. In [Hopkins 1983] findet man eine Variante in Fortran 66 in Fließkommaarithmetik. Für größere Moduli kann man z. B. die Implementation von [Karian und Goyal] für Maple verwenden.

Der Spektraltest sucht das Minimum

$$\nu_t(a,m) = \min \left\{ \begin{array}{l} \sqrt{x_1^2 + \ldots + x_t^2} \;\Big|\; \\ \\ \qquad x_1 + a x_2 + \ldots + a^{t-1} x_t \equiv 0 \mod m \end{array} \right\} . \tag{8.1}$$

Die ganzen Zahlen a und m sind teilerfremd und es gilt $0 < a < m$. In Abhängigkeit vom Modulus m und der Dimension t kann man obere Schranken für dieses Minimum angeben.

$$\nu_t(m)^* = \begin{cases} \left(\frac{4}{3}\right)^{\frac{1}{4}} m^{\frac{1}{2}} & \text{falls } t = 2 \\ 2^{\frac{1}{6}} m^{\frac{1}{3}} & \text{falls } t = 3 \\ 2^{\frac{1}{4}} m^{\frac{1}{4}} & \text{falls } t = 4 \\ 2^{\frac{3}{10}} m^{\frac{1}{5}} & \text{falls } t = 5 \\ \left(\frac{64}{3}\right)^{\frac{1}{12}} m^{\frac{1}{6}} & \text{falls } t = 6 \\ 2^{\frac{3}{7}} m^{\frac{1}{7}} & \text{falls } t = 7 \\ 2^{\frac{1}{2}} m^{\frac{1}{8}} & \text{falls } t = 8 \end{cases} \tag{8.2}$$

Die Ergebnisse des Spektraltests für verschiedene Moduli und Dimensionen kann man besser vergleichen, wenn man die Ergebnisse auf diese oberen Schranken normiert.

$$S_t = \frac{\nu_t(a,m)}{\nu_t(m)^*} \tag{8.3}$$

Ausführlich auf die Theorie des Spektraltests einzugehen, würde hier den Rahmen sprengen, daher soll hier nur auf [Knuth 1981a] und [Fishman 1997] verwiesen werden.

In Tabelle 8.1 sind die Ergebnisse des Spektraltests für die implementierten Zufallszahlengeneratoren nach der Methode der linearen Kongruenzen gelistet. Je näher die Werte bei eins sind, desto besser ist der Generator. Die letzte Spalte gibt den Mittelwert der Testergebnisse an. Der Multiplikator der Klasse LCG64 wurde nach einer mehrstündigen Zufallssuche gefunden.

Tabelle 8.1: Die implementierten Generatoren im Spektraltest

$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
class ParkMiller $r_i \equiv 16807 \cdot r_{i-1} \mod (2^{31}-1)$							
0,338	0,441	0,575	0,736	0,645	0,571	0,610	0,560
class RAND $r_i \equiv 69069 \cdot r_{i-1} + 1 \mod 2^{32}$							
0,925	0,789	0,755	0,804	0,299	0,407	0,576	0,651
class LCG64 $r_i \equiv 1814546000247786997 \cdot r_{i-1} + 1 \mod 2^{64}$							
0,933	0,614	0,741	0,638	0,734	0,633	0,615	0,701

Für die nach der Gleichung (3.28) kombinierten Generatoren schlägt [L'Ecuyer 1988] Einzelgeneratoren vor, die für sich jeweils gut im Spektraltest abschneiden. Für die Generatoren CLCG2, CLCG3 und CLCG4 wurde eine erschöpfende Suche nach guten Multiplikatoren durchgeführt. Die primen

Moduli 2 147 482 951, 2 147 482 949, 2 147 482 943 und 2 147 482 859 garantieren, dass die kombinierten Generatoren eine maximale Periode haben. Die Ergebnisse der erschöpfenden Suche für diese Moduli ist im Anhang zu finden. Die untere Schranke für S_t betrug 0,75.

8.2 Empirische Tests nach KNUTH

In den folgenden Tests wurden die implementierten Zufallszahlengeneratoren untersucht. Bei all diesen wurden mittels leapfrog-Methode 1 bis 256 Subfolgen erzeugt und jeweils die erste siebenmal dem Test unterzogen. Der Test gilt als nichtbestanden, wenn dreimal oder öfter für den χ^2-Test eine Wahrscheinlichkeit kleiner 1 % oder größer 99 % errechnet wurde. Alle nummerischen Ergebnisse hier aufzulisten, ist aus Platzgründen nicht möglich. Die Testergebnisse können aber von jederman nachvollzogen werden, die Programme befinden sich auf der beiliegenden CD-ROM.

8.2.1 Equidistribution test

Der Gleichverteilungstest wurde mit drei verschiedenen Parametern durchgeführt. Ordnet man 5000 zufällig gewählte Zahlen zwischen 0 und 1 100 Teilintervallen zu, so besteht jeder Generator den Test. Bei 1000 Teilintervallen und 50 000 gewählten Zahlen fällt der Generator RAND durch den Test, wenn man ihn in 92 Teilfolgen aufteilt, CLCG3 fällt bei 251 Teilfolgen durch. Mit 1000 Teilintervallen und 500 000 gewählten Zahlen kann der Generator RAND nicht bestehen, wenn man ihn in 256 Teilfolgen aufteilt.

Etwas empfindlicher ist das Programm bitequidisttest, dass die Gleichverteilung der einzelnen Bits testet. Die untersten acht bis zwölf Bits des Generators RAND wiederholen sich in sehr kurzen Zyklen und könne diesen Test nicht bestehen. Beim Generator EINV hingegen können die obersten drei Bits den Test nicht bestehen, kombiniert man jedoch diesen Generator mit LCG64 zu EINVLCG64, findet man kein solches pathologisches Verhalten mehr. Aufgrund des bei EINV verwendeten Modulus von $2^{30} + 2^{28} + 3$ können die obersten Bits aber auch gar nicht gleichverteilt sein.

Bei allen anderen Generatoren kann man einzelne Teilfolgen finden, bei denen einzelne Bits den Test nicht bestehen. Welche Teilfolgen und Bits dies betrifft, hängt aber sehr von den Parametern ab, mit denen man das Testprogramm bitequidisttest aufruft.

8.2.2 Serial test

Im serial test zeigt keiner der Pseudozufallszahlengeneratoren irgendwelche Auffälligkeiten. Dies gilt auch für deren Teilfolgen.

8.2.3 Gap test

Im gap test zeigt nur der Generator RAND ein schlechtes Verhalten. Spaltet man die Zufallszahlenfolge in 64, 128, 192 oder 256 Teilfolgen, so können diese den Test nicht bestehen, wenn man

```
user@random:~ > gaptest 7 256 0 0.5 12 1000000
```

aufruft.

8.2.4 Partition test

Auch hier fällt wieder nur der Generator RAND negativ auf. Beim partition test mit 10-Tupeln, deren Elemente 100 verschiedene Werte annehmen können, und 100 000 Tupeln je χ^2-Test kann der Generator RAND den Test nicht bestehen, wenn er in 96, 128, 188, 192 oder 256 Teilfolgen aufgespalten wird. Alle weiteren Generatoren bestehen den Test.

8.2.5 Coupon Collector's test

Wie auch schon beim partition test kann hier der Generator RAND den Test nicht bestehen. Diesmal wenn die ursprüngliche Folge in 64, 128, 192 oder 256 Teilfolgen gespalten wird. Die anderen Pseudozufallszahlengeneratoren sind wieder ohne Befund. Zehn Coupons sollten im Test gesucht werden.

```
user@random:~ > couponcollectorstest 7 256 40000 10 40
```

8.2.6 Permutation test

Ein weiteres mal kann der Generator RAND einen Test nicht bestehen. Den Permutation test kann diesen nicht bestehen, wenn seine ursprüngliche Folge in 32, 64, 96, 128, 135, 160, 192, 224 oder 256 Teilfolgen aufgespalten wird. Es wurde hier in jedem Test die relative Ordnung von 150 000 7-Tupeln untersucht. Die anderen Pseudozufallszahlengeneratoren bestehen auch diesen Test.

8.2.7 Run test

Beim run test wurden die Längen von je 100 000 aufsteigenden Teilfolgen analysiert. Für den χ^2-Test wurden Sequenzen länger als sieben zusammengefasst. Ein weiteres mal zeigt RAND Schwächen, wenn seine ursprüngliche Folge in 32, 64, 96, 128, 160, 192, 224 und 256 Teilfolgen aufgespalten wird. Der Generator LMC3 besteht den Test bei 246 Teilfolgen nicht. Die anderen Generatoren absolvieren jedoch den Test erfolgreich.

8.2.8 Maximum-of-t test

Bei fast jedem Generator finden sich Teilfolgen, die diesen Test nicht bestehen können. Welche Teilfolgen davon betroffen sind hängt stark von den Parametern ab, mit denen das Testprogramm aufgerufen wird. Eine Systematik lässt sich nicht erkennen. Die Ergebnisse eines Tests, der jeweils mit 10 000 20-Tupeln durchgeführt wurde, sind der Tabelle 8.2 zu entnehmen. Für den χ^2-Test wurden 200 Klassen verwendet.

Tabelle 8.2: Wenn man die Generatoren in n Teilfolgen spaltet, bestehen einige dieser Folgen den maximum-of-t test nicht.

Generator	Anzahl n der Teilfolgen
ParkMiller	185, 239
RAND	48, 256
LCG64	keine
LMC2	30, 53, 111, 139, 164, 198
LMC3	62, 124, 154
LMC4	105, 114
CLCG2	16, 109, 152
CLCG3	75, 125, 157, 188, 192, 209
CLCG4	7, 61, 139, 159, 192, 226, 244
EINV	123, 152, 182
EINVLCG64	38

8.2.9 Collision test

Auch in diesem Test erkennt man die Schwächen von Lineare-Kongruenzen-Generatoren mit Modulus 2^e. Der durchgeführte Test entspricht jeweils dem Werfen von 2^{12} Bällen in 2^16 Urnen, wobei gezählt wurde in wievielen Urnen sich danach mehr als ein Ball befindet. Man beachte, dass bei diesem Test insgesamt eine Anzahl von Pseudozufallszahlen verwendet wurde, die ein Vielfaches der Periode der Generatoren ParkMiller, RAND und EINV entspricht. Das die obersten Bits von EINV den Test nicht bestehen, ist nicht verwunderlich, da diese wegen des Modulus $2^{30} + 2^{28} + 3$ nicht gleich verteilt sind.

Tabelle 8.3: Wenn man die Generatoren in n Teilfolgen spaltet, bestehen einige dieser Folgen den collision test nicht. Beim Generator RAND sind nur die einige der schlechtesten Ergebnisse gelistet.

Generator	(Anzahl n der Teilfolgen; betroffene Bits)
ParkMiller	(8; 11), (158; 9)
RAND	(1; 0–13), (2; 0–18), (3; 0–13), (4; 0–18), (5; 0–16), (6; 0–14, 18), (7; 0–17), (8; 0–19), (32; 0–30), (64; 0–30), (96; 0–30), (128; 0–30), (160; 0–30), (192; 0–30), (224; 0–30), (256; 0–30)
LCG64	(2; 26), (64, 0–2), (128, 0–6), (183, 0), (192, 0–2), (256, 0–13)
LMC2	(81, 3), (180, 6), (234, 1)
LMC3	(212, 20)
LMC4	(236, 27)
CLCG2	(84, 23)
CLCG3	(76, 5), (122, 4)
CLCG4	keine Teilfolge betroffen
EINV	(1-256; 28-30)
EINVLCG64	(16; 13), (24; 5), (173; 2), (190; 14), (229; 10)

8.2.10 Serial correlation test

Beim Korrelationstest zeigt sich, dass die beiden Lineare-Kongruenzen-Generatoren mit Modulus 2^{32} (RAND) bzw. 2^64 (LCG64) langreichweitige Korrelationen besitzen. Nimmt man zwei Generatoren RAND und stellt einen von beiden um 2^{29}, 2^{30} oder 2^{31} Schritte vor, so sind beide Folgen stark korreliert. Die gleiche Aussage gilt für LCG64, wenn dieser um 2^{61}, 2^{62} oder 2^{63} Schritte vorgestellt wird. LCG64 ist Bestandteil von EINVLCG64, auch dieser Generator zeigt ab 2^{63} signifikante Korrelationen. Bei den anderen Generatoren finden sich keine Korrelationen.

8.3 Random walks

8.3.1 Random walk in zwei Dimensionen

Bei diesem Test wurden random walker benutzt die mittels leapfrog-Methode auf ein bis 256 Prozessoren aufgeteilt wurden. Jeder der $25 \cdot 8^7$ random walker machte sieben Schritte in acht mögliche Richtungen auf einem zweidimensionalem Gitter. Siebenmal wurde die Verteilung der Endpositionen der random walker untersucht. Wiederum sticht der Generator RAND negativ heraus. In Abbildung 8.1 wird bezogen auf den Generator RAND für jeden Punkt auf dem 15×15-Gitter die auf die jeweilige Wahrscheinlichkeit normierte Differenz zwischen der erwarteten Anzahl von random walker und beobachteten random walker dargestellt. Das heiß, bei einem Wert von 1 beendeten doppelt so viele random walker ihre Wanderung auf einem bestimmten Punkt als erwartet, bei $-0{,}5$ nur halb so viele. Die Pseudozufallszahlenfolge wurde hier auf 256 Prozessoren verteilt.

Tabelle 8.4: Wenn man die Generatoren in n Teilfolgen spaltet, bestehen einige dieser Folgen den random-walk-Test in zwei Dimensionen nicht.

Generator	Anzahl n der Teilfolgen
ParkMiller	35, 143
RAND	64, 128, 135, 192, 193, 256
LCG64	keine
LMC2	keine
LMC3	keine
LMC4	keine
CLCG2	keine
CLCG3	keine
CLCG4	keine
EINV	keine
EINVLCG64	2

8.3.2 S_N-Test

Im Gegensatz zu [Vattulainen 1999] konnte bei den hier vorgestellten Pseudozufallszahlengeneratoren kein Generator gefunden werden, bei dem der empirisch gefundene Exponent γ für das Skalierungsgesetz $S_{N,t} \sim \sqrt{\ln N}\, t^{\gamma}$ vom erwarteten Wert $\frac{1}{2}$ signifikant abweicht. Der Test wurde mit zwei und vier random walker auf dem eindimensionalen Gitter mit 2 500 Schritten durchgeführt. Jeder der random walkers machte insgesamt 1 000 000 Wanderungen.

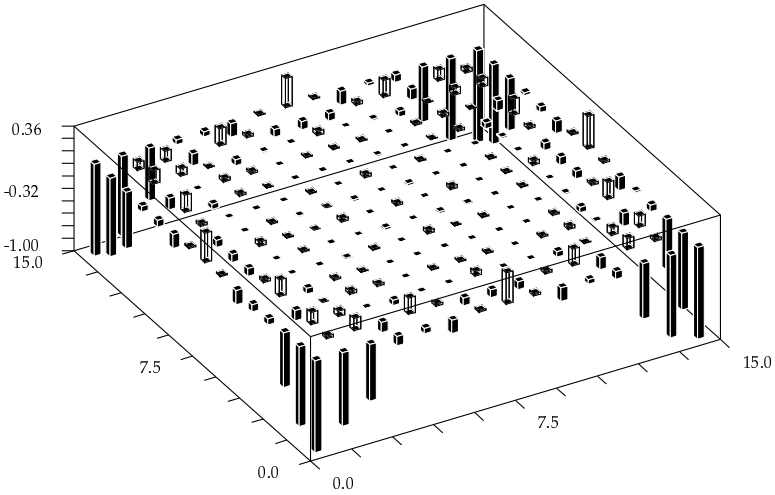

Abbildung 8.1: Relative Abweichung der empirischen Wahrscheinlichkeitsverteilung von der erwarteten für den Generator RAND bei Verteilung der Pseudozufallszahlenfolge auf 256 Prozessoren

Tabelle 8.5: Ermittelte Exponenten γ der Skalierungsfunktion $S_{N,t} \sim \sqrt{\ln N}\, t^{\gamma}$

Generator	Exponent γ	
	2 random walkers	4 random walkers
ParkMiller	0,4998	0,5000
RAND	0,4999	0,5000
LCG64	0,4995	0,4999
LMC2	0,4999	0,4999
LMC3	0,5000	0,4999
LMC4	0,4998	0,4999
CLCG2	0,4997	0,5000
CLCG3	0,4996	0,4997
CLCG4	0,4996	0,4998
EINV	0,5004	0,5001
EINVLCG64	0,4998	0,4997

8.4 Monte-Carlo-Integration

Um die Pseudozufallszahlengeneratoren zu testen, wurde das achtdimensionale Integral

$$\int_{[0,1]^8} \prod_{i=1}^{s} g(x_i, i^2) - 1 \, \mathrm{d}x_1 \, \mathrm{d}x_2 \ldots \mathrm{d}x_8 \qquad \text{mit} \qquad g(x,a) = \frac{|4x-2|}{1+a} \tag{8.4}$$

jeweils 32-mal berechnet. Dabei wurde die Pseudozufallszahlenfolge über 1, 2, 4, 8, 16, 32, 64, 128 bzw. 256 Berechnungen verteilt (leapfrog-Methode). Die aus den Ergebnissen der 32 Integrationen gewonnene Standardabweichung ist ein Maß dafür, ob die Abweichungen für den Integralwert vom exakten Wert Null stochastisch sind. Wenn der Generator gut arbeitet, erhält man Ergebnisse wie in Abbildung 8.2 dargestellt. Auf der Abszisse ist Anzahl der Punkte aus dem Gebiet $[0,1]^8$, an denen der Integrand berechnet wurde, abgetragen, die Ordinate gibt den Integralwert I an. Die hinterlegte Fläche entspricht dem Intervall $[-0{,}485\sigma, 0{,}485\sigma]$. Beim Generator ParkMiller und 16 Teilfolgen liegen die berechneten Werte nicht mehr im Intervall $[-0{,}485\sigma, 0{,}485\sigma]$, siehe Abbildung 8.3. Spektakulär fällt wieder der Generator RAND durch den Test. Der exakte Wert des Integrals liegt für 32, 128 und 256 Teilfolgen weit außerhalb des Intervalls $[-0{,}485\sigma, 0{,}485\sigma]$. InAbbildung 8.4 kann man dies exemplarisch für 32 Teilfolgen sehen. Die anderen Generatoren liefern sämtlich Integralwerte im Intervall $[-0{,}485\sigma, 0{,}485\sigma]$.

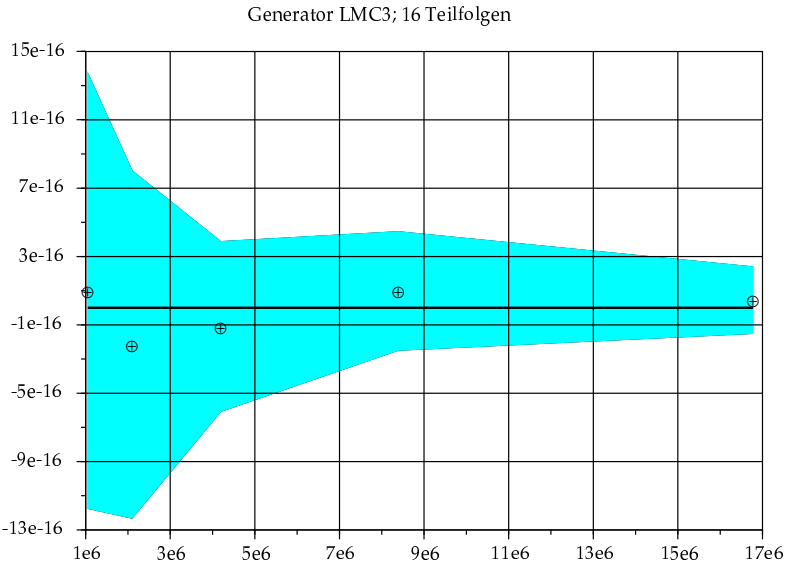

Abbildung 8.2: Monte-Carlo-Integration in acht Dimensionen mit dem Generator LMC3 bei 16 Teilfolgen

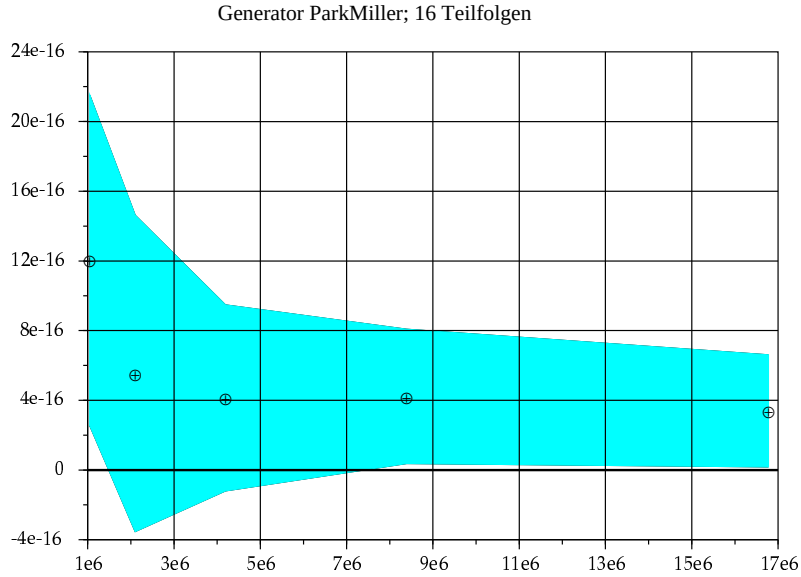

Abbildung 8.3: Monte-Carlo-Integration in acht Dimensionen mit dem Generator ParkMiller bei 16 Teilfolgen

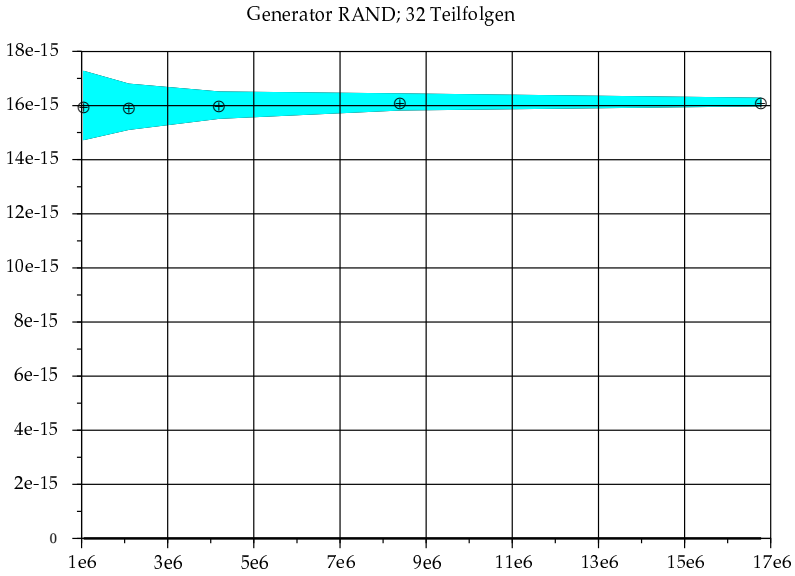

Abbildung 8.4: Monte-Carlo-Integration in acht Dimensionen mit dem Generator RAND bei 32 Teilfolgen

8.5 Parallelisierter WOLFF-Algorithmus am ISING-Modell

Im Gegensatz zu [Ferrenberg und Landau 1992] zeigt keiner der hier getesteten Generatoren (LCG64, LMC2, LMC3, LMC4, CLC2, CLC3, CLC4, EINVLCG64) deutliche Schwächen in der Anwendung beim WOLFF-Algorithmus. Dies gilt sowohl für die Testprozedur mit Stack als auch für die mit Queue. Im Falle der Queue als Datenstruktur liegt die maximale Abweichung vom exakten Wert bei 2,01 Standardabweichungen für die Energie (beim Generator LMC2) und bei Verwendung des Stacks bei 2,23 Standardabweichungen (beim Generator LMC3). Der Test wurde mit 1, 2, 4, 8, 16, 32, 128, 144 und 256 Teilfolgen durchgeführt. Bei so vielen Tests (insgesamt über $10 \cdot 5000$) muss man mit einigen Ausreißern durchaus rechnen. Die Generatoren, die in [Ferrenberg und Landau 1992] den Test nicht bestanden, lieferten systematisch Ergebnisse, die bis über 100 Standardabweichungen neben dem exakten Wert lagen.

8.6 Geschwindigkeit

Die Pseudozufallszahlengeneratoren wurden auf verschiedenen Computern getestet. Damit sich das Testprogramm auf dem HP-Computer compilieren ließ, waren größere Anpassungen des Quellcodes notwendig, da der Compiler noch keine namespaces unterstützte und die STL nur teilweise implementiert war.

Tabelle 8.6: Rechenaufwand um eine bestimmte Anzahl von Zufallszahlen auf verschiedenen Computern zu generieren; Zeitangaben in Sekunden

Generator	Variablentyp	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
Pentium I MMX 200 MHz, gcc V 2.95.2, höchste Optimierungsstufe						
ParkMiller	long	0,04	0,17	0,66	2,65	10,68
	double	0,07	0,26	1,05	4,19	16,80
RAND	long	0,01	0,04	0,17	0,66	2,65
	double	0,04	0,14	0,55	2,20	8,80
LCG64	long	0,02	0,09	0,34	1,37	5,46
	double	0,04	0,18	0,73	2,91	11,62
LMC2	long	0,10	0,42	1,68	6,71	26,84
	double	0,13	0,52	2,06	8,24	32,98
LMC3	long	0,14	0,55	2,19	8,75	35,03
	double	0,16	0,64	2,58	10,31	41,29

Fortsetzung nächste Seite

Generator	Variablentyp	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
LMC4	long	0,17	0,68	2,71	10,86	43,39
	double	0,19	0,78	3,10	12,41	49,64
CLCG2	long	0,10	0,42	1,69	6,80	27,00
	double	0,13	0,52	2,08	8,29	33,15
CLCG3	long	0,15	0,56	2,25	9,00	35,97
	double	0,16	0,67	2,63	10,56	42,24
CLCG4	long	0,18	0,70	2,82	11,26	45,02
	double	0,20	0,80	3,20	12,80	51,19
EINV	long	1,23	4,97	19,84	79,37	317,54
	double	1,26	5,06	20,23	80,96	323,84
EINVLCG64	long	1,25	5,03	20,09	80,36	325,18
	double	1,27	5,14	20,59	82,64	330,40
Pentium III 800 MHz, gcc V 2.95.2, höchste Optimierungsstufe						
ParkMiller	long	0,01	0,04	0,13	0,54	2,15
	double	0,01	0,06	0,23	0,93	3,70
RAND	long	0,00	0,00	0,02	0,09	0,35
	double	0,01	0,02	0,08	0,33	1,30
LCG64	long	0,00	0,02	0,06	0,24	0,95
	double	0,01	0,03	0,11	0,46	1,82
LMC2	long	0,02	0,09	0,38	1,49	5,96
	double	0,03	0,11	0,47	1,87	7,48
LMC3	long	0,02	0,12	0,48	1,90	7,60
	double	0,04	0,14	0,58	2,29	9,18
LMC4	long	0,04	0,15	0,59	2,35	9,44
	double	0,04	0,17	0,68	2,73	10,88
CLCG2	long	0,03	0,09	0,35	1,43	5,71
	double	0,02	0,12	0,45	1,82	7,26
CLCG3	long	0,03	0,13	0,49	1,95	7,84
	double	0,04	0,14	0,59	2,35	9,39
CLCG4	long	0,04	0,15	0,60	2,42	9,67
	double	0,04	0,18	0,70	2,80	11,20
EINV	long	0,25	0,99	3,98	15,89	63,55
	double	0,26	1,02	4,06	16,27	65,09
EINVLCG64	long	0,25	1,00	4,02	16,05	64,24
	double	0,25	1,02	4,07	16,27	65,10
HP N-Class-Server, aCC mit Optimierung -O2						
ParkMiller	long	0,04	0,15	0,61	2,43	9,76
	double	0,04	0,15	0,63	2,52	10,06

Fortsetzung nächste Seite

Generator	Variablentyp	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
RAND	long	0,01	0,06	0,24	0,97	3,87
	double	0,01	0,07	0,27	1,09	4,35
LCG64	long	0,01	0,05	0,19	0,76	3,07
	double	0,01	0,06	0,22	0,90	3,59
LMC2	long	0,11	0,46	1,82	7,28	29,14
	double	0,12	0,46	1,86	7,42	29,69
LMC3	long	0,16	0,65	2,59	10,37	41,49
	double	0,16	0,66	2,64	10,52	42,08
LMC4	long	0,21	0,83	3,34	13,34	53,38
	double	0,21	0,85	3,36	13,48	53,91
CLCG2	long	0,12	0,45	1,80	7,20	28,82
	double	0,12	0,45	1,83	7,29	29,19
CLCG3	long	0,16	0,64	2,54	10,18	40,69
	double	0,16	0,64	2,57	10,27	41,11
CLCG4	long	0,20	0,84	3,36	13,41	53,65
	double	0,21	0,85	3,38	13,50	54,05
EINV	long	1,53	6,16	24,72	99,10	397,33
	double	1,55	6,22	24,89	99,51	398,42
EINVLCG64	long	1,55	6,18	24,82	99,51	398,94
	double	1,56	6,24	24,98	99,91	399,97

Kapitel 9

Zusammenfassung

In dieser Arbeit wurden grundlegende Methoden zur Erzeugung von Pseudozufallszahlen beschrieben. Das besondere Interesse galt hier der Erzeugung von Pseudozufallszahlen auf Parallelrechnern. Dazu wird eine Folge von Pseudozufallszahlen auf verschiedene CPUs aufgeteilt. Dies geschieht entweder indem die ursprüngliche Folge in zusammenhängende Blöcke aufgeteilt wird oder n aufeinander folgende Zahlen fortlaufend auf n CPUs verteilt werden.

Das Hauptergebnis dieser Arbeit ist eine C++-Bibliothek, in der verschiedenen Pseudozufallszahlengeneratoren implementiert wurden, die auf die Verwendung auf Parallelrechnern abgestimmt sind. Die Klassen besitzen Methoden zum Teilen der Pseudozufallszahlenfolge und zum Vervielfältigen der Generatoren. Der Zustand eines Generators kann gespeichert und später rekonstruiert werden. Es können Zahlenfolgen mit verschiedensten Verteilungen erzeugt werden.

Die Generatoren wurden umfangreichen empirischen Tests unterworfen. Als Ergebnis dieser Tests sollen folgende Empfehlungen an den Benutzer dieser Bibliothek ausgesprochen werden:

- Die Generatoren ParkMiller, RAND und EINV sollte man wegen ihren vergleichsweise kurzen Periode nicht in großen Monte-Carlo-Simulationen verwenden, RAND wegen der schlechten statistischen Eigenschaften gar nicht.

- Die Generatoren LCG64 und EINVLCG64 sind Generatoren mit guten statistischen Eigenschaften solange in einer Simulation deutlich weniger als 2^{64} Pseudozufallszahlen verwendet werden. LCG64 hat den Vorteil sehr schnell zu sein, was man sich aber mit den für Lineare-Kongruenzen-Generatoren typischen langreichweitigen Korrelationen erkauft.

- Die Generatoren LMC2, LMC3, LMC4, CLGC2, CLGC3 und CLGC4 bestanden alle empirischen Tests. Für welchen Generator man sich konkret entscheidet, wird in der Praxis von der Anzahl der in der Monte-Carlo-Simulation benötigten Pseudozufallszahlen abhängen.

- Um auszuschließen, dass der Pseudozufallszahlengenerator mit den Modell der Simulation interferiert, ist es ratsam die Simulation mehrmals mit verschiedenen Generatoren durchzuführen.

Dieser Vorschungsbeleg entstand am Institut für Theoretische Physik der Universität Magdeburg. Die empirischen Tests wurden auf Tina, dem Linux-Beowulf-Cluster des Instituts, durchgeführt. Dr. Stephan Mertens, der diese Arbeit betreut hat, möchte ich für Kritik und Anregung danken.

Eine Beispielapplikation

Zuletzt soll hier noch in einer kleinen Beispielapplikation die Verwendung der hier vorgestellten Klassenbibliothek verdeutlicht werden. Die Mathematik kennt zahlreiche Formeln (Reihendarstellungen) für die Berechnung der Kreiszahl π. Aber auch mit stochastischen Methoden lässt sich π berechnen. Dazu wähle man n zufällige Punkte $(x, y) \in [0, 1]^2$ aus dem Einheitsquadrat. In das Einheitsquadrat sei ein Viertelkreis einbeschrieben. k sei die Anzahl der Punkte im Viertelkreis. Da die gewählten Punkte im Quadrat gleichverteilt sind, ist das Verhältnis von k zu n

$$\frac{k}{n} = \frac{\frac{1}{4}\pi}{1}. \tag{A.1}$$

Die Zahl π lässt sich also aus diesem Verhältnis berechnen.

$$\pi = 4\frac{k}{n} \tag{A.2}$$

Mit folgendem Beispielprogramm kann man π parallel auf verschieden Computern „erwürfeln“, zum Schluss wird der Mittelwert der Berechnungen ausgegeben. Das Programm benutzt für die Kommunikation MPI ([Gropp u. a. 2000]).

Listing A.1: MPI-Programm zum parallelen Erwürfeln von π

```
// ************************************************************
//
// eine Beispielapplikation
//
// Monte-Carlo-pi-Berechnung
//
// ************************************************************


#include <cstdlib>
#include <cmath>
#include <iostream>
#include "trng.h"
#include "mpi++.h"
```

```
using namespace TRNG;
using namespace std;

int main(int argc, char *argv[]) {

  // Gesamtzahl der Punkte im Quadrat
  const long all_samples=1000001;

  // Variablen
  long i, in, num_samples;
  double x, y, pi, pi_mean, pi_sigma;

  // ein Pseudozufallszahlengeneratorobjekt
  EINV r;

  // MPI initialisieren
  MPI::Init(argc, argv);

  // eigenen Rang und Geasamtzahl aller Prozesse bestimmen
  int size=MPI::COMM_WORLD.Get_size();
  int rank=MPI::COMM_WORLD.Get_rank();

  // Pseudozufallszahlenfolge auf die Prozesse aufteilen
  r.split(size, rank);

  // bisher keine Punke im Halbkreis
  in=0l;

  // Anzahl der je Prozess zu ermittelnden Punkte
  num_samples=all_samples/size+1l;
  for (i=0l; i<num_samples; ++i) {
    x=r.uniform();
    y=r.uniform();
    // liegt Punkt im Halbkreis
    if (x*x+y*y<=1.0)
      // ja? inkrementiere in
      ++in;
  }

  // pi berechnen
  pi=4.0*static_cast<double>(in)/static_cast<double>(num_samples);

  // eigener Rang 0?
  if (rank==0) {
    // ja, dann Ergebnisse der anderen Prozesse einsmmeln
    // Feld anlegen
    vector<double> pi_array(size);
    // eigenes Ergebnis ausgeben
    cout << "rank " << 0 << '\t' << pi << endl;
    pi_array[0]=pi;
    pi_mean=pi;
    // einsammeln und ausgeben
    for (i=1l; i<size; ++i) {
      MPI::COMM_WORLD.Recv(&pi, 1, MPI::DOUBLE, i, 0);
      cout << "rank " << i << '\t' << pi << endl;
      pi_array[i]=pi;
      pi_mean+=pi;
    }
    // Mittelwert berechnen und ausgeben
    pi_mean/=static_cast<double>(size);
    cout << "--------------" << endl;
```

```
      cout << "mean\t" << pi_mean << endl;
      // mehr als ein Prozess?
      if (size>1) {
        // dann Standardabweichung berechnen und ausgeben
        pi_sigma=0.0;
        for (i=0l; i<size; ++i)
          pi_sigma+=(pi_array[i]-pi_mean)*(pi_array[i]-pi_mean);
        pi_sigma/=(static_cast<double>(size)-1.0)
                  *static_cast<double>(size);
        pi_sigma=sqrt(pi_sigma);
        cout << "sigma\t" << pi_sigma << endl;
      }
    } else
      // sonst Ergebnis versenden
      MPI::COMM_WORLD.Send(&pi, 1, MPI::DOUBLE, 0, 0);

    // MPI beenden
    MPI::Finalize();

    return(EXIT_SUCCESS);
  }
```

Multiplikatoren für lineare Kongruenzen

gute Multiplikatoren für Pseudozufallszahlengeneratoren mit linearen Kongruenzen $r_i = a \cdot r_{i-1} \mod m$ mit $m = 2147482859$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
26330520	0,7571	0,7941	0,8144	0,7528	0,7721	0,7528	0,7559	0,7713
27686330	0,8213	0,9028	0,7880	0,8515	0,7706	0,7598	0,7574	0,8073
54087806	0,7534	0,8325	0,7961	0,8252	0,8049	0,7504	0,7590	0,7888
55566740	0,8124	0,8815	0,7760	0,7732	0,7526	0,7645	0,7544	0,7878
79514916	0,7571	0,7941	0,8144	0,7528	0,7721	0,7528	0,7559	0,7713
134456737	0,9089	0,8791	0,7979	0,7524	0,8069	0,7845	0,7816	0,8159
162143541	0,8810	0,7943	0,7955	0,7989	0,7787	0,7653	0,7605	0,7963
214691359	0,8124	0,8815	0,7760	0,7732	0,7526	0,7645	0,7544	0,7878
245459988	0,9013	0,8609	0,7870	0,8093	0,8008	0,7575	0,7513	0,8097
274923700	0,7871	0,8175	0,8016	0,7753	0,7956	0,7868	0,7605	0,7892
320214076	0,8613	0,9110	0,8185	0,8405	0,7649	0,7528	0,7544	0,8148
333695781	0,7611	0,8303	0,7753	0,7804	0,7749	0,7567	0,7756	0,7792
377063564	0,7567	0,8009	0,7778	0,7829	0,7736	0,7746	0,7528	0,7742
390636780	0,9456	0,7640	0,8129	0,7556	0,7508	0,7838	0,7741	0,7981
449317215	0,8810	0,7943	0,7955	0,7989	0,7787	0,7653	0,7605	0,7963
503960459	0,8869	0,8733	0,8344	0,7547	0,7643	0,7769	0,7544	0,8064
618767237	0,8844	0,8568	0,8233	0,8178	0,8249	0,7676	0,7620	0,8195
639676806	0,9000	0,7917	0,7606	0,7731	0,7624	0,7575	0,7696	0,7878
649602696	0,9188	0,8587	0,8044	0,7704	0,8229	0,7583	0,7590	0,8132
683529555	0,9050	0,8667	0,8205	0,7751	0,7673	0,7738	0,7635	0,8103
692373805	0,7611	0,8303	0,7753	0,7804	0,7749	0,7567	0,7756	0,7792
704715182	0,8640	0,8876	0,7570	0,7940	0,7618	0,7536	0,7590	0,7967
735460283	0,9867	0,7509	0,8511	0,7587	0,7811	0,7898	0,7681	0,8123
755742334	0,8139	0,7514	0,8448	0,8015	0,7703	0,7676	0,7620	0,7874
766618277	0,8613	0,9110	0,8185	0,8405	0,7649	0,7528	0,7544	0,8148
893738509	0,9089	0,8791	0,7979	0,7524	0,8069	0,7845	0,7816	0,8159

Fortsetzung nächste Seite

Fortsetzung $m = 2147482859$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
916894978	0,9260	0,8370	0,8449	0,7722	0,7775	0,7504	0,7756	0,8119
929486929	0,8217	0,8046	0,7606	0,7901	0,7953	0,7838	0,7574	0,7876
959371149	0,9188	0,8587	0,8044	0,7704	0,8229	0,7583	0,7590	0,8132
970928145	0,8448	0,7860	0,7709	0,8247	0,7597	0,7614	0,7605	0,7869
986344873	0,9000	0,7917	0,7606	0,7731	0,7624	0,7575	0,7696	0,7878
996785708	0,8217	0,8046	0,7606	0,7901	0,7953	0,7838	0,7574	0,7876
1013155105	0,9211	0,8499	0,7660	0,7879	0,7697	0,7875	0,7544	0,8052
1018905882	0,8351	0,7513	0,8424	0,8695	0,7832	0,7921	0,7635	0,8053
1022304065	0,8213	0,9028	0,7880	0,8515	0,7706	0,7598	0,7574	0,8073
1055571265	0,8057	0,8085	0,7672	0,7531	0,8072	0,7567	0,7831	0,7831
1057246111	0,9152	0,9027	0,7902	0,8194	0,7832	0,7669	0,7666	0,8206
1057291266	0,7567	0,8009	0,7778	0,7829	0,7736	0,7746	0,7528	0,7742
1080570375	0,9447	0,8164	0,8727	0,7820	0,8436	0,7591	0,7651	0,8262
1126648678	0,7574	0,8671	0,7950	0,7643	0,8000	0,7520	0,7890	0,7893
1147364589	0,8057	0,8085	0,7672	0,7531	0,8072	0,7567	0,7831	0,7831
1291008930	0,7609	0,8671	0,8510	0,7855	0,7758	0,7520	0,7845	0,7967
1319242753	0,8767	0,8688	0,7732	0,7719	0,8138	0,7583	0,7590	0,8031
1344856231	0,9152	0,9027	0,7902	0,8194	0,7832	0,7669	0,7666	0,8206
1476555886	0,7609	0,8671	0,8510	0,7855	0,7758	0,7520	0,7845	0,7967
1484554012	0,7574	0,8671	0,7950	0,7643	0,8000	0,7520	0,7890	0,7893
1534452738	0,8711	0,8224	0,7574	0,7865	0,7847	0,7853	0,7696	0,7967
1538293691	0,8844	0,8568	0,8233	0,8178	0,8249	0,7676	0,7620	0,8195
1553833646	0,8139	0,7514	0,8448	0,8015	0,7703	0,7676	0,7620	0,7874
1559642086	0,7686	0,8221	0,7712	0,7957	0,8175	0,7875	0,7574	0,7886
1565280260	0,8952	0,8033	0,8226	0,7914	0,7652	0,8128	0,7620	0,8075
1575819868	0,8711	0,8224	0,7574	0,7865	0,7847	0,7853	0,7696	0,7967
1579831687	0,8957	0,8040	0,7551	0,8403	0,8075	0,7622	0,7681	0,8047
1623949803	0,8767	0,8688	0,7732	0,7719	0,8138	0,7583	0,7590	0,8031
1704683378	0,9260	0,8370	0,8449	0,7722	0,7775	0,7504	0,7756	0,8119
1715276592	0,8957	0,8040	0,7551	0,8403	0,8075	0,7622	0,7681	0,8047
1732971220	0,8448	0,7860	0,7709	0,8247	0,7597	0,7614	0,7605	0,7869
1747712142	0,9410	0,9213	0,7996	0,7568	0,8138	0,7777	0,7559	0,8237
1772432721	0,8869	0,8733	0,8344	0,7547	0,7643	0,7769	0,7544	0,8064
1777059941	0,8640	0,8876	0,7570	0,7940	0,7618	0,7536	0,7590	0,7967
1799153212	0,7871	0,8175	0,8016	0,7753	0,7956	0,7868	0,7605	0,7892
1804408780	0,7843	0,8353	0,7629	0,7990	0,7554	0,7906	0,7620	0,7842
1833525088	0,9211	0,8499	0,7660	0,7879	0,7697	0,7875	0,7544	0,8052
1849068344	0,7534	0,8325	0,7961	0,8252	0,8049	0,7504	0,7590	0,7888
1901226055	0,9456	0,7640	0,8129	0,7556	0,7508	0,7838	0,7741	0,7981

Fortsetzung nächste Seite

Fortsetzung $m = 2147482859$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
1960390801	0,8952	0,8033	0,8226	0,7914	0,7652	0,8128	0,7620	0,8075
1982895280	0,9013	0,8609	0,7870	0,8093	0,8008	0,7575	0,7513	0,8097
1995157022	0,9537	0,8536	0,8416	0,7879	0,7703	0,7504	0,7651	0,8175
2004619313	0,7686	0,8221	0,7712	0,7957	0,8175	0,7875	0,7574	0,7886
2028073966	0,9447	0,8164	0,8727	0,7820	0,8436	0,7591	0,7651	0,8262
2028127843	0,9410	0,9213	0,7996	0,7568	0,8138	0,7777	0,7559	0,8237
2034019811	0,9867	0,7509	0,8511	0,7587	0,7811	0,7898	0,7681	0,8123
2059526553	0,9050	0,8667	0,8205	0,7751	0,7673	0,7738	0,7635	0,8103
2076098085	0,8351	0,7513	0,8424	0,8695	0,7832	0,7921	0,7635	0,8053
2118252412	0,9537	0,8536	0,8416	0,7879	0,7703	0,7504	0,7651	0,8175
2128358145	0,7843	0,8353	0,7629	0,7990	0,7554	0,7906	0,7620	0,7842

gute Multiplikatoren für Pseudozufallszahlengeneratoren mit linearen Kongruenzen $r_i = a \cdot r_{i-1} \mod m$ mit $m = 2147482943$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
225802979	0,8894	0,9157	0,7656	0,8008	0,7664	0,7853	0,7635	0,8124
273990138	0,8323	0,7704	0,7799	0,7923	0,8187	0,7973	0,8021	0,7990
405917409	0,8315	0,8060	0,8872	0,7506	0,7603	0,7981	0,7786	0,8018
408152769	0,8972	0,8198	0,7946	0,7855	0,7594	0,7520	0,7544	0,7947
460729197	0,8894	0,9157	0,7656	0,8008	0,7664	0,7853	0,7635	0,8124
473275611	0,7592	0,7865	0,7660	0,7779	0,7894	0,7512	0,7696	0,7714
476631011	0,9698	0,8132	0,7779	0,7545	0,7697	0,7575	0,7590	0,8002
698505167	0,8689	0,7883	0,7703	0,7530	0,7918	0,7684	0,7590	0,7857
729734838	0,9372	0,7917	0,8240	0,8123	0,7682	0,7591	0,7544	0,8067
737825920	0,9488	0,7848	0,7835	0,7583	0,7514	0,7591	0,7875	0,7962
749416893	0,8295	0,8071	0,7644	0,7651	0,7894	0,7536	0,7590	0,7812
821492751	0,8072	0,8293	0,8058	0,7787	0,7773	0,7591	0,7696	0,7896
867407017	0,9528	0,8294	0,7620	0,7579	0,7582	0,7883	0,7860	0,8049
897027462	0,7601	0,8323	0,8437	0,8159	0,7823	0,7723	0,7741	0,7972
938359667	0,7572	0,7737	0,7626	0,7948	0,7557	0,7799	0,7590	0,7690
1043082965	0,8295	0,8071	0,7644	0,7651	0,7894	0,7536	0,7590	0,7812
1102059829	0,7525	0,8578	0,8123	0,8735	0,7891	0,7559	0,7574	0,7998
1198342912	0,8048	0,9094	0,7757	0,7687	0,7691	0,7520	0,7831	0,7947
1245966270	0,8972	0,8198	0,7946	0,7855	0,7594	0,7520	0,7544	0,7947
1302498050	0,7601	0,8323	0,8437	0,8159	0,7823	0,7723	0,7741	0,7972
1419442085	0,8689	0,7883	0,7703	0,7530	0,7918	0,7684	0,7590	0,7857
1458054656	0,7736	0,8079	0,7606	0,7970	0,7930	0,7875	0,7801	0,7857

Fortsetzung nächste Seite

Fortsetzung $m = 2147482943$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
1477179516	0,9372	0,7917	0,8240	0,8123	0,7682	0,7591	0,7544	0,8067
1577999271	0,9698	0,8132	0,7779	0,7545	0,7697	0,7575	0,7590	0,8002
1633051374	0,7592	0,7865	0,7660	0,7779	0,7894	0,7512	0,7696	0,7714
1671938982	0,9528	0,8294	0,7620	0,7579	0,7582	0,7883	0,7860	0,8049
1695700981	0,8225	0,7688	0,7734	0,7631	0,7667	0,7520	0,7513	0,7711
1696442560	0,7525	0,8578	0,8123	0,8735	0,7891	0,7559	0,7574	0,7998
1708896371	0,7644	0,8131	0,7545	0,7984	0,7538	0,7830	0,7605	0,7754
1830551215	0,8323	0,7704	0,7799	0,7923	0,8187	0,7973	0,8021	0,7990
1882265621	0,8225	0,7688	0,7734	0,7631	0,7667	0,7520	0,7513	0,7711
1963868570	0,8264	0,8067	0,7656	0,7991	0,7761	0,7723	0,7963	0,7918
1975618530	0,8315	0,8060	0,8872	0,7506	0,7603	0,7981	0,7786	0,8018
1993952185	0,7644	0,8131	0,7545	0,7984	0,7538	0,7830	0,7605	0,7754
2027371693	0,8072	0,8293	0,8058	0,7787	0,7773	0,7591	0,7696	0,7896
2034902793	0,8264	0,8067	0,7656	0,7991	0,7761	0,7723	0,7963	0,7918
2050253570	0,7572	0,7737	0,7626	0,7948	0,7557	0,7799	0,7590	0,7690
2079900696	0,7736	0,8079	0,7606	0,7970	0,7930	0,7875	0,7801	0,7857
2146149772	0,9488	0,7848	0,7835	0,7583	0,7514	0,7591	0,7875	0,7962

Gute Multiplikatoren für Pseudozufallszahlengeneratoren mit linearen Kongruenzen $r_i = a \cdot r_{i-1} \mod m$ mit $m = 2147482949$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
3923004	0,8658	0,8567	0,8154	0,7544	0,7743	0,7692	0,7559	0,7988
20297108	0,8893	0,8746	0,7859	0,7544	0,7947	0,7622	0,7786	0,8057
88965912	0,7837	0,7955	0,7522	0,7783	0,7982	0,7715	0,7513	0,7758
183665376	0,9003	0,8194	0,8144	0,7910	0,8115	0,7622	0,7513	0,8072
312624697	0,8968	0,7767	0,7863	0,7535	0,7591	0,8003	0,7681	0,7915
359583966	0,8658	0,8567	0,8154	0,7544	0,7743	0,7692	0,7559	0,7988
376010321	0,7738	0,7606	0,7718	0,7824	0,7930	0,7614	0,7651	0,7726
381066617	0,9003	0,8194	0,8144	0,7910	0,8115	0,7622	0,7513	0,8072
386921706	0,8329	0,9454	0,7985	0,7798	0,7921	0,8136	0,7590	0,8173
482451071	0,8404	0,7682	0,8039	0,8308	0,7591	0,7723	0,7816	0,7938
487641310	0,8613	0,9071	0,8033	0,8031	0,7802	0,7815	0,7544	0,8130
506401837	0,8613	0,9071	0,8033	0,8031	0,7802	0,7815	0,7544	0,8130
639965562	0,8968	0,7767	0,7863	0,7535	0,7591	0,8003	0,7681	0,7915
740689028	0,8037	0,8243	0,7771	0,7757	0,7529	0,7606	0,7605	0,7793
819225186	0,7952	0,8155	0,7668	0,7638	0,7554	0,7645	0,7741	0,7765
855203042	0,7837	0,7955	0,7522	0,7783	0,7982	0,7715	0,7513	0,7758

Fortsetzung nächste Seite

Fortsetzung $m = 2147482949$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
889744001	0,8341	0,8803	0,7543	0,8235	0,8000	0,7951	0,7620	0,8070
943314717	0,8893	0,8746	0,7859	0,7544	0,7947	0,7622	0,7786	0,8057
971834035	0,8341	0,8803	0,7543	0,8235	0,8000	0,7951	0,7620	0,8070
989455250	0,7952	0,8155	0,7668	0,7638	0,7554	0,7645	0,7741	0,7765
1023948992	0,8037	0,8243	0,7771	0,7757	0,7529	0,7606	0,7605	0,7793
1025471019	0,8404	0,7682	0,8039	0,8308	0,7591	0,7723	0,7816	0,7938
1028879659	0,8329	0,9454	0,7985	0,7798	0,7921	0,8136	0,7590	0,8173
1043575453	0,7738	0,7606	0,7718	0,7824	0,7930	0,7614	0,7651	0,7726
1103907496	0,7738	0,7606	0,7718	0,7824	0,7930	0,7614	0,7651	0,7726
1118603290	0,8329	0,9454	0,7985	0,7798	0,7921	0,8136	0,7590	0,8173
1122011930	0,8404	0,7682	0,8039	0,8308	0,7591	0,7723	0,7816	0,7938
1123533957	0,8037	0,8243	0,7771	0,7757	0,7529	0,7606	0,7605	0,7793
1158027699	0,7952	0,8155	0,7668	0,7638	0,7554	0,7645	0,7741	0,7765
1175648914	0,8341	0,8803	0,7543	0,8235	0,8000	0,7951	0,7620	0,8070
1204168232	0,8893	0,8746	0,7859	0,7544	0,7947	0,7622	0,7786	0,8057
1257738948	0,8341	0,8803	0,7543	0,8235	0,8000	0,7951	0,7620	0,8070
1292279907	0,7837	0,7955	0,7522	0,7783	0,7982	0,7715	0,7513	0,7758
1328257763	0,7952	0,8155	0,7668	0,7638	0,7554	0,7645	0,7741	0,7765
1406793921	0,8037	0,8243	0,7771	0,7757	0,7529	0,7606	0,7605	0,7793
1507517387	0,8968	0,7767	0,7863	0,7535	0,7591	0,8003	0,7681	0,7915
1641081112	0,8613	0,9071	0,8033	0,8031	0,7802	0,7815	0,7544	0,8130
1659841639	0,8613	0,9071	0,8033	0,8031	0,7802	0,7815	0,7544	0,8130
1665031878	0,8404	0,7682	0,8039	0,8308	0,7591	0,7723	0,7816	0,7938
1760561243	0,8329	0,9454	0,7985	0,7798	0,7921	0,8136	0,7590	0,8173
1766416332	0,9003	0,8194	0,8144	0,7910	0,8115	0,7622	0,7513	0,8072
1771472628	0,7738	0,7606	0,7718	0,7824	0,7930	0,7614	0,7651	0,7726
1787898983	0,8658	0,8567	0,8154	0,7544	0,7743	0,7692	0,7559	0,7988
1834858252	0,8968	0,7767	0,7863	0,7535	0,7591	0,8003	0,7681	0,7915
1963817573	0,9003	0,8194	0,8144	0,7910	0,8115	0,7622	0,7513	0,8072
2058517037	0,7837	0,7955	0,7522	0,7783	0,7982	0,7715	0,7513	0,7758
2127185841	0,8893	0,8746	0,7859	0,7544	0,7947	0,7622	0,7786	0,8057
2143559945	0,8658	0,8567	0,8154	0,7544	0,7743	0,7692	0,7559	0,7988

Gute Multiplikatoren für Pseudozufallszahlengeneratoren mit linearen Kongruenzen $r_i = a \cdot r_{i-1} \mod m$ mit $m = 2147482951$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
22202193	0,9474	0,8263	0,8080	0,7725	0,8043	0,7575	0,7620	0,8111
129022192	0,8170	0,7797	0,7537	0,7598	0,8008	0,7520	0,7528	0,7737
203842426	0,8746	0,7557	0,8463	0,7747	0,7924	0,7567	0,7513	0,7931
209052253	0,9025	0,7790	0,7594	0,7743	0,8141	0,7676	0,7831	0,7971
225418408	0,7555	0,8545	0,7697	0,8111	0,7545	0,7761	0,7513	0,7818
270960248	0,7537	0,7941	0,7659	0,7805	0,7775	0,7512	0,7544	0,7682
272714686	0,7537	0,7941	0,7659	0,7805	0,7775	0,7512	0,7544	0,7682
376555083	0,9657	0,9115	0,8281	0,7682	0,7790	0,7645	0,7711	0,8269
453674028	0,9229	0,9035	0,7872	0,7839	0,7542	0,7630	0,7651	0,8114
470423360	0,9317	0,8293	0,7510	0,7621	0,8333	0,8018	0,7801	0,8128
512938007	0,9025	0,7790	0,7594	0,7743	0,8141	0,7676	0,7831	0,7971
645541517	0,7832	0,7550	0,7650	0,7702	0,7532	0,7551	0,7590	0,7630
744935495	0,9369	0,7839	0,7861	0,7947	0,7915	0,7551	0,7513	0,7999
859361257	0,9343	0,8170	0,7621	0,7567	0,7618	0,7669	0,7513	0,7929
865273208	0,8170	0,7797	0,7537	0,7598	0,8008	0,7520	0,7528	0,7737
1013390747	0,9317	0,8293	0,7510	0,7621	0,8333	0,8018	0,7801	0,8128
1037208473	0,8331	0,8000	0,8325	0,7802	0,7594	0,8010	0,7574	0,7948
1043638356	0,9138	0,9038	0,8475	0,7589	0,7520	0,7723	0,7528	0,8144
1189307301	0,9229	0,9035	0,7872	0,7839	0,7542	0,7630	0,7651	0,8114
1216539347	0,8054	0,9005	0,7688	0,7643	0,7938	0,7807	0,7559	0,7956
1299502298	0,8054	0,9005	0,7688	0,7643	0,7938	0,7807	0,7559	0,7956
1322284174	0,8126	0,7801	0,8371	0,7694	0,7582	0,7528	0,7590	0,7813
1345510147	0,9343	0,8170	0,7621	0,7567	0,7618	0,7669	0,7513	0,7929
1350053210	0,9369	0,7839	0,7861	0,7947	0,7915	0,7551	0,7513	0,7999
1464597337	0,9474	0,8263	0,8080	0,7725	0,8043	0,7575	0,7620	0,8111
1744076095	0,8746	0,7557	0,8463	0,7747	0,7924	0,7567	0,7513	0,7931
1821757854	0,7555	0,8545	0,7697	0,8111	0,7545	0,7761	0,7513	0,7818
1823997492	0,9138	0,9038	0,8475	0,7589	0,7520	0,7723	0,7528	0,8144
1828713613	0,9657	0,9115	0,8281	0,7682	0,7790	0,7645	0,7711	0,8269
1853713065	0,8126	0,7801	0,8371	0,7694	0,7582	0,7528	0,7590	0,7813
1886015324	0,7832	0,7550	0,7650	0,7702	0,7532	0,7551	0,7590	0,7630
1951957619	0,8331	0,8000	0,8325	0,7802	0,7594	0,8010	0,7574	0,7948
2052747842	0,9384	0,8168	0,8210	0,7689	0,8249	0,7536	0,7786	0,8146
2077065358	0,9384	0,8168	0,8210	0,7689	0,8249	0,7536	0,7786	0,8146

Gute Multiplikatoren für Pseudozufallszahlengeneratoren mit linearen Kongruenzen $r_i = a \cdot r_{i-1} \mod m$ mit $m = 2147483647$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
14293512	0,8858	0,8170	0,8066	0,7915	0,7752	0,7966	0,7528	0,8036
187606023	0,7846	0,7684	0,7876	0,7520	0,8132	0,7567	0,7544	0,7738
210014925	0,8937	0,8399	0,8274	0,8559	0,7712	0,7567	0,7711	0,8166
218096097	0,8251	0,8179	0,7858	0,7792	0,7588	0,7860	0,7559	0,7870
233506352	0,9625	0,8106	0,8529	0,7654	0,7585	0,7669	0,7574	0,8106
307370217	0,8251	0,8179	0,7858	0,7792	0,7588	0,7860	0,7559	0,7870
369035587	0,7844	0,8561	0,8921	0,7880	0,7758	0,7614	0,7635	0,8030
379983904	0,8191	0,8861	0,7952	0,8109	0,7811	0,7792	0,7620	0,8048
389713771	0,9062	0,7884	0,7591	0,8046	0,7582	0,7598	0,7771	0,7933
389798186	0,8761	0,7521	0,7915	0,7667	0,8344	0,7512	0,7651	0,7910
498302299	0,8173	0,7545	0,8385	0,7805	0,7508	0,7504	0,7741	0,7809
523007613	0,8742	0,8811	0,8751	0,7531	0,7820	0,7898	0,7620	0,8168
554271143	0,9039	0,7630	0,8186	0,7840	0,8187	0,8055	0,7831	0,8110
588399478	0,8654	0,7991	0,8393	0,8681	0,7953	0,7622	0,7711	0,8144
652892167	0,7912	0,7609	0,8353	0,8141	0,7520	0,7868	0,7605	0,7858
676696642	0,8858	0,8170	0,8066	0,7915	0,7752	0,7966	0,7528	0,8036
756894484	0,8937	0,8399	0,8274	0,8559	0,7712	0,7567	0,7711	0,8166
766688699	0,8742	0,8811	0,8751	0,7531	0,7820	0,7898	0,7620	0,8168
885300443	0,7762	0,8075	0,8357	0,8491	0,7814	0,7738	0,7666	0,7986
912296025	0,7925	0,7533	0,7686	0,7657	0,7829	0,7830	0,7528	0,7713
912589622	0,9281	0,8209	0,7945	0,8034	0,7594	0,7645	0,7559	0,8038
935294390	0,9039	0,7630	0,8186	0,7840	0,8187	0,8055	0,7831	0,8110
989837987	0,7846	0,7684	0,7876	0,7520	0,8132	0,7567	0,7544	0,7738
995272310	0,8408	0,7934	0,7715	0,7557	0,7847	0,7875	0,8079	0,7916
1033683247	0,8654	0,7991	0,8393	0,8681	0,7953	0,7622	0,7711	0,8144
1072285299	0,9625	0,8106	0,8529	0,7654	0,7585	0,7669	0,7574	0,8106
1087046123	0,8743	0,7852	0,7610	0,7659	0,7906	0,7738	0,7711	0,7888
1101592370	0,7844	0,8561	0,8921	0,7880	0,7758	0,7614	0,7635	0,8030
1103150874	0,8313	0,8543	0,7503	0,8073	0,7799	0,7520	0,7590	0,7906
1103673533	0,7525	0,8101	0,8755	0,7820	0,7582	0,7622	0,7904	0,7901
1148155553	0,8173	0,7545	0,8385	0,7805	0,7508	0,7504	0,7741	0,7809
1231562056	0,8743	0,7852	0,7610	0,7659	0,7906	0,7738	0,7711	0,7888
1326972428	0,7912	0,7609	0,8353	0,8141	0,7520	0,7868	0,7605	0,7858
1328793191	0,7525	0,8101	0,8755	0,7820	0,7582	0,7622	0,7904	0,7901
1364327320	0,9062	0,7884	0,7591	0,8046	0,7582	0,7598	0,7771	0,7933
1674674096	0,8408	0,7934	0,7715	0,7557	0,7847	0,7875	0,8079	0,7916
1754698889	0,9281	0,8209	0,7945	0,8034	0,7594	0,7645	0,7559	0,8038
1900894678	0,7925	0,7533	0,7686	0,7657	0,7829	0,7830	0,7528	0,7713

Fortsetzung nächste Seite

Fortsetzung $m = 2147483647$

a	$S_2(a,m)$	$S_3(a,m)$	$S_4(a,m)$	$S_5(a,m)$	$S_6(a,m)$	$S_7(a,m)$	$S_8(a,m)$	$\overline{S(a,m)}$
1913106673	0,8191	0,8861	0,7952	0,8109	0,7811	0,7792	0,7620	0,8048
1957291644	0,8761	0,7521	0,7915	0,7667	0,8344	0,7512	0,7651	0,7910
1977654935	0,7762	0,8075	0,8357	0,8491	0,7814	0,7738	0,7666	0,7986
2016773576	0,8313	0,8543	0,7503	0,8073	0,7799	0,7520	0,7590	0,7906

Literaturverzeichnis

[Anderson 1990] ANDERSON, Stuart L.: Random number gnerators on vector supercomputers and other advanced architectures. In: *SIAM Review* 32 (1990), Juni, Nr. 2, S. 221–251

[Barnett et al. 1996] BARNETT ET AL., R. M.: Review of Particle Physics. In: *Physical Review D* 54 (1996), Nr. 1

[Brent 1992] BRENT, Richard P.: Uniform Random Number Generators for Vector and Parallel Computers. Canberra 0200 ACT, Australia, 1992 (TR-CS-92-02). – Forschungsbericht

[Brent 1994] BRENT, Richard P.: On the periods of generalized fibonacci recurrences. In: *Mathematics of Computation* 63 (1994), S. 389–401

[Buchmann 1999] BUCHMANN, Johannes: *Einführung in die Kryptographie*. Berlin, Heidelberg : Springer, 1999

[Cigler 1995] CIGLER, Johann: *Körper Ringe Gleichungen*. Heidelberg : Spektrum Akademischer Verlag, 1995

[Coddington 1997] CODDINGTON, Paul D.: Random Number Generators for Parallel Computers. In: *NHSE Review* (1997)

[ComScire 2000] COMSCIRE. `http://comscire.com`. 2000

[Eichenauer und Lehn. 1986] EICHENAUER, J. ; LEHN., J.: A non-linear congruential pseudo random number generator. In: *Statical Papers* 27 (1986), S. 315–326

[Eichenauer-Hermann 1993] EICHENAUER-HERMANN, J.: Statistical independence of a new class of inversive congruential pseudorandom numbers. In: *Mathematics of Computation* 60 (1993), S. 375–384

[Entacher u. a. 1998] ENTACHER, Karl ; UHL, Andreas ; WEGENKITTL, Stefan: Linear and Inversive Pseudorandom Numbers for Parallel and Distributed Simulation. In: *Workshop on Parallel and Distributed Simulation*, 1998, S. 90–97

[Ferdinand und Fisher 1969] FERDINAND, Arthur E. ; FISHER, Michael E.: Bounded and Inhomogeneous Ising Models. I. Specific-Heat Anoomaly of a Finite Lattice. In: *Physical Review* 185 (1969), Nr. 2, S. 832–846

[Ferrenberg und Landau 1992] FERRENBERG, Alan M. ; LANDAU, D. P.: Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators. In: *Physical Review Letters* 69 (1992), Nr. 23, S. 3382–3384

[Fishman 1997] FISHMAN, George S.: *Monte Carlo*. Springer, 1997 (Springer Series in Operations Research)

[Forster 1996] FORSTER, Otto: *Algorithmische Zahlentheorie*. Braunschweig : Vieweg Verlag, 1996

[Gell-Mann 1996] GELL-MANN, Murray: *Das Quark und der Jaguar*. Serie Pieper. München, Zürich : Pieper, 1996

[Gropp u. a. 2000] GROPP, William ; LUSK, Ewing ; THAKUR, Rajeev: *Using MPI-2*. MIT Press, 2000

[Hellekalek] HELLEKALEK, Peter. *Inversive Pseudorandom Numbergenerators: Concepts, Results and Links*

[Hopkins 1983] HOPKINS, T. R.: A revised algorithm for the spectral test. In: *Appied Statistics* 32 (1983), Nr. 3, S. 328–335

[James 1990] JAMES, F.: A review of pseudorandom number gernerators. In: *Computer Physics Communications* 60 (1990), S. 329–344

[Karian und Goyal] KARIAN, Zavin ; GOYAL, Rohit. *spectral*. http://www.mapleapps.com/maplelinks/share/spectral.html

[Knuth 1981a] KNUTH, Donald E.: *The art of computer programming*. Bd. 2. Reading, Messachusetts : Addison-Wesley, 1981a

[Knuth 1981b] KNUTH, Donald E.: *The art of computer programming*. Bd. 1. Reading, Messachusetts : Addison-Wesley, 1981b

[L'Ecuyer 1988] L'ECUYER, Pierre: Efficient and protable random number Generators. In: *Communications of the ACM* 31 (1988), Juni, Nr. 6, S. 742–749, 774

[L'Ecuyer 1990] L'ECUYER, Pierre: Random numbers for simulation. In: *Communications of the ACM* 33 (1990), Oktober, Nr. 10, S. 85–97

[L'Ecuyer 1998] Kap. Random Number Generation In: L'ECUYER, Pierre: *Handbook on Simulation*. Wiley, 1998, S. 93–137

[L'Ecuyer 1999] L'ECUYER, Pierre: Good parameters and implementations for combined multiple recursive random number gnerators. In: *Operations Research* 47 (1999), Januar, Februar, Nr. 1, S. 159–164

[Lucha und Schöbel 1993] LUCHA, Wolfgang ; SCHÖBEL, Franz F.: *Gruppentheorie Eine elementare Einführung für Physiker*. Mannheim : Bibliographisches Institut & F. A. Brockhaus AG, 1993 (B. I.-Hochschultaschenbuch)

[Lueker 1980] LUEKER, George S.: Some Techniques for solving Recurrences. In: *Computing Surveys* 12 (1980), Dezember, Nr. 4, S. 419–436

[Marsaglia 1968] MARSAGLIA, George: Random numbers fall mainly in the planes. In: *Proc. Nat. Acad. Sci.*, 1968, S. 25–28

[Marsaglia 1984] MARSAGLIA, George: A current view of random number generators. In: *The Proceedings*, 1984

[Marsaglia 1992] MARSAGLIA, George: The mathematics of random number generators. In: *Proceedings of Sympsia in Applied Mathematics* Bd. 46 American Mathematical Society, 1992, S. 73–90

[Marsaglia 1995] MARSAGLIA, George. `http://stat.fsu.edu/pub/diehard/cdrom/`; `http://stat.fsu.edu/~geo/`. 1995

[Marsaglia und Zaman 1994] MARSAGLIA, George ; ZAMAN, Arif: Some portable very-long-period random number generators. In: *Computers in Physics* 8 (1994), Nr. 1, S. 117–121

[May 1997] MAY, Mike: What is random? In: *American Scientist* (1997), Mai, Juni

[Menezes u. a. 1996] MENEZES, Alfred J. ; OORSCHOT, Paul C. ; VANSTONE, Scott A.: *Handbook of Applied Cryptography*. CRC Press, 1996 (CRC Press Series on Discrete Mathematics and Its Applications)

[Metropolis und Ulam 1949] METROPOLIS, Nicholas ; ULAM, Stanislaw: The Monte Carlo method. In: *J. Am. Statistical Association* (1949), Nr. 44, S. 335–341

[Newman und Barkema 1999] NEWMAN, M. E. ; BARKEMA, G. T.: *Monte Carlo Methods in Statistical Physics*. Oxford University Press, 1999

[Newman und Odell 1971] NEWMAN, Thomas G. ; ODELL, Patrick L.: *The generation of random variates*. London : Griffin, 1971 (Griffin's Statistical Monographs & Courses)

[Park und Miller 1988] PARK, Stephen K. ; MILLER, Keith W.: Random number generators: good ones are hard to find. In: *Communications of the ACM* 31 (1988), Oktober, Nr. 10, S. 1192–1201

[Press u. a. 1997] PRESS, William H. ; TEUKOLSKY, Saul A. ; WILLIAM T. VETTERLING, Brain P. F.: *Numerical recipes in C*. Cambridge University Press, 1997

[Remmert und Ullrich 1995] REMMERT, Reinhold ; ULLRICH, Peter: *Elementare Zahlentheorie*. 2. Basel : Birkhäuser Verlag, 1995

[Vattulainen u. a. 1995] VATTULAINEN, I. ; ALA-NISSILA, T. ; KAKAALA, K.: Physical models as tests of randomness. In: *Physical Review E* 52 (1995), Nr. 3, S. 3205–3214

[Vattulainen 1995] VATTULAINEN, Ilpo: Mission Imposible: Find a random pseudorandom number generator. In: *Computers in Physics* 9 (1995), September/ Oktober, Nr. 5, S. 500–504

[Vattulainen 1999] VATTULAINEN, Ilpo: Framework for testing random numbers in parallel calculations. In: *Physical Review E* 59 (1999), Juni, Nr. 9, S. 7200–7204

[Vesely 1993] VESELY, Franz: *Computational Physics Einführung in die Computative Physik*. WUV-Universitätsverlag, 1993

[Wolff 1989] WOLFF, Ulli: Collective Monte Carlo Updating for Spin Systems. In: *Physical Review Letters* 62 (1989), Nr. 4, S. 361–363