ZUFALLSZAHLEN

Tony Hürlimann

Working Paper

April 1997

INSTITUT D'INFORMATIQUE UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg

site Regina Mundi, rue de Faucigny 2, CH-1700 Fribourg / Switzerland

tony.huerlimann@unifr.ch

phone: ++41 26 300 2845 fax: ++41 26 300 9726

This research is supported by the Federal National Fond of Switzerland and financed by the project no. 1217-45922.95.

Zufallszahlen

Tony Hürlimann, Dr. lic. rer. pol.

Stichworte: Pseudo-Zufallszahlen

Zusammenfassung: Dieses Paper gibt einen Überblick über Generatoren von Pseudo-Zufallszahlen. Es werden portable, effiziente und einfach zu implementierende Funktionen für uniformverteilte Zahlen vorgestellt. Aufwendige Fliesskomma-Operationen werden vermieden, und es werden nur 32-Bit-Integer-Typen verwendet. Es wird kurz ein Einblick in die Theorie der linearen Kongruenzverfahren – die meistverwendeten Generatoren – gegeben. Zudem werden Implementationen für Pseudo-Zufallszahlen für die wichtigsten daraus abgeleiteten Verteilungen vorgestellt. Leider muss hier die dazu gehörige Theorie aus Platzgründen weggelassen werden. Auf einschlägige Literatur wird jeweils verwiesen. Einige Testverfahren, um die Güte von Zufallszahlensequenzen zu untermauern, werden ebenfalls kurz angesprochen.

Dieses Paper scheint so überflüssig zu sein wie die Implementation von mathematischen Standardfunktionen, wird doch heute jede Computersprache mit einem Zufallsgenerator ausgerüstet. Wozu das Rad zweimal erfinden? Leider sind viele Zufallsgeneratoren miserabel, und man sollte, wenn professionelle Simulations-applikationen geschrieben werden muss, jedem Generator misstrauen, bis seine Güte getestet worden ist. Zudem kann ein wenig Theorie über Pseudo-Zufallszahlen, die oft genug gemieden wird, das Verständnis für eine vorsichtige Haltung nur vertiefen, – und das ist beileibe vonnöten.

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

— John von Neumann, 1951

"It is not easy to invent a fool-proof random-number generator."

"…random numbers should not be generated with a method chosen at random."

— Donald E. Knuth, 1969

1. EINFÜHRUNG

Zufallszahlen (Zahlen, die durch "Zufall" gewählt werden) sind ausserordentlich nützlich in einer ganzen Reihe von Anwendungen:

Simulation: Der Computer soll physikalische Phänomene simulieren,

Stichproben (Sampling): Eine kleine Auswahl von Elementen aus eine Grundmenge kann Information überihre Struktur vermitteln,

Numerische Analyse: Komplexe numerische Probleme (wie verschiedene Differentialgleichungen) können nur durch stochastische Approximationsheuristiken angenähert werden.

Algorithmen: Zufallszahlen sind eine geeignete Quelle, um die Effizienz von Algorithmen zu testen,

Spiele: Viele Computerspiele sind ohne Zufallszahlen undenkbar.

Es ist interessant, dass Zufallszahlen in Anwendungen Eingang gefunden haben, die keinerlei probabilistischen Gehalt haben. Viele Probleme, die eine nicht-polynomiale algorithmische Komplexität besitzen, können mit Erfolg durch Approximations-verfahren, wie etwa die Monte-Carlo-Methoden, angegangen werden. Ein kurzes Beispiel mag dies veranschaulichen: Aus einer grossen Menge von Zahlen soll eine ausgewählt werden, die grösser als der Median (der mittlere Wert) der Menge ist. Ein deterministischer Algorithmus muss mindestens die Hälfte aller Zahlen in der Menge prüfen, bis sicher eine Zahl darunter ist, die grösser als der Median ist. Die Monte-Carlo-Methode geht so vor: Wähle aus der Menge k Elemente aus. Wähle daraus das grösste. Die Wahrscheinlichkeit, dass diese Element nicht grösser als der Median ist, ist $1/2^k$. Wird $k > 50$ gewählt, so ist diese Wahrscheinlichkeit kleiner als die Wahrscheinlichkeit, dass der Algorithmus versagt aufgrund eines Hardwarefehlers, der durch die kosmische Strahlung verursacht wird!

Verschiedene Spiele im 17. und 18. Jahrhundert weckten das Interesse an der Erzeugung von Zufallsprozessen. Ausgehend von der Beobachtung, dass das Mittel einer kontinuierlichen Zufallsverteilungsfunktion als Integral darstellbar ist, verlagerte sich das Interesse in 19. Jahrhundert auf approximative Lösungen von Integralen durch die Erzeugung von Zufallszahlen. Lord Rayleigh zeigte 1899, dass ein eindimensionaler Randomwalk eine Approximationslösung für gewisse parabolische Differentialgleichungen liefern kann. Diese Entwicklung wurde im 20. Jahrhundert

systematisch von Courant, Kolmogorov, Petrowsky und andern weiterverfolgt. Während und nach dem Zweiten Weltkrieg, als verschiedene schwierige Probleme der Neutronenfusion im Zusammenhang mit der Kernenergie gelöst werden mussten, schlugen John von Neumann und Stanislaw Ulam verschiedene Randomwalk-Modelle vor, die dann auf dem Computer gelöst werden konnten. Die algorithmische Komplexität hat in den siebziger Jahren das Interesse an den Monte Carlo Methoden intensiviert. Es gibt heute wahrscheinlich kein Gebiet in der Algorithmik, in der nicht stochastische Approximationen hilfreich sein können. Selbst im Bereich der polynomialen Algorithmen haben stochastische Verfahren Einzug gehalten: Ein Beispiel ist der Quicksort, ein anderes das oben vorgestellt Medianproblem.

Leider hat das Bewusstsein, wie wichtig es ist, "gute" Zufallszahlen zu generieren, mit ihrer erweiterten Anwendung nicht Schritt gehalten. Zufallszahlen werden in der Informatik stiefmütterlich behandelt. Man vertraut blindlings den in den Programmiersprachen vordefinierten Zufallsgeneratoren. Dabei gibt es allen Grund, diese kritisch zu betrachten. "Many random-number generators in use today are not very good. There is a tendency for people to avoid learning anything about random-number generators; quite often we find that some old method which is comparatively unsatisfactory has blindly been passed down from one programmer to another, and today's users have no understanding of its limitations." [Knuth, p. 4].

Was Knuth vor 25 Jahren schrieb, gilt auch heute noch weitgehend. Gute Zufallsgeneratoren sind immer noch rar. Dies hat verschiedene Gründe. Vor allem ist es – entgegen unserer Intuition – falsch zu glauben, Zufallszahlen könnten mit Hilfe von "zufällig gewählten Methoden" erzeugt werden. Knuth gibt ein eindrückliches Beispiel. Er gibt folgendes Verfahren für einen "Superzufallszahlen-Generator":

Gegeben sei eine 10stellige Zahl X.

S1: Y := X div 10^9 (* Y ist die rechte Ziffer von X. Die Schritte S2 bis S13 werden Y+1 Mal ausgeführt, *)

S2: Z := (X div 10^8) mod 10. (* Z ist die zweit-rechteste Ziffer von X *). Gehe zu Schritt S(3+Z).

S3: if X<5'000'000'000 then X:=X+5'000'000'000

S4: X := (X*X/100'000) mod 10^10 (* ersetze X durch sein mittleres Quadrat *)

S5: X := (1'001'001'001*X) mod 10^10

S6: if X<100'000'000 then X := X+9'814'055'677 else X := 10^10-X.

S7: X := 10^5*(X mod 10^5) + X div 10^5 (* Vertausche die letzten fünf Ziffern von X mit den ersten fünf. *)

S8: X := (1'001'001'001*X) mod 10^10 (* wie Schritt S5 *)

S9: Verkleinere jede Ziffer in X, die grösser als null ist um eins.

```
S10:  if X<10^5 then X := X*X+99'999          else X := X-99'999
S11:  while X<10^9 do X := 10*X  (* X ist hier sicher nicht null *)
S12:  X := (X*(X-1)/10^5) mod 10^10  (* ersetze X durch die mittleren Ziffern von X*(X-1)
S13:  if Y>0 then Y := Y-1 und gehe zu Schritt S2  else stop; X ist die produzierte Zufallszahl.
```

Dieses Prozedur scheint gänzlich zufällig zu sein. Ist es da nicht plausibel, anzunehmen, dass diese eine endlose Quelle von Zufallszahlen ist? Überhaupt nicht! Als Knuth das Programm zum ersten Mal laufen liess, da konvergierte X in die Zahl 6'065'038'420, welche – durch einen aussergewöhnlichen Zufall (!) – durch die Prozedur wieder in sich selbst übergeht. Mit andern Ausgangszahlen blieb das Programm in Zyklen von 3'000-4'000 Zahlen hängen.

"The moral to this story", schreibt Knuth "is that random numbers should not be generated with a method chosen at random." Die Geschichte sollte uns eine Warnung sein.

In diesem Paper soll allerdings nicht darüber nachgedacht werden, was Zufallszahlen sind, sondern es soll eine kurze, kondensierte Übersicht über gute, portable und effiziente Zufallsgeneratoren gegeben werden. Im Abschnitt 2 werden verschiedene Verfahren und deren theoretischer Hintergrund kurz gestreift. Abschnitt 3 stellt zwei gute Generatoren vor, die uniforme Zufallszahlen produzieren. Dabei werden die neusten Entwicklungen [Kemp 1996] berücksichtigt. Die hier vorgestellten Generatoren für uniforme Zufallszahlen haben einen wichtigen Vorteil: Sie benützen nur 4-Byte-Integer-Zahlen. Überläufe werden speziell behandelt. Abschnitt 4 geht kurz auf die Testverfahren ein, um die "Güte" von Zufallszahlenfolgen zu analysieren. Abschnitt 5 schliesslich gibt verschiedene Verfahren an, mit denen Zufallszahlen anderer Verteilungen generiert werden können.

Zufallszahlen können durch zwei unterschiedliche Prozesse beschafft werden: durch einen physikalischen Zufallsgenerator (Würfel, radioaktiver Zerfall) oder durch einen Algorithmus, der dann sogenannte *Pseudo-Zufallszahlen* liefert. Ein wesentlicher Vorteil dieses Verfahrens ist die Wiederholbarkeit der Zahlen. In diesem Paper befassen wir uns natürlich nur mit Pseudo-Zufallszahlen.

Damit Pseudo-Zufallszahlen auch wirklich "gute" Zufallszahlen sind, müssen diese einige statistische Anforderungen erfüllen. Die Folge der produzierten Zahlen müssen (1) voneinander stochastisch unabhängig und (2) in einem Bereich, sagen wir [0,1], gleichverteilt sein. Diese Anforderungen werden gewöhnlich mit einer Reihe von statistischen Tests geprüft (siehe Abschnitt 4).

Allgemein sind Pseudo-Zufallsgeneratoren algorithmische Verfahren, die eine deterministische Folge von Zahlen aus einem endlichen Abschnitt der ganzen positiven Zahlen im Bereich [m,n] (mit m<n) erzeugen. Diese werden dann im Einheitsintervall [0,1] durch lineare Transformation wie folgt normiert, um uniforme Zahlen zu erhalten:

$$z_{[0,1]} = \frac{z_{[m,n]} - m}{n - m} \qquad (1)$$

Die meisten Zufallsgeneratoren sind durch eine Iterationsbildung F (eine Funktion) definiert, so dass die nachfolgende Zahl eine Funktion der Vorgängerzahl ist:

$$z_{i+1} = F(z_i) \quad i \in \mathrm{N} \qquad (2)$$

Die Zahl z_0 ist vorgegeben und wird *Seed* genannt. Die Formel (2) beschreibt ein diskretes, dynamisches System mit der Zahlenfolge $z_0, z_1, z_2, \ldots$.

(Natürlich sind Generatoren der Form $z_{i+1} = F(z_i, \ldots z_{i-r}) \quad i \in \mathrm{N}$ ebenfalls möglich, werden aber selten verwendet.)

Wegen $z_i \in [m, n] \quad i \in \mathrm{N}$ sind die Zufallszahlen periodisch, d.h. es existiert eine Periodenlänge p mit $z_i = z_{i+p}$.

Die obere Schranke für die Periode ist $n - m$. Kurze Perioden sind unerwünscht, aber lange Perioden garantieren jedoch nicht nicht die “Güte” der Zahlen.

Zufallszahlengeneratoren sollten schnell, einfach in der Benützung und portabel sein. Zudem sollen sie “gute” Zufallszahlen erzeugen; vier Anforderungen, die nicht so leicht zu erfüllen sind, wie es auf den ersten Blick scheint. Allzuoft wird die Güte der Geschwindigkeit geopfert.

Es gibt noch einen andern Grund, wieso Generatoren für Zufallszahlen mittelmässig sind: Die Behandlung des Überlaufs bei ganzzahligen Multiplikationen ist nicht so einfach zu implementieren, wenn keine Fliesskommazahlen verwendet werden. Dies wird hier ausführlich behandelt.

2. VERFAHREN

In diesem Abschnitt werden kurz verschiedene generelle Verfahren der Generierung von Zufallszahlen vorgestellt.

2.1. VON-NEUMANN-GENERATOR

Die Idee geht auf John von Neumann zurück:

1 quadriere eine dreistellige Zahl (Startzahl)

2 entnehme die drei mittleren Ziffern (ergibt die Zufallszahl)

3 Dividiere durch 1000 (um eine Zahl im Bereich [0,1] zu erhalten)

4 Wiederhole 1–3

(Natürlich können auch mehr als dreistelligen Zahlen vewendet werden.) Das Verfahren hat sich nicht bewährt. Schon nach wenigen Iterationen läuft es in kurze Perioden hinein. Zudem zeigt eine kurze Überlegung, dass einmal auftretende Nullen wiederum Nullen produzieren. Diesem Verfahren kommt eigentlich nur historische Bedeutung zu.

2.2. LINEARE KONGRUENZ-VERFAHREN

Die linearen Kongruenz-Verfahren sind wohl die bedeutendsten und am häufigsten verwendeten Typen von Generatoren. Die Iterationgleichung (2) ist durch eine lineare Funktion gegeben wie folgt:

$$z_{i+1} = (a \cdot z_i + b) \bmod m, \quad z_0 \text{ ist Startwert} \qquad (3)$$

wobei gelten muss: $z_0 \geq 0,\ a \geq 0,\ b > 0,\ z_0 < m,\ a < m,\ b < m.$

Bei b>0 sprechen wir von einem *affinen linearen* Generator (3). Ist dagegen b=0, so wird dieser als *multiplikativer* Generator bezeichnet (4).

$$z_{i+1} = (a \cdot z_i) \bmod m, \quad z_0 \text{ ist Startwert} \qquad (4)$$

Nicht jede Belegung der vier Zahlen a, b, m, z_0 führt zu einem zweckmässigen Generator. Verschiedene theoretische Resultate beschränken zunächst die Wahl des Quadrupels:

Der Fall a=1 kann unmittelbar zurückgewiesen werden, da dies zur Sequenz

$$z_n = (z_0 + nb) \bmod m$$

führt, die sicher nicht als zufällig gelten kann. Der Fall a=0 ist noch schlimmer. Wir setzen daher für alle praktischen Fälle $a \geq 2$ voraus.

Dann gilt für den Generator (3) [Knuth S. 10]:

$$z_{n+k} = (a^k z_n + (a^k - 1)c / (a-1)) \bmod m \quad (\text{with } k \geq 0) \qquad (5)$$

Die Rekurrenzformel (5) zeigt, dass eine Subsequenz, bestehend aus den k-ten Elementen von (3), wieder eine lineare Kongruenz-Sequenz ergibt.

Wir wenden uns jetzt der Frage zu, wie die Parameter zu wählen sind.

Die Wahl von *b* kann im wesentlichen reduziert werden auf die Menge {0,1}, denn jede Sequenz (3) kann durch eine affine Transformation auf die Sequenz:

$$z_{i+1} = (a \cdot z_i + 1) \bmod m, \quad z_0 \text{ ist Startwert} \qquad (6)$$

überführt werden (Beweis: Fishman 1996, S. 601). Dies ist ein Hinweis auf die relative Bedeutungslosigkeit von *b* gegenüber *m* und *a*.

Die Wahl des Modulus *m* wird durch zwei Faktoren beeinflusst: die Periodenlänge und die Effizienz. Die Periode sollte natürlich möglichst lang sein. Um die Modulo-Operation möglichst schnell zu machen, könnte *m* beispielsweise aus der Menge $\{2^e : e \in \mathbf{N}\}$ genommen werden. In diesem Fall kann die zeitraubende Modulo-Funktion nämlich durch einen schnellen Register-Shift ersetzt werden.

Theorem 1 [Knuth S.19]: Für den Generator (4) [das heisst wenn b=0], ist die maximale Periode durch λ(m) gegeben. Diese Periode wird erreicht, wenn

(i) z_0 relative-prim zu *m* ist ($\gcd(z_0, m) = 1$),

(ii) *a* ist eine primitive Wurzel von Modulo *m*.

Definition: *a* ist eine primitive Wurzel (primitives Element) von Modulo p^e genau dann, wenn [Knuth S. 19]:

(i) $(p^e = 2,\ a \text{ ist ungerade})$ oder $(p^e = 4,\ a \bmod 4 = 3)$ oder
$(p^e = 8,\ a \bmod 8 = 3, 5, 7)$ oder $(p = 2, e \geq 4,\ a \bmod 8 = 3, 5)$

oder (ii) p ist ungerade, $e = 1,\ a \not\equiv 0 \pmod p,\ a^{(p-1)/q} \not\equiv 1 \pmod p$
$(\forall q\colon q \text{ ist Primddivisor von} p - 1)$

oder (iii) p ist ungerade, $e > 1$, a erfüllt (ii), $a^{p-1} \not\equiv 1 \pmod{p^2}$

Definition: Es sei *m* fakorisierbar als $m = p_1^{e_1} \mathrm{K}\ p_t^{e_t}$; dann ist λ(m) gegeben durch:

$$\lambda(2) = 1, \quad \lambda(4) = 2, \quad \lambda(2^e) = 2^{e-2} \ (\text{mit } e \geq 3)$$
$$\lambda(p^e) = p^{e-1}(p-1) \ \text{ mit } p > 2$$
$$\lambda(p_1^{e_1} \mathrm{K}\ p_t^{e_t}) = \mathrm{lcm}(\lambda(p_1^{e_1}), \mathrm{K}\ , \lambda(p_t^{e_t}))$$

Aus Theorem 1 folgt (bei $b = 0$):

(a) Wenn $m = 2^e$ with $e \in \mathbf{N}$, so kann die Periode höchstens 2^{e-2} sein. In diesem Fall vereinfacht sich das Theorem 1 wie folgt: Die maximale Periode 2^{e-2} wird erreicht wenn $a \equiv \pm 3 \pmod 8$ und z_0 ungerade ist, das heisst, ein Viertel aller Multiplikatoren *a* ergibt eine maximale Periode. (Merke: $a \equiv -3 \pmod 8$ ist dasselbe wie $a \equiv 5 \pmod 8$).

(b) Die maximale Periodenlänge *m*–1 kann erreicht werden, wenn *m* eine

Primzahl ist. In diesem Fall vereinfacht sich die Definition der primitiven Wurzel wie folgt: a ist eine primitive Wurzel von m (Primzahl) genau dann wenn [Fishman S. 593]:

(i) $a^{m-1} - 1 \equiv 0 (\mathrm{mod}\, m)$

(ii) $\forall (i < m-1)$: der Audruck: $(a^i - 1) / m$ ist keine ganze Zahl

Theorem 2 [Knuth S.15]: Für den Generator (3) [das heisst wenn b>0], wird die maximale Periode m genau dann erreicht, wenn:

(i) b relativ-prim zu m ist ($\gcd(b,m) = 1$),

(ii) a–1 ist ein Multipel von p, für alle Primdivisoren p von m.

(iii) a–1 ist ein Multipel von 4, wenn m ein Multipel von 4 ist.

Obwohl die Wahl von m aus der Menge $\{2^e : e \in \mathbf{N}\}$ auf den ersten Blick für die Laufzeit wesentliche Vorteile bietet, können diese Vorteile auf die Menge $\{2^e \pm 1 : e \in \mathbf{N}\}$ durch geschickte Dekomposition (siehe Abschnitt 3) ausgedehnt werden. (Eine Verallgemeinerung auf $\{2^e \pm \gamma : e, \gamma \in \mathbf{N}\}$ findet sich in [Fishman 1996, S. 597f].) In der Menge $\{2^e \pm 1 : e \in \{15..64\}\}$ (siehe Tabelle 1 in [Knuth S.13]) befinden sich genau 4 Primzahlen, nämlich: $2^{17} - 1 = 131071$; $2^{19} - 1 = 524287$, $2^{31} - 1 = 2147483647$ und $2^{16} + 1 = 65537$, wobei die interessanteste für unsere Zwecke die dritte ist. Generatoren von Typ (4) mit $m = 2^{31} - 1$ wurden auch ausgiebig getestet [Fishman/Moore 1986].

Die Wahl von *a* wird im wesentlichen durch die beiden Theoreme bestimmt. Denn wir wollen ja eine möglichst lange Periode erreichen. Man könnte versucht sein, a so zu wählen, dass nur Shift- und Additionsoperationen anfallen. Dies kann erreicht werden, wenn (mit $b = 0, m = 2^e$) a aus der Menge $\{2^\alpha + 3 : \alpha \geq 3\}$ gewählt wird. Dann vereifacht sich (4) nämlich zu [Fishman S. 600]:

$$z_{i+1} = (2^\alpha z_i + 2 z_i + z_i) (\mathrm{mod}\, 2^e) \qquad \{7\}$$

Im allgemeinen Fall von Generator (3) könnte a aus der Menge $\{x^\alpha + 1 : 2 \leq \alpha < e\}$ genommen werden, um ähnliche Laufzeitgewinne zu erzielen [Knuth S.21]. Von beiden Verfahren raten die Autoren jedoch dringend ab, weil die so erhaltenen Generatoren schlechte Zufallsresultate liefern.

Die beiden Theoreme bilden das theoretische Fundament, das eine Auswahl guter, lineare Kongruenz-Verfahren erlaubt. Die Theorie lässt dabei genug Raum für die Wahl des Quadrupels (z_0, a, b, m).

Im Buch von Moeschlin [1995] sind verschiedene – die meisten leider nur mittelmässige – linear-kongruente Generatoren zusammengestellt und werden mit verschiedenen Tests verglichen. Diese sind:

a	b	m	Bezeichnung	p	z_0
134775813	1	2^{32}	Turbo Pascal	2^{32}	$[-2^{31}, 2^{31}-1]$
65539	0	2^{31}	Multi	2^{29}	$[1, 2^{31}-1]$
383	263	10000	Linear1	10000	[1,10000]
12241	11999111	10^8	Linear2	10^8	$[1, 10^8]$

Am besten scheint der Generator von Turbo Pascal abzuschneiden. (Man beachte, dass in diesem Generator auch negative Zahlen anfallen.)

Ein genereller Nachteil von linearen Kongruenz-Verfahren könnte sein, dass aus einer Zahlensequenz einer gegebenen Länge die Zahlen (z_0, a, b, m) in polynomialer Zeit bestimmbar sind. Abgesehen davon (was in der Praxis von eher untergeordneter Bedeutung ist), sind jedoch diese Verfahren die mit Abstand am meisten verbreiteten Verfahren, um Pseudo-Zufallszahlen zu generieren. Die Theorie dazu ist überblickbar und gut ausgebaut, sie sind einfach, und eine Implementation kann effizient gestaltet werden. Ein weiter Vorteil ist auch, dass der Spektraltest [Knuth S. 82ff] alle bekannten schlechten Verfahren ausscheidet. Viele kompliziertere Verfahren liefern eindeutig "schlechte" Zufallszahlen. Ein bekanntes Beispiel ist die Fibonacci-Sequenz $z_i = (z_{i-1} - z_{i-2}) \bmod m$.

Eine verbreitete Auffassung ist, dass ein guter Zufallsgenerator nur ein wenig abgeändert werden muss, um einen noch "besseren" zu erhalten. Knuth gibt ein Beispiel: Das hier vorgestellte lineare Kongruenz-Verfahren $z_{i+1} = (a \cdot z_i + b) \bmod m$ liefert "gute" Zahlen, wenn die Parameter geeignet gewählt sind. Ist es da nicht plausibel anzunehmen, dass der Generator $z_{i+1} = ((a \cdot z_i) \bmod (m+1) + b) \bmod m$ noch bessere Zahlen liefern kann? Die Antwort ist wahrscheinlich nein, denn die gesamte Theorie für lineare Kongruenz-Verfahren kann hier nicht verwendet werden, und *in Ermangelug einer geeigneten Theorie sollte man keinem Zufallsgenerator trauen.*

2.3. ANDERE VERFAHREN

Das lineare Kongruenz-Verfahren kann natürlich verallgemeinert werden zu einer linearen Rekurrenzgleichung:

$$z_i = (a_1 \cdot z_{i-1} + a_2 \cdot z_{i-2} + ... + a_k \cdot z_{i-k} + b) \bmod m. \qquad (8)$$

(Theoretische Hinweise finden sich in Knuth [S. 27].) Ein wesentlicher Nachteil dieser Verfahren ist natürlich, dass *k* Ausgangswerte (seeds) zur Verfügung gestellt werden müssen. Ausserdem ist es nicht so einfach, geeignete Parameter zu finden.

Quadratische Kongruenzverfahren (Quadratrestverfahren) gehen von der Formel

$$z_{i+1} = (c \cdot z_i^{\,2} + a \cdot z_i + b) \bmod m \qquad (9)$$

aus. Ein interessanter Generator von Coveyou [Knuth S. 25] ist gegeben durch die Formel:

$$z_{i+1} = z_i(z_n + 1) \bmod 2^e \quad (\text{mit } z_0 \bmod 4 = 2). \qquad (10)$$

Dieser kann ebenso effizient wie ein lineares Kongruenz-Verfahren implementiert werden, ohne dass ein Überlauf auftritt. Er besitzt interessante Verbindungen zu den von Neumann Generatoren, hat aber längere Perioden.

Der oben erwähnte Nachteil von linearen Kongruenz-Verfahren besteht für bestimmte Quadratrestverfahren nicht mehr. Ein solcher Generator ist durch eine Gleichung wie

$$z_{i+1} = z_i^{\,2} \bmod (p \cdot q), \quad z_0 \text{ ist Startwert; } p, q \text{ sind Primzahlen} \qquad (11)$$

gegeben. Die Vorgängerzahl z_i aus den Zahlen z_{i+1} und $m (= p \cdot q)$ zu berechnen entspricht einer Faktorisierung von *m*. Die Zahl *m* muss aber schon sehr gross sein, damit sie nicht mehr mit heutigen Verfahren in vernünftiger Zeit faktorisiert werden kann. Ein weiterer Nachteil ist die Periodenlänge: Diese ist enttäuschend kurz, zudem hängt sie vom Startwert ab, was ganz unerwünscht ist. Mit $p = 4'783$, $q = 4'027$ und $z_0 = 400, 401, 402, \text{K}$, beispielsweise wurde experimentell eine Periodenlänge von 11'940 ermittelt; für $z_0 = 196$ haben wir gar nur eine Periodenlänge von 3'980 [Moeschlin].

Weitere Verfahren sind denkbar, doch mit jeder Verallgemeinerung wird die Analyse aufwendiger und – was schwerer wiegt – die mathematische Theorie dazu existiert nicht.

3. UNIFORME ZUFALLSZAHLEN – IMPLEMENTATIONEN

Die meistverwendeten Generatoren für Zufallszahlen sind multiplikative lineare Kongruenzverfahren, welche die Formel

$$z_i = az_{i-1} \bmod m \qquad (12)$$

mit $0 < a < m$ verwenden. Um eine uniforme Zufallszahl im Bereich [0,1] zu erhalten, wird die erhaltene Zahl durch *m* geteilt:

$$u_i = z_i / m \qquad (13)$$

Für viele Anwendungen hat sich der Generator: $a = 16807, m = 2^{31} - 1$ als ausreichend erwiesen. Dieser war für viele Jahre auch der Standard für das System 360 von IBM [Lewis al. 1969] und ist einer der in Simulationssoftware am häufigsten verwendeten Generatoren.

Allerdings muss, um die Zufallsfolge exakt zu berechnen, mit Zahlen bis zu 2^{45} gearbeitet werden können, was bei Integer-Zahlen mit vier Bytes zu einem Überlauf führt. Die Standardlösung für dieses Problem bestand lange Zeit darin,auf Fliesskomma-Arithmetik mit doppelter Präzision auszuweichen. Dies hat jedoch den Nachteil, dass die Berechnung ineffizient wurde. In FORTRAN wurde also die Instruktion

```
Z = DMOD(16807.0D0*Z , M)
```

verwendet. Diese Lösung steht aber anderen Sprachen oft gar nicht zur Verfügung, da die Modulo-Operation für Fliesskommazahlen nicht implementiert ist.

Daher wurde vorgeschlagen [Bratley al. 1983], die Berechnung so zu zerlegen, dass die Zwischenresultate nur im Bereich $[0, 2^{31} - 1]$ anfallen. Um dies zu ermöglichen, definiere man zwei positive Zahlen *q* und *r* so, dass $m = aq + r$; oder: $q = \lfloor m / a \rfloor$, $r = a \bmod m$. Es ist dann leicht einzusehen, dass

$$\begin{aligned} az \bmod m &= (az - m\lfloor z / q \rfloor) \bmod m \\ &= (az - (aq + r)\lfloor z / q \rfloor) \bmod m \\ &= (a(z - q\lfloor z / q \rfloor) - r\lfloor z / q \rfloor) \bmod m \\ &= (a(z \bmod q) - r\lfloor z / q \rfloor) \bmod m \end{aligned}$$

Wenn $a^2 \le m$, dann ist $r < a \le q$. Zudem gilt: $a(z \bmod q) < aq \le m$ und $r\lfloor z / q \rfloor < z \le m$.

Eine konkrete Implementation dieser Zerlegung ist dann (in Pascal):

```
var z:longint;
```

```
function Uniform0:REAL;
CONST a=16807; m=2147483647; q=m DIV a; r=m MOD a;
VAR k:longint;
BEGIN
  k:=a*(z MOD q) - r*(z DIV q);
  IF k>0 THEN z:=k ELSE z:=k+m;
  Uniform0:=z*(1.0/m);
END;
```

Man beachte, dass *z* eine globale Variable ist, die jeweils den Seed angibt. Der Typ *longint* enthält 4 Bytes und muss fähig sein, den Zahlenbereich $[-2^{31},2^{31}]$ darzustellen.

Die folgende Implementation ist in Turbo Pascal etwa um 10% schneller, da eine Modulo Berechnung durch eine Multiplikation ersetzt wird:

```
function Uniform:REAL;
CONST a=16807; m=2147483647; q=m DIV a; r=m MOD a;
VAR k:LONGINT;
BEGIN
  k:=z div q;
  z:=a*(z-k*q)-k*r;
  IF z<0 THEN z:=z+m;
  Uniform:=z*(1.0/m);
END;
```

Nach Studien von [Fishman al 1986] sind bestimmte Generatoren mit grösseren a's besser. Der Generator $a = 950'706'376,\ m = 2^{31}-1$ schneidet gemäss dieser Studie am besten ab. Leider kann das erwähnte Zerlegungsverfahren nicht mehr angewendet werden. Die Standardlösung für dieses Problem war [Gentle 1981] wiederum eine Umsetzung in Fliesskomma-Arithmetik mit doppelter Präsision und die Zerlegung in drei FORTRAN Instruktionen wie folgt:

```
Z1 = DMOD(32768.0D0*Z , M)
Z2 = DMOD(8392.0D0*Z , M)
Z = DMOD(29013.0D0*Z1+X2 , M)
```

Da nun für alle drei Modulo-Berechnungen $a^2 \le m$ (wobei jeweils $a = 32'768,\ a = 8'392,\ a = 29'013$) gilt, kann die obige Zerlegung von Bratley hier für jede einzelne Instruktion angewendet werden [Kemp 1996]. Die daraus folgende Implementation ist:

```
function Uniform1:REAL;
CONST a=950706376; m=2147483647;
  a1=32768; q1=65535;  r1=32767; {q1=m div a1; r1=m mod a1;}
  a2=8392;  q2=255895; r2=12807; {q2=m div a2; r2=m mod a2; not quite!}
```

```
  a3=29013; q3=74017;              r3=28426; {q3=m div a3; r3=m mod a3;}
VAR k,z1,z2,x3:LONGINT;
BEGIN
  k:=z div q1;
  z1:=a1*(z-k*q1)-k*r1;
  if z1<0 then z1:=z1+m;
  k:=z div q2;
  z2:=a2*(z-k*q2)-k*r2;
  if z2<0 then z2:=z2+m;
  k:=z1 div q3;
  z1:=a3*(z1-k*q3)-k*r3;
  if z1>0 then z1:=z1-m;
  z:=z1+z2;
  if z<0 then z:=z+m;
  Uniform1:=z*(1.0/m);
END;
```

(Man beachte, dass die in Kemp angegebene Funktion 3 Fehler enthält.) Um die Richtigkeit dieses Generators einwandfrei zu testen, werden fünf Zahlen aus der Sequenz durch folgende Tabelle [Fishman 1996, S. 675] angegeben.

z_0	$z_{1000000}$	$z_{2000000}$	$z_{3000000}$	$z_{4000000}$
1114547998	875023723	1830850445	1751231441	1927519856

(Diese Zahlen wurden durch drei Verfahren unabhängig voneinander getestet: (1) stehen sie im genannten Buch, (2) es wurde eine Bibliothek mit Operationen von exakt arbeitenden ganzen Zahlen verwendet, (3) die obige Funktion wurde in Turbo Pascal ausgeführt. Alle drei Verfahren führten zu den selben Zahlen.)

Die Funktion *Uniform1* ist allerdings um etwa 2.7 Mal langsamer als der Generator $a = 16'807,\ m = 2^{31} - 1$ (*Uniform*).

Das Statistikpaket IMSL [ISML 1987] verwendet ebenfalls die beiden obigen zuletzt erwähnten Generatoren und bietet zudem den Generator: $a = 397'204'094,\ m = 2^{31} - 1$ an.

Turbo Pascal 7.0 wie auch Delphi 2 benützen den Generator:

$$z_{i+1} = (134775813 \cdot z_i + 1) \bmod 2^{32}$$

Dieser ist nach Angaben von [Moeschlin] recht gut, aber wahrscheinlich weniger gut als die oben vorgestellten. Allerdings ist er etwa 3mal schneller als *Uniform*.

TESTVERFAHREN

Wie kann man entscheiden, ob ein Generator “gute” Zufallszahlen produziert? Dies kann natürlich nie mit Sicherheit entschieden werden. Wird eine Person aufgefordert, 100 Ziffern in “zufälliger” Weise aufzuschreiben, so sind die Chancen gross, dass

dabei keine zufällige Ziffernfolge entsteht. Wieso? Weil diese "Nicht-zufälliges" (wie Wiederholen von Ziffern) tunlichst vermeiden wird, obwohl in einer zufälligen Folge im Schnitt ein Zehntel aller Ziffern sich selbst folgen. Andererseits werden eine zufallsgenerierte Tabelle bei genauerem Hinschauen immer Muster entdeckt werden können und somit als "nicht-zufallsgeneriert" betrachtet. Die Quintessenz daraus ist, dass wir unserer Intuition nicht vertrauen sollten, sondern mechanische Tests einsetzen müssen. Nun gibt es unendlich viele Tests, die man sich ausdenken kann, und es gibt keine Garantie dafür, dass eine Zahlenfolge, die *n* Tests erfolgreich besteht, beim *n*+1-ten Test nicht noch versagt.

In der Regel wird ein halbes Dutzend Tests auf solche Zahlenfolgen angewandt. Zwei wichtige Typen von Anpassungsverfahren können unterschieden werden: der Chi-Quadrat-Test und der Kolmogorov-Smirnov-Test [Knuth S. 35, 41]. Daneben gibt es auch eine Reihe von visuellen Tests.

Beim Chi-Quadrat-Test wird die Zahlenfolge in *n* Klassen partitioniert, und es wird die Summe der quadratischen Abweichungen vom Erwartungswert der entsprechenden Verteilung mit dem Chi-Quadrat-Wert mit Freiheitsgrad $\upsilon = n-1$ verglichen. Ist die Wahrscheinlichkeit zu klein (sprich kleiner als 5%), so bedeutet dies, dass die Abweichungen zu gross sind, die Zahlenfolge der Verteilung also nicht entspricht. Die Hypothese, dass die Zahlenfolge der Verteilung "entspricht" wird daher abgelehnt. Ist die Wahrscheinlichkeit zu hoch (sprich grösser als 95%), so bedeutet dies, dass die Zahlenfolge zu perfekt der Verteilung entspricht. Die Hypothese wird daher ebenfalls abgelehnt. Im Wahrscheinlichkeitsbereich [5%,10%] und [90%,95%] sollte die Hypothese zumindest als "suspekt" betrachtet werden.

Ein Beispiel für den Chi-Quadrat-Test führt das vor Augen: Gegeben sei die Zahlenfolge $u_1, u_2 \ldots, u_n$, welche ganzzahlige, uniform-verteilte Zufallszahlen im Bereich [1, *k*] liefert. Die Zufallszahlen werden in k Klassen eingeteilt und die Häufigkeiten in den Klassen gezählt. Diese serden mit $n_1, n_2 \ldots, n_k$ bezeichnet. Der Erwartungswert der Klassen (bei Gleichverteilung) ist n/k. Für den Chi-Quadrat-Test wird nun die Summe der quadratischen Abweichungen der Häufigkeiten vom Erwartungswert der Klasse berechnet. Wir berechnen also den Wert [Knuth S. 37]:

$$V = \sum_{1 \le i \le k} \frac{(n_i - \frac{n}{k})^2}{\frac{n}{k}} = \frac{1}{n} \sum_{1 \le i \le k} \left(\frac{{n_i}^2}{\frac{1}{k}} \right) - n \qquad (14)$$

Dieser Wert wird mit dem Wert der Chi-Quadrat-Verteilung (Tabelle Knuth S. 39) verglichen.

Der Kolmogorov-Smirnov-Test wird gewöhnlich für kontinuierliche Verteilungen angewandt. Eine ausführliche Beschreibung gibt Knuth S. 41ff.

Beide Testtypen können für verschiedene praktische Tests eingesetzt werden.

Der Gleichverteilungstest (oder **Häufigkeitstest)**: Die Zahlenfolge wird in *k* Klassen partitioniert. Die Häufigkeiten der Klassen werden mit dem Erwartungswert der Klassen verglichen. (Bei uniformen Zahlen ist der Erwartungswert für alle Klassen $1/k$.)

Der Paartest: Die Zahlenfolge wird in Paare aufeinanderfolgender Zahlen eingeteilt. Diese werden in die k^2 mögliche Klassen partitioniert. Die Häufigkeiten der Paare wird mit dem Erwartungswert der Klassen verglichen. (Bei uniformen Zahlen ist der Erwartungswert für alle Klassen $1/k^2$.) Auch Zahlentripel, Quadrupel oder längere Tupel sind möglich.

Der Permutationstest: *t* aufeinanderfolgende Zufallszahlen werden jeweils zu Klassen zusammengefasst. Dabei werden den Zufallszahlen Ränge innerhalb der Klassen zugeordnet. Wir setzen natürlich voraus, dass die *t* Zufallszahlen alle verschieden sind. Eine solche Annahme ist gerechtfertigt, denn die Wahrscheinlichkeit, dass zwei Realisierungen gleich sind, ist gleich null. Die Ränge innerhalb einer Klasse definieren eine Permutation der Länge *t*. Die verschiedenen Permutationen werden nun auf Gleichverteilung verglichen. Der Erwartungswert einer Permutation ist $1/t!$.

Der Runtest: Die Zufallsfolge wird in Läufe (monoton aufsteigende (oder absteigende) Untersequenzen zerlegt. Aufeinanderfolgende Läufe sind aber nicht unabhängig. Wenn hingegen die erste Realisierung nach einem Lauf weggelassen wir, sind die Längen der Läufe unabhängig. Dabei ist der Erwartungswert eines Laufes der Länge *i* gleich $1/(i+1)!$. Die verschiedenen Klassen bestehen somit aus Läufe verschiedener Längen. Damit die Klassenbildung nicht ins Unendliche wächst, werden alle Läufe, die länger als eine Zahl *k* sind, in einer Klasse zusammengefasst. Deren Wahrscheinlichkeit beträgt $1/(k+1)!$. (Der Beweis für diese Behauptung findet sich in Moeschlin S. 31.)

Verschiedene andere Testvarianten sind möglich. Knuth etwa gibt noch ein weiteres halbes Dutzend an.

ZUFALLSVERTEILUNGEN

Alle folgenden Zufallsverteilungszahlen werden aus uniformen Zufallszahlen im Bereich [0,1] abgeleitet. Im Prinzip ist es möglich, Zufallszahlen für beliebige Verteilungen direkt zu erhalten, statt über uniforme Zahlen zu gehen. Jedoch haben sich solche Methoden nicht als praktisch erwiesen.

Viele der folgenden Methoden wurden, laut Knuth, bereits von John von Neumann entwickelt und später von verschiedenen Statistikern graduell verbessert. (Es würde hier zu weit führen, die Theorie hinter diesen Methoden zu erläutern. Der interessiert Leser wird jeweils auf die Literatur verwiesen.) Jede der folgenden Verteilung enthält einen oder mehrere Parameter, die in einer Prozedur mit dem Namen *RandXXInit* initialisiert werden muss, wobei *XX* die entsprechende Verteilung angibt. Die Funktion, welche dann eine entsprechende Zufallszahl der Verteilung *XX* zurückgibt, wird jeweils *RandXX* genannt. Man beachte, dass die Parameter jeweils global definiert sind. Alle Prozeduren wurden mit einem visuellen Test auf Richtigkeit überprüft.

UNIFORME VERTEILUNG IM INTERVALL [L,U]:

Diese wird erzeugt durch [Oliver, S. 236, Knuth S. 101]:

```
var Bottom,Interval:longint;
procedure RandIntInit(lower,upper:longint);
begin
  Bottom:=lower; Interval:=upper-lower+1;
end;

function  RandInt:longint;
var r:real;
begin
  r:=Uniform*Interval;
  RandInt:=Bottom+trunc(r);
end;
```

Die Prozedur *RandIntInit* initialisiert die untere und obere ganzzahlige Schranke für die uniformen (ganzzahligen) Zufallszahlen. Die Funktion *RandInt* ergibt eine solche Zahl.

NORMALVERTEILUNG

Es gibt mehrere Methoden, um normalverteilte Zufallszahlen zu generieren. Die erste ist die Polar-Marsaglia Methode [Moeschlin S. 64, Knuth S. 104]:

```
var Mu,Sigma,Saved:real;
procedure RandNormInit(mean,StdDev:real);
begin
  Mu:=mean; Sigma:=StdDev; Saved:=0;
end;

function  RandNorm:real;
{Polar-Marsaglia method}
var r,s,r1:real;
begin
  if Saved=0 then begin
    repeat
      r:=2*Uniform-1;
      s:=2*Uniform-1;
      r1:=(r*r)+(s*s);
    until r1<1;
    r1:=sqrt(-2*ln(r1)/r1);
    Saved:=s*r1*Sigma+Mu;
```

```
    r1   :=r*r1*Sigma+Mu;
  end else begin
    r1:=Saved; Saved:=0;
  end;
  RandNorm:=r1;
end;
```

Man beachte, dass das Verfahren zwei normalverteilte Zahlen liefert: die erste ergibt sich aus dem ersten Lauf, die zweite wird in der Variable *Saved* zurückbehalten. Diese wird beim zweiten Lauf zurückgegeben. Nach Knuth wird die REPEAT-Schlaufe im Schnitt 1.27 Mal durchlaufen, bei einer Standardabweichung von 0.587.

Die zweite Methode ist die Box-Müller-Methode [Moeschlin S. 61]:

```
function  RandNorm1:real;
{Box-Müller method}
var r,s,r1:real;
begin
  if Saved=0 then begin
    r:=Uniform; r:=sqrt(-2*ln(1-r));
    s:=Uniform; s:=2*PI*s;
    r1   :=r*cos(s)*Sigma+Mu;
    Saved:=r*sin(s)*Sigma+Mu;
  end else begin
    r1:=Saved; Saved:=0;
  end;
  RandNorm1:=r1;
end;
```

Auch hier werden, wie oben, zwei Zufallsvariablen geliefert.

Eine dritte Methode leitet sich aus dem zentralen Grenzwertsatz ab [Moeschlin S. 68]:

```
function  RandNorm2:real;
{zentraler Grenzwertsatz}
const n=30;
var u,s:real; i:integer;
begin
  s:=0;
  for i:=1 to n do begin u:=Uniform; s:=s+u; end;
  s:=(s-0.5*n)/sqrt(n/12)*Sigma+Mu;
  RandNorm2:=s;
end;
```

Der Nachteil dieser letzten Funktion besteht darin, dass relativ viele uniforme Zufallszahlen generiert werden müssen, um eine einzige normalverteilte zu erhalten.

Anwendungsbeispiel: Einschlagprofil von Gewehrkugeln um ein Zentrum (eindimensional aufgetragen).

EXPONENTIAL-VERTEILUNG

Diese wird erzeugt durch [Oliver, S. 237, Knuth S. 113, Moeschlin S. 74]:

```
var Mu1:real;
procedure RandExpoInit(mean:real);
begin
  Mu1:=mean;
end;
```

```
function  RandExpo:real;
var r:real;
begin
  r:=Uniform;
  RandExpo:=-ln(r)*Mu1;
end;
```

Anwendung: Zeitintervall zwischen (unabhängigen) Ereignissen (Partikeleinschlag aus kosmischer Strahlung auf eine Fläche, Personen an einem Rendezvous-Punkt, Fahrzeuge an einer Kreuzung).

DIE FERMI-DIRAC-VERTEILUNG

Diese wird erzeugt durch [Moeschlin S. 75]:

```
var Beta:real; k:integer;
procedure RandFermiDiracInit(b:real;k1:integer);
begin
  Beta:=b; k:=k1;
end;

function  RandFermiDirac:real;
{Moeschlin S.76}
var r:real;
begin
  r:=Uniform;
  r:=-ln(exp((1/k)*ln(1/r))-1)*Beta;
  RandFermiDirac:=r;
end;
```

DIE ERLANG-VERTEILUNG

Diese wird erzeugt durch [Moeschlin S. 76]:

```
var nn:integer; alpha:real;
procedure RandErlangInit(a:real;n:integer);
begin
  nn:=n; alpha:=a;
end;

function  RandErlang:real;
var i:integer; r,s:real;
begin
  s:=0;
  for i:=1 to nn do begin
    r:=Uniform;
    r:=-ln(r)/alpha;
    s:=s+r;
  end;
  RandErlang:=s;
end;
```

DIE CHI-QUADRAT-VERTEILUNG

Diese wird erzeugt durch [Moeschlin S. 77]:

```
var nnn:integer;
procedure RandChiQuaInit(n:integer);
```

```
begin
  nnn:=n;
end;

function  RandChiQua:real;
var i:integer; r,r1,s:real;
begin
  s:=0;
  for i:=1 to nnn do begin
    r :=Uniform;
    r1:=Uniform;
    r:=cos(2*PI*r)*sqrt(-2*ln(r1));
    s:=s+r*r;
  end;
  RandChiQua:=s;
end;
```

Die nachfolgenden Verteilungen sind diskrete:

DIE POISSON-VERTEILUNG

Diese wird erzeugt durch [Knuth 117, Oliver 238]:

```
var ExpMean:real;
procedure RandPoissonInit(mean:real);
begin
  ExpMean:=exp(-mean);
end;

function  RandPoisson:longint;
var r,q: real; N:longint;
begin
  N:=0;
  q:=Uniform;
  while q>=ExpMean do begin
    inc(N);
    r:=Uniform;
    q:=q*r;
  end;
  RandPoisson:=N;
end;
```

Anwendung: die Anzahl der Ereignisse (Auftreffen kosmischer Partikel, Fahrzeuge an einer Kreuzung usw.) in einer bestimmten Zeitperiode

DIE GEOMETRISCHE VERTEILUNG

Diese wird erzeugt durch [Knuth S. 116, Oliver S. 238]:

```
var LogProb:real;
procedure RandGeomInit(p:real);
begin
  LogProb:=ln(1-p);
end;

function  RandGeom:real;
var r:real;
begin
  r:=Uniform;
  RandGeom:=trunc(round(ln(r)/LogProb+0.5));  {ceil}
```

```
end;
```

Anwendung: nötige Anzahl Versuche, um ein Produkt zurückzuweisen, bei einer gegebenen, bekannten Durchschnittsrate von fehlerhaften Produkten. (the number of attempts required to make the product fail.)

DIE BINOMIAL-VERTEILUNG

Diese wird erzeugt durch [Oliver S. 239]:

```
var Q,T,ZZ:real; G:boolean; M:longint;
procedure RandBinomInit(n:longint; p:real);
begin
  if p>0.5 then Q:=p else Q:=1-p;        {1-p}
  if p<0.5 then T:=p/Q else T:=(1-p)/Q;  {p}
  ZZ:=exp(ln(Q)*n);                      {(1-p)^N}
  G:=p-0.5>=0;
  M:=n+1;
end;

function  RandBinom:longint;
var c:longint; r,qq:real;
begin
  c:=0;
  qq:=ZZ;
  r:=Uniform;
  while r>qq do begin
    r:=r-qq;
    inc(c);
    qq:=((M/c)-1)*T/Q*qq;
  end;
  if G then RandBinom:=M-c-1 else RandBinom:=c;
end;
```

ZUSAMMENFASSUNG

In diesem Paper wurden portable, effiziente und einfach zu implementierende Funktionen für uniformverteilte Pseudo-Zufallszahlen vorgestellt; portable, da nur Integer-Berechnungen mit 4-Bytes-Zahlen verwendet werden; effiziente, da keine Fliesskomma-Zahlen – wie das häufig der Fall ist – eingesetzt werden, um das Überlaufproblem zu bewältigen; einfach schliesslich, da der resultierende Kode aus wenigen Zeilen besteht und auf der bekannten Modulezerlegung beruht.

Es ist erstaunlich, wie wenig das Wissen um diese Zusammenhänge verbreitet ist. Dem abzuhelfen und zudem das Verständnis für die Problematik der Zufallszahlen zu schärfen, sind zwei Hauptanlieben dieses Papers.

ÜBUNG

Angenommen, Sie wollen eine Zahl zwischen 0 und 9 zufällig generieren. Welche der folgenden Methoden liefert “gute” Zahlen [Knuth, S. 6]:

(a) Telephonbuch aufschlagen und die erste Ziffer wählen, auf die der Finger gesetzt wurde;

(b) wie (a), aber die erste Ziffer der Seitenzahl;

(c) werfe einen Würfel in der Form eines Icosahedrons (reguläres 20-Eck), dessen Seiten mit den Zahlen 0, 0, 1, 1, ... , 9, 9 beschriftet sind, und wähle die Ziffer der Seite, auf welcher der Würfel zu liegen kommt;

(d) setze einen digitalen Geigerzähler eine Minute lang einer radioaktiven Quelle aus. Wähle dann die letzte Ziffer auf der Anzeige;

(e) schaue auf die Uhr und wähle als *n* die Zahl, die der Sekundenzeiger zwischen 6n und 6(n+1) Sekunden anzeigt;

(f) frage einen Freund, sich eine Zahl ausdenken, und nimm diese;

(g) frage einen Feind, sich eine Zahl ausdenken, und nimm diese;

(h) ordne jedem von 10 Pferden, über deren Qualifikation nichts bekannt ist, ein Zahl von 0 bis 9 zu. Nach einem Rennen wähle die Zahl des siegreichen Pferdes.

REFERENCES

BRATLEYP., FOX B.L., SCHRAGE L.E., [1983], Guide to Simulation, Springer-Verlag, New York.

FISHMAN G.S., [1996], Monte Carlo, Concepts, Algorithms, and Applications, Springer, New York.

FISHMAN G., MOORE L.R., [1986], An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^31-1, in: SIAM J. Sci. Stat. Comput. Vol. 7.

GENTLE J.E., [1981], Portability Considerations for Random Number Generators, in: Computer Science and Statistics: Proc. 13th Symposium on the Interface, Eddy W.F.(ed), Springer-Verlag, New York.

ISML, [1987], Stat/Library, ISML Inc., Houston, Texas.

KEMP C.D., [1996], The Construction of Fast Portable Multiplicative Congruential Random Number Generators, in: Communication of the ACM, Vol. 39, No.12, December 1996, p. 119.

KNUTH D.E., [1969], The Art of Computer Programming, Vol. 2 / Seminumerical Algorithms, Addison-Wesley Publ. Comp., Reading, Massachusetts.

KOHLAS J., [????], The Generation of Random Numbers, Working Paper, Institute for Automation and Operations Research, University of Fribourg, Switzerland.

MOESCHLIN O. & POHL C. & GRYCKO E & STEINERT F., [1995], Künstlicher Zufall, Exerimentelle Statistik I, Birkhäuser, Basel.

LEWIS P.A., GOODMAN A.S., MILLER J.M., [1969], A Pseudo-Random Number Generator for the System 360, in: IBM System Journal, Vol. 8.

OLIVER I., [1993], Programming Classics, Implementing the World's Best Algorithms, Prentice Hall, New York.