Zufallsgeneratoren

von
Tobias Litte

Anwendungen der statistischen Physik
WS 2004/05

Inhaltsverzeichnis

1. Einleitung

Heutzutage benötigen wir für Simulationen in der statistischen Physik häufig Zufallszahlen, dabei gibt es jedoch einige Probleme. Denn aus einem Computer, der ja eigentlich konstruiert wurde um logische Schritte schnell und fehlerfrei auszuführen, bekommt man durch Programmierung niemals wirkliche echte Zufallszahlen. Allgemein kann man sagen, dass alle Zufallsgeneratoren wie folgt aufgebaut sind:

$$x_i = f(x_{i-1}, x_{i-2}, ..., x_{i-q})$$

Die einzige echte Zufälligkeit liegt darin, wie die Startwerte festgelegt werden und welche Funktion genau benutzt wird.

2. Linear kongruente Zufallsgeneratoren

Eine Gruppe der Zufallsgeneratoren ist die Gruppe der linear kongruenten Zufallsgeneratoren, bei diesen wird die Zufallszahl aus der zuvor ausgegebenen Zufallszahl mithilfe einer Multiplikation und einer eventuellen Addition, sowie einer abschließenden Modulo Operation berechnet.

$$I_n = (a \cdot I_{n-1} + c) \mod m$$

Diese einfache Formel für die Berechnung der jeweils nächsten Zufallszahl bringt natürlich auch einige Probleme mit sich.

2.1 Bekannte Probleme von linear kongruenten Zufallsgeneratoren

Im nachfolgenden wollen wir die wichtigsten Probleme der linear kongruenten Zufallsgeneratoren betrachten, die meisten dieser Probleme finden sich jedoch auch so oder ähnlich bei anderen Zufallsgeneratoren.

2.1.1 Linearer Zusammenhang

Unter dem linearen Zusammenhang versteht man die Kopplung zwischen zwei aufeinanderfolgenden Zufallszahlen. Diese Kopplung ist bei einem linear kongruenten Zufallsgenerator besonders groß. Am deutlichsten wird dies, wenn man zwei Zufallszahlen nacheinander generiert und diese als Koordinaten für Punkte im zweidimensionalen Raum verwendet.

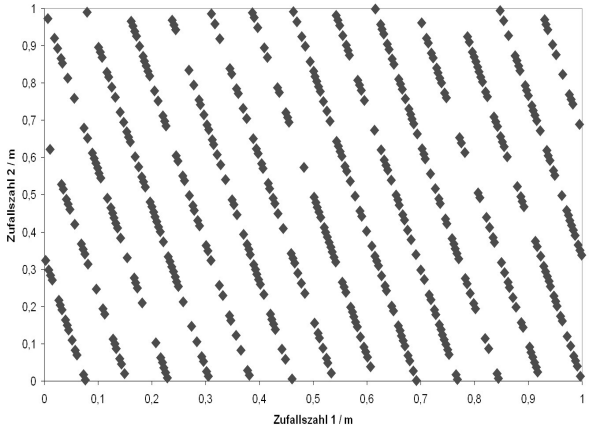

Fig. 1a: Hier wurde ein ungeeigneter Zufallsgenerator verwendet. Alle Zufallszahlen befinden sich auf lediglich 15 verschiedenen Geraden. Als Generator wurde ein linear kongruenter Generator mit a=643, m= 971 und c=0 verwendet

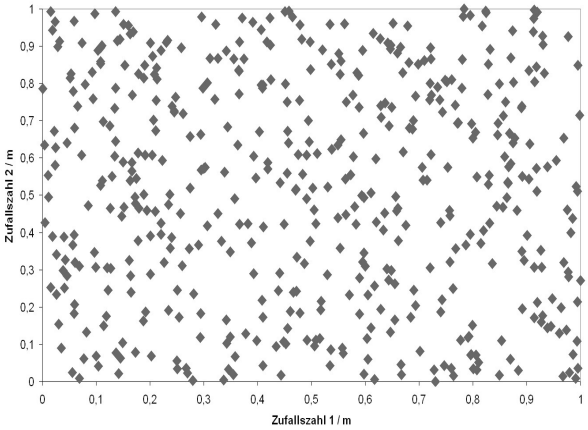

Fig. 1b: Dies ist ein Plot mit annähernd gleicher Anzahl von Punkten wie in a, jedoch diesmal aus einem guten Generator.

Dies kann man auch für den k-dimensionalen Fall verallgemeinern, denn da gilt: Punkte k-dimensionalen Raum liegen immer auf k-1 dimensionalen Hyperebenen. Die maximale Anzahl dieser Hyperebenen wird dabei durch den Modulo Operator m und die Dimension k gegeben. Hierbei gilt:

$$Z_{\max} = m^{1/k}$$

Bei anderen Zufallsgeneratoren kann es jedoch auch zu anderen Arten der Korrelation zwischen den Zufallszahlen kommen. So hat zum Beispiel der lagged Fibonacci Generator eine starke 3 Punkt Korrelation (siehe [2]). Solche Zusammenhänge sollten bei der Wahl für einen Zufallsgenerator für ein bestimmtes Problem immer bedacht werden.

2.1.2 Zufallszahlenmenge

Alle Zufallszahlen entstammen einer Menge, die bedingt durch den Modulo Operator nicht beliebig viele Werte annehmen kann. Also maximal m Elemente hat. Wenn m nicht sorgfältig ausgesucht wurde sogar noch deutlich darunter.

Beispiel: a=23, c=0, m=29

Eigentlich sollten bei diesem Generator insgesamt 29 Werte möglich sein, es werden jedoch nur 7 wirklich angenommen. (Siehe Fig. 2)

Achtung: Der C-Zufallsgenerator rand() hat nur **32767** verschiedene Werte!

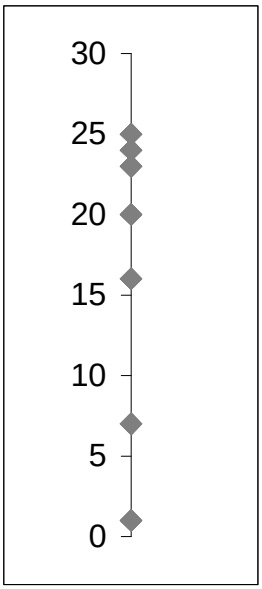

Fig. 2 Auftretende Werte des Zufallsgenerators

2.1.3 Variablenüberlauf

Es soll möglichst viele Zufallszahlen geben, darum wird m sehr groß gewählt. Dies führt jedoch zu Problemen mit der Größe der Variablen, die zu einer ungewollten Modulo Operation durch einen Variablenüberlauf führen kann. Um dieses zu unterbinden hat Schrage einen Algorithmus entwickelt.
Für:

$$z_{neu} = az \mod m \text{ gilt:}$$

$$m = aq + r, \quad q = [m/a], \; r = m \bmod a$$

und damit:

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{falls} \geq 0 \\ a(z \bmod q) - r[z/q] + m & \text{ansonsten} \end{cases}$$

2.1.4 Schwäche der low-order Bits

Bei schlecht gewählten a, c und m können die letzten Bits stark korreliert sein.
Beispiel:

$$I_{n+1} = 65539 \cdot I_n \mod 2^{31}$$

Hier werden als Zufallszahlen nur ungerade Zahlen entstehen, was sich im letzten Bit niederschlägt. Dieses wird sich also NIE ändern! Zwar ist dies ein Extremfall, doch sollte man trotzdem solche Risiken vermeiden und einfach Bits höherer Ordnung nehmen.
Darum immer: $j = \text{int}(10.0 \cdot \text{rand}()/(\text{RAND\_MAX} + 1.0))$
Anstatt: $j = \text{rand}() \bmod 10$

2.2 Beispiele für linear kongruente Zufallsgeneratoren

Als nächstes wollen wir nun ein paar Realisierungen von linear kongruenten Generatoren betrachten.

2.2.1 Minimal Standard Generator

Hierbei handelt es sich um einen ganz einfachen linearen Generator, vorgestellt von Park und Miller im Jahre 1969. Dieser Generator soll kein Allheilmittel sein und ist auch nicht der beste Generator der Welt, sondern es handelt sich hierbei um einen minimalen Standard, an dem sich jeder andere Generator messen lassen muss.

$$a = 75 = 16807 \text{ , } m = 231 - 1 = 2147483647 \text{ , } c = 0$$

Da diese Werte fast immer zu einem Overflow der Variablen führen würden, verwendet man Schrage's Algorithmus, damit ergibt sich:

$$q = 127773 \text{ , } r = 2836$$

Dividiert man die erhaltene Zufallszahl noch durch m, so erhält man eine Zufallszahl zwischen 0 und 1. Eines der Probleme dieses Generators ist, dass auf eine Zahl kleiner als 10^{-6} bei diesem Generator immer eine Zahl unter 0,0168 folgt. Dies führt bei einigen Simulationen zu falschen Ergebnissen, denn statistisch ist dieser Fall eigentlich ziemlich selten.

2.2.2 Gemischter Generator

Dieser Zufallsgenerator von Bays und Durham unterscheidet sich von dem minimal Standard Generator dadurch, dass er einen Zwischenspeicher besitzt. Hier werden 32 Zufallzahlen zwischen gespeichert, die als nächstes ausgegebene Zahl wird durch eine Modulo Operation aus der gerade ausgegebenen Zahl bestimmt Nachdem diese Zahl ausgegeben wurde, wird der frei gewordene Speicherplatz mit einer neuen Zufallszahl gefüllt.

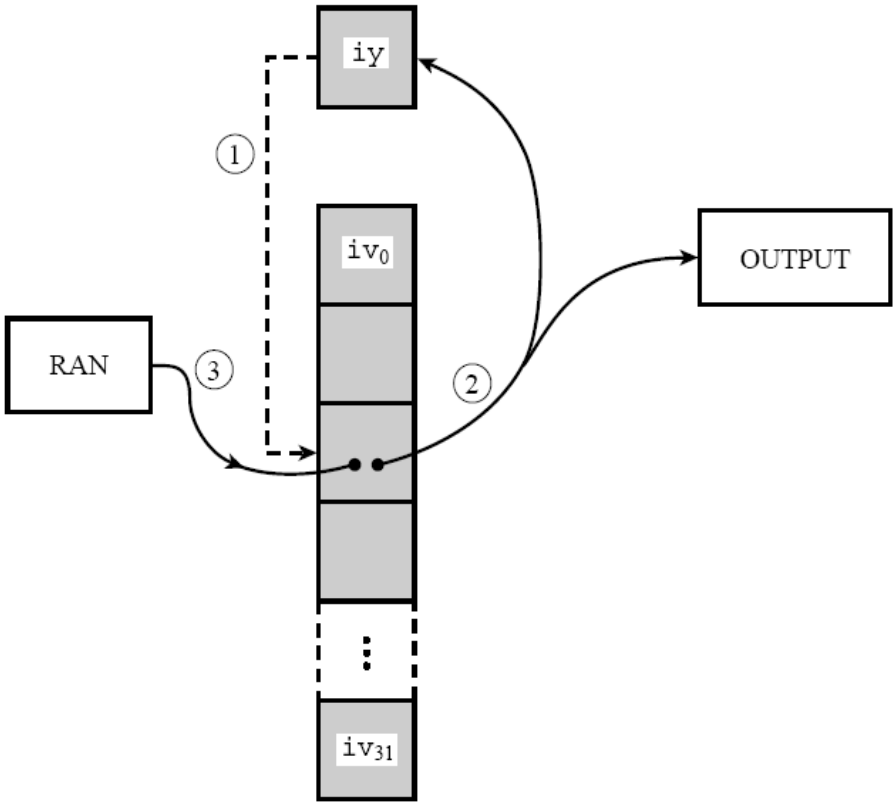

Fig. 3: Der Kasten RAN steht in diesem Fall für den einfachen minimal Standard Generator. Mit den Zufallszahlen aus diesem Generator wird der graue Speicher mit seinen 32 Plätzen befüllt. Die Entscheidung welche Zufallszahl als nächstes ausgegeben wird, geschieht durch eine einfache Modulo 32 Rechnung der zuvor ausgegeben Zufallszahl (1). Danach erfolgt das ausgeben der Zufallszahl (2). Und zu guter letzt wird noch der frei gewordenen Speicherplatz neu befüllt (3). Aus [1]

2.2.3 Kombinierter Generator

Dieser Zufallsgenerator kombiniert den gemischten Generator von Bays und Durham mit einem weiteren linearen Generator, um so die Periode noch weiter zu steigern. Er wurde 1988 von L'Ecuyer vorgestellt. Die gemeinsame Periode dieser beiden Zufallsgeneratoren ist dann ~2,3*10^{18}.
Die Kombination der beiden Zufallsgeneratoren geschieht in dem zuerst der gemischte Zufallsgenerator benutzt wird um eine erste Zufallszahl zu erzeugen und erst danach kommt der zweite einfache linear kongruente Zufallsgenerator ins Spiel. Die Kombination der beiden Zufallszahlen geschieht dann durch eine Subtraktion, bei der die zweite Zufallszahl einfach von der ersten abgezogen wird. Natürlich muss man danach noch eventuelle negative Ergebnisse ins positive überführen.

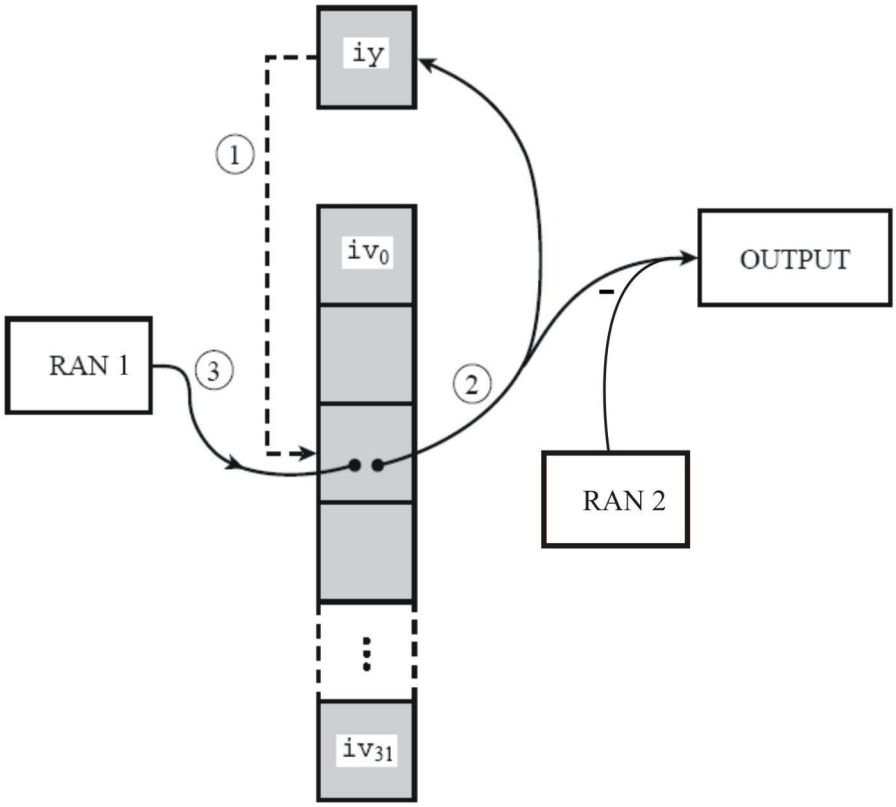

Fig. 4: Im linken Teil des Diagramms ist wieder dasselbe wie beim gemischten Generator aus Fig. 3 zu sehen. Der eigentliche Unterschied ist der einfache linear kongruente Zufallsgenerator RAN2, der eine Zufallszahl generiert, welche von der Zufallszahl aus dem gemischten Generator abgezogen wird.

3. Zufallsgeneratoren mit bestimmten Verteilungen

Häufig werden jedoch nicht gleichverteilte Zufallszahlen benötigt, sondern die Verteilung der Zufallszahlen sollte einer bestimmten Verteilung, z.B. Gaußverteilung, entsprechen. Um eine solche Verteilung zu bekommen erstellt man zuerst eine Zufallszahl mit Gleichverteilung und bearbeitet diese danach so, dass eine Zufallszahl mit der gewünschten Verteilung entsteht. Dazu gibt es im wesentlichen zwei Möglichkeiten:

3.1 Transformationsmethode

Bei der Transformationsmethode bekommen wir die gewünschte Verteilung $p(y)$ über eine Transformation der Gleichverteilung. Dazu müssen wir folgende Differentialgleichung lösen:

$$\frac{dx}{dy} = p(y)$$

Die Lösung davon ist gerade $F(y) = x$ mit $F(y) = \int_0^y p(y')dy'$.

Somit ergibt sich für unsere Transformation *y(x)* folgendes:

$$y(x) = F^{-1}(x)$$

Dies bedeutet, dass wir diese Methode nur anwenden können, wenn die Stammfunktion der Verteilungsfunktion bekannt und invertierbar ist.

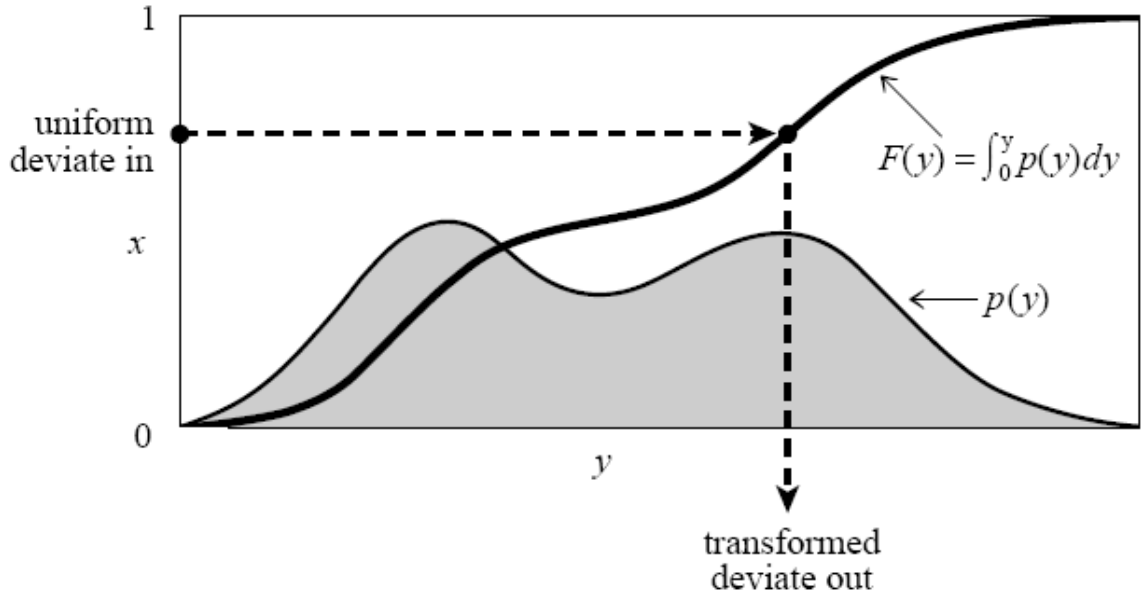

Fig. 5: Grafische Darstellung der Transformationsmethode. Eingezeichnet wurde neben der Verteilungsfunktion auch noch deren Stammfunktion. Wenn man nun eine gleichverteilte Zufallszahl *x* eingibt, so erhält man die gewichtete Zufallszahl als Funktionswert von $F^{-1}(x)$. Aus [1]

Beispiel Gaußverteilung:

Die Gaußverteilung ist eine häufig benötigte Verteilung. Leider lässt sie sich im eindimensionalen Fall die invertierte Stammfunktion bestimmen. Daher verwenden wir einen Trick und betrachten den zweidimensionalen Fall.

Gaußverteilung:

$$f(y) = \frac{1}{\sqrt{2\pi}} \exp(-y^2/2)$$

Hier ergibt die Transformation:

$$y_1 = \sqrt{-2\ln x_1}\cos 2\pi x_2$$
$$y_2 = \sqrt{-2\ln x_1}\sin 2\pi x_2$$

Bzw. nach *x* aufgelöst:

$$x_1 = \exp(-0{,}5\cdot(y_1^2 + y_2^2))$$
$$x_2 = \frac{1}{2\pi}\arctan\frac{y_2}{y_1}$$

Dies kann man wie folgt überprüfen:

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = -\left[\frac{1}{2\pi}\exp\left(-\frac{y_1^2}{2}\right)\right]\left[\frac{1}{2\pi}\exp\left(-\frac{y_2^2}{2}\right)\right]$$

Dies bedeutet, dass wir für eine Gaußverteilte Zufallszahl immer mindestens zwei gleichverteilte Zufallszahlen benötigen, dann aber auch gleich zwei Gaußverteilte Zufallszahlen erzeugen können. Es bietet sich also an die zweite Zufallszahl nicht zu verwerfen, sondern zwischenzuspeichern.

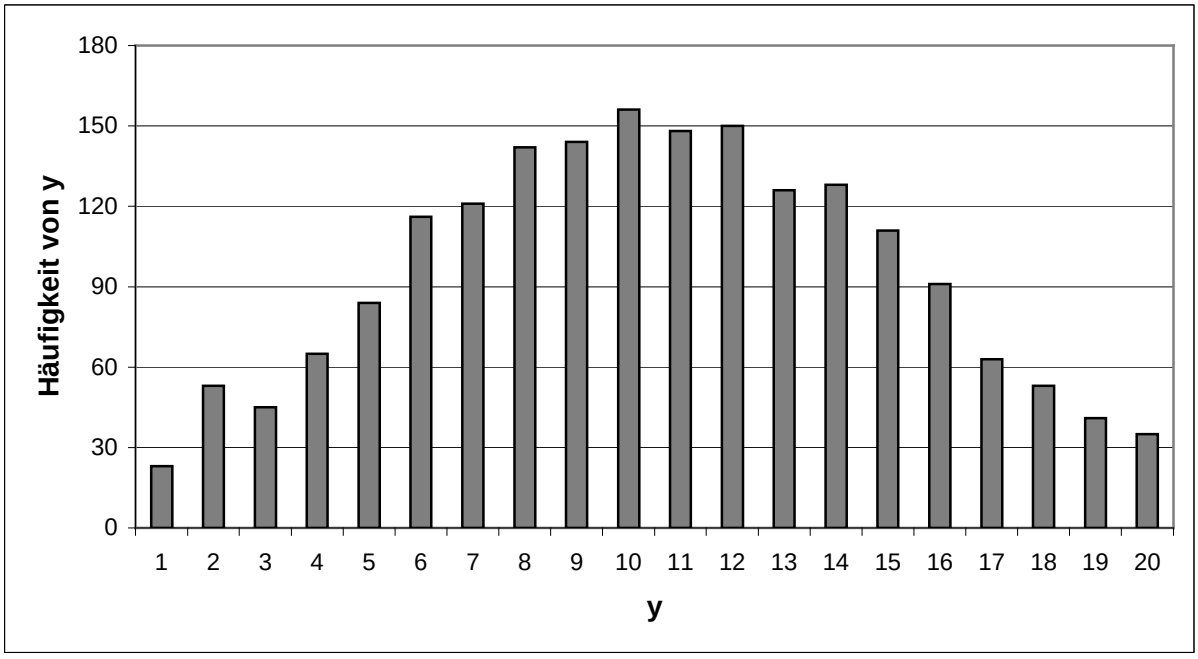

Fig. 6: Gaußverteilung mit 1000 Zufallszahlen.

Das Aufrufen der Trigonometrischen Funktionen kann man sich ersparen, in dem man zwei Zufallszahlen v_1 und v_2 als Punkt innerhalb des Einheitskreises generiert. Dies führt allerdings dazu, dass man einige generierte Zufallszahlen verwerfen muss. Damit ergibt sich:

$$y_1 = \sqrt{\frac{-2\ln(R^2)}{R^2}} \cdot v_1$$

$$y_2 = \sqrt{\frac{-2\ln(R^2)}{R^2}} \cdot v_2$$

Mit:

$$R^2 = v_1^2 + v_2^2 \quad ; \quad R < 1$$

3.2 Ablehnungsmethode

Alternativ kann die Ablehnungsmethode verwendet werden. Hierzu benötigen wir eine Vergleichsfunktion *f(x)* für die gilt:

$$f(x) \geq p(x) \quad \forall\, x \in D$$

Durch eine erste Zufallszahl wird der Arbeitspunkt x_0 wie bei der Transformationsmethode festgelegt. Danach erzeugen wir eine zweite Zufallszahl *z* und überprüfen ob $zf(x_0)$ kleiner oder größer ist als $p(x_0)$. Ist es kleiner so wird x_0

genommen, ist es größer so wird x_0 verworfen. Damit nicht zu viele Werte verworfen werden müssen, sollte *f(x)* möglichst dicht an *p(x)* liegen. Da bei dieser Methode gleich zwei Zufallszahlen generiert werden, ist diese Methode wesentlich langsamer als die Transformationsmethode.

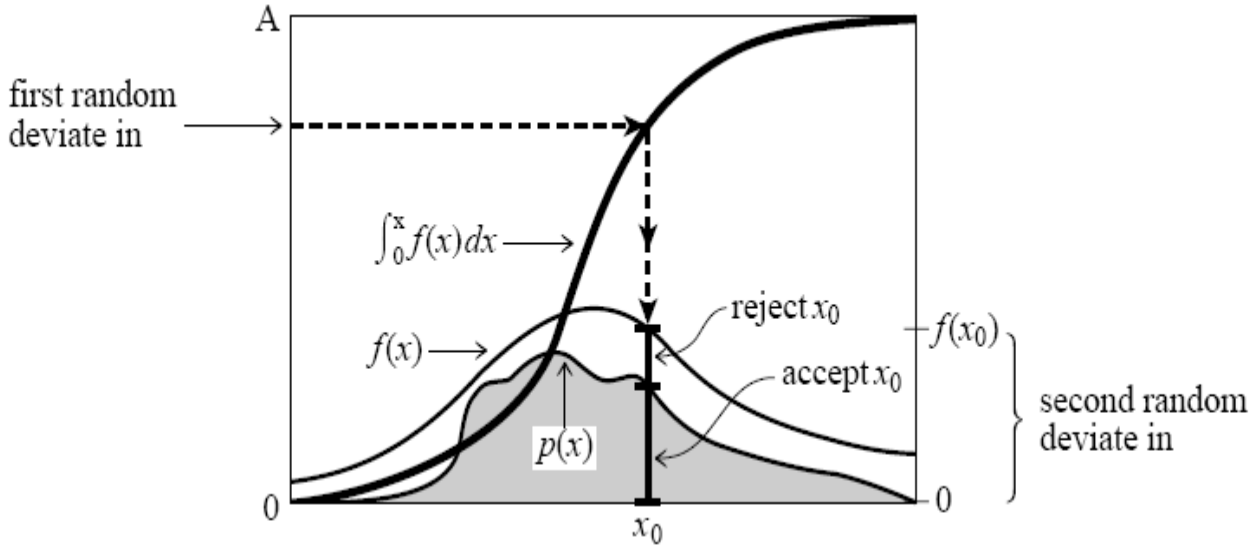

Fig. 7: Zuerst generiert man eine Zufallszahl, die der Verteilungsfunktion f(x) gehorcht. Dann überprüft man mit einer zweiten Zufallszahl, ob die gerade generierte Zufallszahl auch der eigentlichen Verteilung p(x) genügt.

4. Monte Carlo Integration

Man kann Integrationen auch über Zufallszahlen durchführen. Dazu erzeugt man einfach eine Zufallszahl und setzt diese in die Funktion ein. Das Integral erhält man dann als Mittelwert dieser Funktionswerte.

$$\int f\, dV \approx V\langle f\rangle \pm \sqrt{\frac{\langle f^2\rangle + \langle f\rangle^2}{N}}$$

Dabei ist $\langle f\rangle$ das arithmetische Mittel von f.

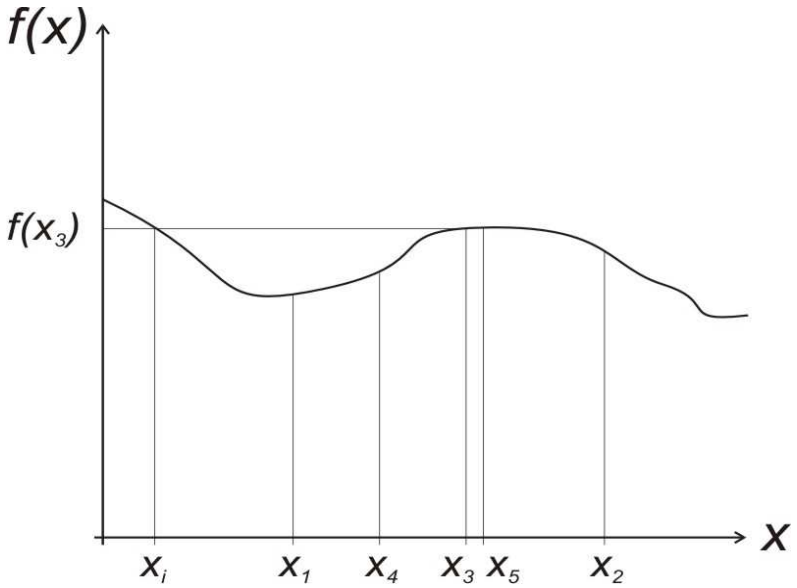

Fig. 8: So in etwa sieht eine Integration nach dem Monte-Carlo Prinzip aus, anstelle von gleichverteilten Stützpunkten wählt man die einzelnen Stützpunkte zufällig aus.

Auch π lässt sich über eine Monte-Carlo Integration berechnen. Hierfür integriert man am besten über die charakteristische Funktion des Einheitskreises in einem Quadranten. Dies bedeutet im Endeffekt, dass man einfach ein Zufallszahlenpaar bildet und dann kontrolliert, ob dieses innerhalb des Einheitskreises liegt oder nicht. Liegt der Punkt innerhalb des Kreises so zählen wir ihn einfach mit einer 1, liegt er außerhalb mit einer 0. π ergibt sich im Endeffekt dann als das vierfache Verhältnis der gezählten Punkte zu den Gesamtpunkten. Damit ergibt sich für 1000 Zufallszahlenpaare π=3,24. Bei 10^9 Zufallszahlenpaaren sieht das Ergebnis mit π=3,141582824 (im Vergleich: π=3,141592654) schon wesentlich besser aus.

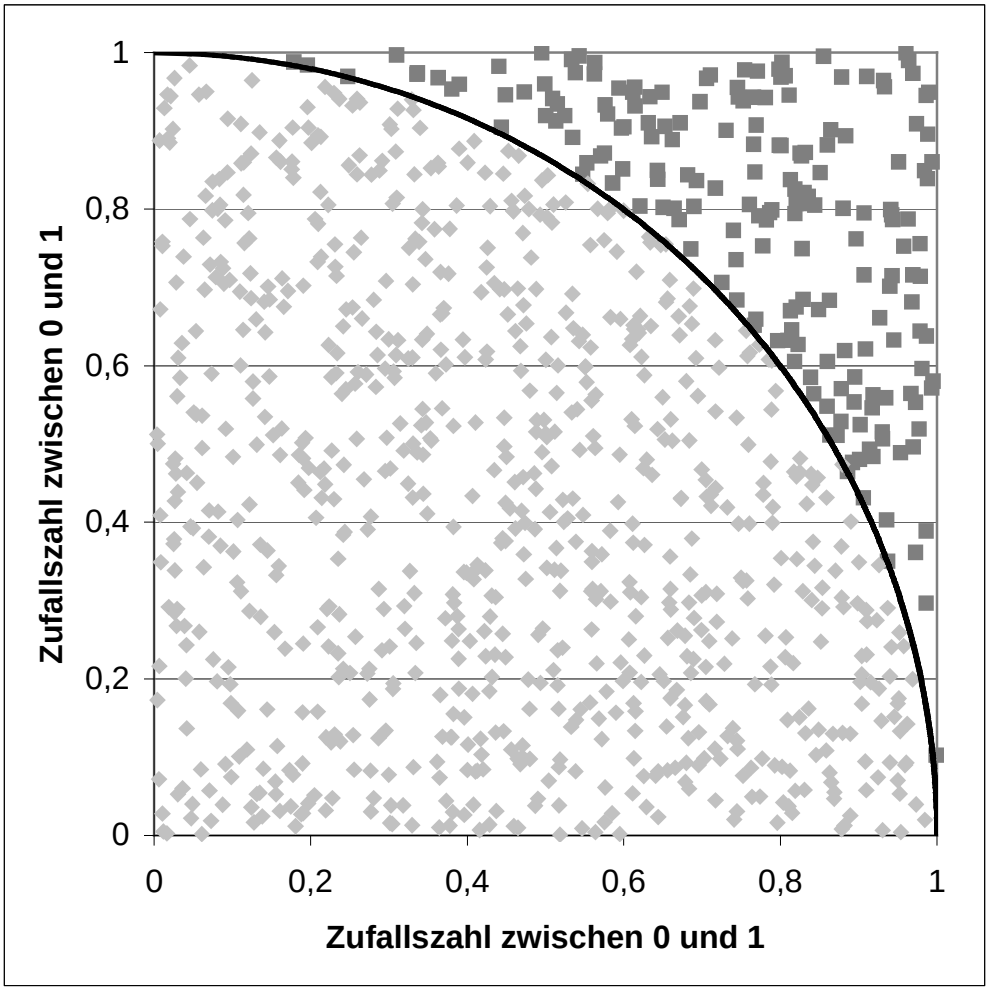

Fig. 9: Dies ist die grafische Darstellung der Berechnung von π mit insgesamt 1000 Punkten, dabei stellen die roten Punkte jene Zahlenpaare dar, die gezählt wurden. Danach wurde $\pi/4$ als Verhältnis dieser Anzahl zu der Gesamtanzahl der Punkte berechnet

Die Monte Carlo Integration ist besonders in höheren Dimensionen interessant, da man hiermit nicht vorher festlegen muss wie viele Stützpunkte man benutzen will. Normalerweise bestimmt man ja ein Integral numerisch, in dem man ein Gitter von Stützpunkten wählt und jeden Stützpunkt einzeln ausrechnet. Das bedeutet man legt die Genauigkeit im Vorfeld nur indirekt über die Anzahl der Stützpunkte fest. Hat das Ergebnis am Ende nicht die nötige Genauigkeit, muss man das ganze Verfahren mit einem feineren Gitter wiederholen. Hier liegt der Vorteil der Monte Carlo Integration, denn man kann einfach so lange weitere zufällige Stützpunkte hinzunehmen, bis man die gewünschte Genauigkeit hat.

5. Entropie von Zufallsgeneratoren

Eine Definition der Entropie auf Ebene der einzelnen Zufallszahlen macht keinen Sinn, da diese über den Generator immer fest miteinander korreliert sind. Erst die Definition auf Makrozustandsebene macht Sinn. Dabei gehen Stephan Mertens und Heiko Bauke den Weg über die Entropie. Sie kamen dabei auf folgende Formel, welche die ’Entropie’ eines Zufallsgenerators wiederspiegelt.

$$H = -\sum_{\{m_j\}} P(m_1,...,m_{n+1}) \log_2 \frac{P(m_1,...,m_{n+1})}{P(m_1,...,m_n)}$$

Dabei ist $P(m_1,...,m_n)$ die Wahrscheinlichkeit, dass eine Folge von Makrozustand $m_1,...,m_n$ eintritt. Das Maximum der Entropie ergibt sich dabei als:

$$H = -\sum_j P(m_j) \log_2 P(m_j)$$

Dies ist gleich bedeutend damit, dass man keine Rückschlüsse aus den bisher angenommen Makrozuständen auf den als nächstes eingenommenen Makrozustand machen kann. Ein Zufallsgenerator, der diese Bedingung erfüllt, ist genau das was wir suchen. Betrachten wir nun den Zufallsgenerator

$$r_i = \alpha(r_{i-p} + r_{i-q}) \mod 1 \quad r_i, \alpha \in \Re$$

So stellen wir fest, dass die Entropie für den Fall von M gleichgewichteten Makrozuständen hauptsächlich von zwei Werten abhängt. Nämlich einmal von der Größe der Lücke zwischen p und q, und zum anderen von der Größe von α .

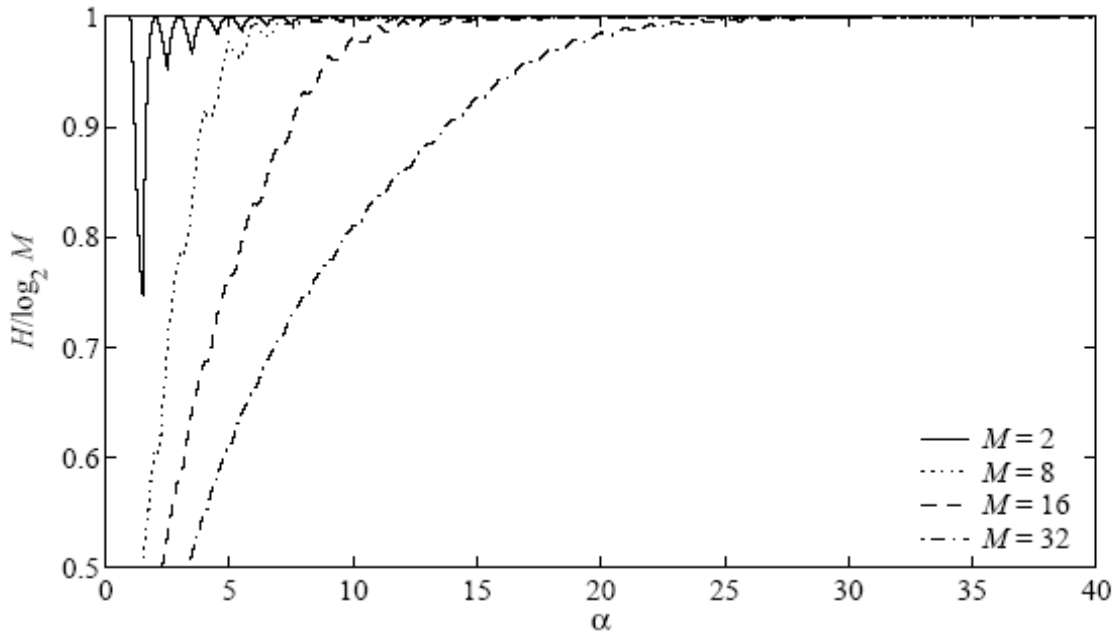

Fig. 10: Hier wurde über dem Parameter α die normierte Entropie aufgetragen. D.h. der Wert 1 auf der y Achse entspricht der maximalen Entropie, die für dieses System möglich ist. Aus [3]

Das interessante an dieser Theorie ist, dass man die Güte eines Zufallsgenerators bezüglich eines Problems nun relativ schnell bestimmen kann, in dem man einfach $H/\log_2 M$ berechnet. Weicht das Ergebnis von 1 ab, so ist der gewählte Zufallsgenerator für dieses Problem ungeeignet.

6. Zusammenfassung

Wenn für ein Problem einen Zufallsgenerator benötigt wird, sollte man sich immer Gedanken über den zu verwendenden Zufallsgenerator machen und nicht einfach nur den implementierten Zufallsgenerator verwenden. Dieser reicht zwar für viele Anwendungen aus, doch besonders bei Simulationen sollte man den verwendeten Generator immer kritisch betrachten. Leicht können durch einen unpassenden Generator Zusammenhänge entstehen, die in der Natur nicht vorkommen. Außerdem wurden Methoden der Anwendung von Zufallszahlengeneratoren gezeigt, die man im Hinterkopf behalten sollte, denn die Monte-Carlo Integration ist kein schlechtes Mittel um komplizierte Integrale numerisch zu lösen. Besonders wenn auch noch die Verteilung der Zufallszahlen an das Problem angepasst ist.

7. Quellcode

Entnommen aus [1] Numerical Recipes in C, Kapitel 7.2

7.1 Minimal Standard Generator

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876
float ran0(long *idum)
```

“Minimal” random number generator of Park and Miller. Returns a uniform random deviate between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK) to initialize the sequence; idum must not be altered between calls for successive deviates in a sequence.

```
{
long k;
float ans;
*idum ^= MASK;                  XORing with MASK allows use of zero and other
k=(*idum)/IQ;                   simple bit patterns for idum.
*idum=IA*(*idum-k*IQ)-IR*k;     Compute idum=(IA*idum) % IM without over-
if (*idum < 0) *idum += IM;     flows by Schrage’s method.
ans=AM*(*idum);                 Convert idum to a floating result.
*idum ^= MASK;                  Unmask before return.
return ans;
}
```

7.2 Gemischter Zufallsgenerator

```
#define IA 16807
#define IM 2147483647
```

```
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
float ran1(long *idum)
```

“Minimal” random number generator of Park and Miller with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with idum a negative integer to initialize; thereafter, do not alter idum between successive deviates in a sequence. RNMX should approximate the largest floating value that is less than 1.

```
{
int j;
long k;
static long iy=0;
static long iv[NTAB];
float temp;
if (*idum <= 0 || !iy) {                Initialize.
if (-(*idum) < 1) *idum=1;              Be sure to prevent idum = 0.
else *idum = -(*idum);
for (j=NTAB+7;j>=0;j--) {               Load the shuffle table (after 8 warm-ups).
k=(*idum)/IQ;
*idum=IA*(*idum-k*IQ)-IR*k;
if (*idum < 0) *idum += IM;
if (j < NTAB) iv[j] = *idum;
}
iy=iv[0];
}
k=(*idum)/IQ;                           Start here when not initializing.
*idum=IA*(*idum-k*IQ)-IR*k;             Compute idum=(IA*idum) % IM without over-
if (*idum < 0) *idum += IM;             flows by Schrage’s method.
j=iy/NDIV;                              Will be in the range 0..NTAB-1.
iy=iv[j];                               Output previously stored value and refill the
iv[j] = *idum;                          shuffle table.
if ((temp=AM*iy) > RNMX)                Because users don’t expect endpoint values.
{
return RNMX;
}
else return temp;
}
```

7.3 Kombinierter Zufallsgenerator

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
```

```
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
float ran2(long *idum)
```

Long period (> 2 × 1018) random number generator of L'Ecuyer with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with idum a negative integer to initialize; thereafter, do not alter idum between successive deviates in a sequence. RNMX should approximate the largest floating value that is less than 1.

```
{
int j;
long k;
static long idum2=123456789;
static long iy=0;
static long iv[NTAB];
float temp;
if (*idum <= 0) {                    Initialize.
if (-(*idum) < 1) *idum=1;           Be sure to prevent idum = 0.
else *idum = -(*idum);
idum2=(*idum);
for (j=NTAB+7;j>=0;j--) {            Load the shuffle table (after 8 warm-ups).
k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += IM1;
if (j < NTAB) iv[j] = *idum;
}
iy=iv[0];
}
k=(*idum)/IQ1;                       Start here when not initializing.
*idum=IA1*(*idum-k*IQ1)-k*IR1;       Compute idum=(IA1*idum) % IM1 without
if (*idum < 0) *idum += IM1;         overflows by Schrage's method.
k=idum2/IQ2;
idum2=IA2*(idum2-k*IQ2)-k*IR2;       Compute idum2=(IA2*idum) % IM2 likewise.
if (idum2 < 0) idum2 += IM2;
j=iy/NDIV;                           Will be in the range 0..NTAB-1.
iy=iv[j]-idum2;                      Here idum is shuffled, idum and idum2 are
iv[j] = *idum;                       combined to generate output.
if (iy < 1) iy += IMM1;
if ((temp=AM*iy) > RNMX)             Because users don't expect endpoint values.
{
return RNMX;
}
else return temp;
}
```

8. Quellenangabe

[1] **Numerical Recipes in C,** William H. Press, Saul A. Teukolsky, William T. Vetterling et al. erschienen 1988-1992 bei Cambridge University Press, insbesondere Kapitel 7.1-4 und 7.6

[2] **Entropy of Pseudo Random Number Generators,** Stephan Mertens und Heiko Bauke erschienen in Phys. Rev. E **69**, 055702(R) (2004)

[3] **Monte Carlo Simulations: Hidden Errors from 'Good' Random Number Generators,** Alan M. Ferrenberg *et al.* erschienen in Phys. Rev. Lett. **69**, 3382 (1992)