Einführung

Bevor wir uns mit dem Thema Zufallszahlengeneratoren befassen, definieren wir zunächst einmal den Begriff Zufall. Was ist der Zufall eigentlich und wann kann man tatsächlich von Zufall sprechen? Eine zufällige Zahl ist zunächst einmal das Ergebnis von speziellen Zufallsexperimenten. Umgangssprachlich wird der Begriff Zufall - oder „reiner“ Zufall - verwendet, wenn Ereignisse scheinbar nicht kausal erklärbar sind. In unserem realen Umfeld erleben wir Phänomene die in unseren Augen zufällig erscheinen und daher nicht vorhersehbar sind. Das Wetter ist in der Regel ein chaotisches System, in welchem der Zufall regiert und daher sind Wettervorhersagen, die über eine bestimmte Zeitspanne hinausgehen ohne praktischen Nutzen. Streng gesehen bezeichnen wir viele Vorgänge als chaotisch und vom Zufall bestimmt, obwohl diese in Wahrheit durchaus deterministisch geprägt sein können. So lässt sich ein System *theoretisch* exakt beschreiben wenn die Anfangsbedingungen und physikalischen Gesetzmäßigkeiten des Systems dem Beobachter vollständig bekannt sind.

Laplacescher Dämon

Die Überlegung das die Welt deterministisch sei, hat in der Geschichte bereits viele Menschen beschäftigt, darunter den berühmten Mathematiker Pierre-Simon Laplace , Schöpfer der Integraltransformation, auch Laplace-Transformation genannt. Laplace stellte die Theorie des Laplacesche'n Dämon auf der die erkenntnis- und wissenschaftstheoretische Auffassung bezeichnet, dergemäß es möglich ist, unter der Kenntnis sämtlicher Naturgesetze und aller Initialbedingungen jeden vergangenen und jeden zukünftigen Zustand zu berechnen.

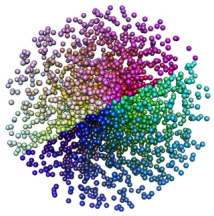
Aus heutiger Sicht wissen wir das der Laplacescher Dämon ein Phantasiegebilde war, da die Heisenbergsche Unschärferelation eine vollständige Kenntnis des Zustands eines Systems unmöglich macht und ein System deshalb in letzter Konsquenz für den Beobachter in seiner Gesamtheit nicht erfassbar bleibt. Zudem haben insbesondere die Phänomene der Quantenphysik Bereiche aufgezeigt, in denen es durchaus „reine“ Zufälle geben könnte. Die Gleichungen der Quantenmechanik nutzen den Zufall allerdings ohne die Existenz eines absoluten Zufalls zu beweisen bzw. diesen in letzter Konsequenz vorauszusetzen. Zudem ist es auch in der Quantenmechanik nicht erlaubt, Ereignisse in der Vergangenheit zu verändern, sodass in jedem Fall das Kausalprinzip vorherrscht.

Wir gelangen an einen Grenzpunkt an dem wir festhalten müssen das Zufall und Determinismus fließend ineinander übergehen. Für uns ist im Prinzip nur wichtig ob wir einen **q uantitativen Zufall**
erreichen können. Das bedeutet ob wir Zahlen generieren können die für uns prinzipiell ausreichend zufällig sind.

Zufallszahlengenerator

Für die Erzeugung von Zufallszahlen gibt es verschiedene Verfahren. Diese werden als Zufallszahlengeneratoren bezeichnet. Der Zufallszahlengenerator, gelegentlich auch einfach nur Zufallsgenerator oder schlicht Generator genannt, bezeichnet ein Verfahren, das eine Folge von Zufallszahlen erzeugt. Der Bereich, aus dem die Zufallszahlen erzeugt werden, hängt dabei vom Zufallszahlengenerator ab. Es kann beispielsweise die Menge aller 32-Bit-Zahlen oder auch die Menge der reellen Zahlen im Intervall [0,1] sein. Ein entscheidendes Kriterium für Zufallszahlen ist, ob das Ergebnis der Generierung als unabhängig von früheren Ergebnissen angesehen werden kann oder nicht. Ein guter Zufallszahlengenerator muss dieses wichtige Kriterium erfüllen. Diese Bedingung mag auf den ersten Blick nicht sonderlich aufregend sein, ist für die Qualität der produzierten Zufallszahlen allerdings von entscheidender Bedeutung.

Von einem Zufallszahlengenerator wird darüberhinaus erwartet, dass er gleichverteilte Werte aus dem jeweiligen Bereich erzeugt. Diese Gleichverteilung ist ein entscheidendes Kriterium für die Güte des Zufallszahlengenerators. Wir betrachten dazu einmal die Funktion rand() aus Microsoft Excel
die Zufallszahlen berechnet und stellen diese in einem dreidimensionalen Raum grafisch dar.



Wir können erkennen das die Anordnung der Ergebnisse teilweise zentrisch ist, allerdings in der Gesamtheit die komplette Sphäre einnehmen. Das bedeutet die Funktion liefert gute, gleichverteilte Werte über das gesamte Spektrum gesehen.

Nun liefern Funktionen, wie die aus Microsoft Excel oder der C-Standardbibliothek bekannten Funktion rand(), in Wahrheit keine tatsächlichen Zufallszahlen, sondern **pseudozufällige Zahlen**. Als Pseudozufall wird bezeichnet, was zufällig erscheint, in Wirklichkeit jedoch berechenbar ist. Ein Generator auf dem Computer ist daher in der Regel ein Pseudozufallszahlengenerator (engl. PRNG, pseudo random number generator) und kein echter Zufallszahlengenerator. PRNG erzeugen eine Zahlenfolge, die zwar zufällig aussieht, es aber nicht wirklich ist, da sie durch einen deterministischen
Algorithmus
berechnet wird. Bei jedem Start der Zufallszahlenberechnung mit gleichem Startwert, der so

genannten Saat (engl. seed), wird die gleiche pseudozufällige Zahlenfolge erzeugt.

Dies lässt sich in der Sprache C sehr gut bei der Funktion rand() beobachten und ist ein häufig gemachter Anfängerfehler. Oft wird nämlich für die Initialisierung mithilfe von srand() ein gleichbleibender Wert herangezogen, wodurch die Zufallszahl dieselbe bleibt. Abhilfe schafft hier beispielsweise die Initialisierung mit der Funktion time() als Seed. Letztenendes bleibt das Ergebnis, nämlich die Zufallszahl, in Wirklichkeit aber eine Pseudozufallszahl.

[code xml:lang="c"]/* Es werden 10 Zufallszahlen zwischen 0.0 und 1.0 ausgegeben. */ #include #include int rand(void); int main() { int i; srand((unsigned)time(NULL)); // Mit aktuellen Zeit initialisieren. for(i = 1; i 0 sind zugleich spezielle Kennwerte dieser allgemeinen Normalverteilung: μ ist der Mittel- oder Erwartungswert, σ die Standardabweichung und σ² die Varianz der normalverteilten stetigen Zufallsvariablen X. Die Verteilungsfunktion der allgemeinen Normalverteilung besitzt dabei die folgende Integraldarstellung:

$$F(x) = P(-\infty < X \leq x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2} dt$$

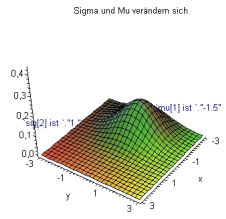
Der Verlauf ist im folgenden Bild wiedergegeben.

[thumb src="images/tutorials/cpp/DistNormalProb.png" arg=";;;F(x) der Gaußschen Normalverteilung"]Verteilungsfunktion[/thumb]

Anhand der grafischen Darstellung im vorletzten Bild kann man erkennen das die Funktion f(x) einer Glocke ähnelt. Daher der Name „Gaußsche Glockenkurve“. Die variablen Parameter μ und σ beeinflussen das Aussehen dieser Glockenkurve. Während μ das Maximum festlegt bestimmt der zweite Parameter σ Breite und Höhe der Kurve. Je *kleiner* die Standardabweichung σ ist, desto
höher
liegt das Maximum und umso steiler fällt die Dichtekurve nach beiden Seiten hin ab. Unser nächstes Bild zeigt drei verschiedene Parameter. Es wird bei der Darstellung sehr oft das Symbol N(μ; σ) verwendet. Bei einer der drei Kurven handelt es sich um eine besondere Normalverteilung, nämlich der Standardnormalverteilung.

[thumb src="images/tutorials/cpp/distri_param.png" arg=";;;Dichten von normalverteilten Zufallsgrößen mit unterschiedlichen Parametern"]Normalverteilungen[/thumb]

Die Abhängigkeit der Funktion von den Parametern μ und σ wird besonders deutlich, wenn man eine mehrdimensionale Verallgemeinerung der Normalverteilung vornimmt. Die so genannte multivariate Normalverteilung ist eine gemeinsame Wahrscheinlichkeitsverteilung mehrerer Zufallsvariablen und wird deshalb auch mehrdimensionale Verteilung genannt.



Standardnormalverteilung

Die behandelte allgemeine Gaußsche Normalverteilung lässt sich mit den speziellen Parameterwerten μ = 0 und σ = 1 auf die so genannte Standardnormalverteilung N(0; 1) zurückführen. Der Übergang wird durch Substitution erreicht. Die Fläche der Dichtefunktion nimmt dabei den Wert 1 an.

Die Dichtefunktion der Standardnormalverteilung lautet, wie folgt:

$$\varphi(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}$$

Die zugehörige Verteilungsfunktion lautet:

$$\phi(u) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{u} e^{-\frac{1}{2}t^2} dt$$

Das Integral hat keine elementare Lösung, aber es existieren viele gute Approximationen die auf numerische Integration zurückgreifen. Der meistverwendete Algorithmus ist Algorithmus 26.2.17 im Nachschlagewerk von Abromowitz und Stegun, Handbuch der Mathematischen Funktionen. Er hat einen maximalen absoluten Fehler von 7.5e^-8. Die Implementierung ist in der nächsten Zeile dargestellt.

[code xml:lang="cpp"]double N(const double x) { const double b1 = 0.319381530; const double b2 = -0.356563782; const double b3 = 1.781477937; const double b4 = -1.821255978; const double b5 = 1.330274429; const double p = 0.2316419; const double c = 0.39894228; if(x >= 0.0) { double t = 1.0 / (1.0 + p * x); return (1.0 - c * exp(-x * x / 2.0) * t * (t *(t * (t * (t * b5 + b4) + b3) + b2) + b1)); } else { double t = 1.0 / (1.0 - p * x); return (c * exp(-x * x / 2.0) * t * (t *(t * (t * (t * b5 + b4) + b3) + b2) + b1)); } }[/code]

Bevor wir uns mit einer vollständigen Implementierung der Standardnormalverteilung in C++ befassen, möchten wir anhand eines Beispiels, ein klassisches Einsatzgebiet der Gaußschen Normalverteilung aufzeigen. Dazu eignet sich die Betrachtung der Auswertung eines Intelligenztests sehr gut.

Wenn man einen Intelligenztest an einer repräsentativen Stichprobe normiert, unterstellt man häufig eine Normalverteilung der Testpunkte, die die Untersuchungspersonen bei dem Test erzielen. Die Testrohwerte werden für jede Altersgruppe so transformiert, dass sich ein durchschnittlicher Punktwert von 100 und eine Standardabweichung von 15 ergibt. Nach dieser Normierung kann der Test für diagnostische Zwecke verwendet werden. Wenn beispielsweise eine Person, deren Intelligenz diagnostiziert werden soll, insgesamt 113 Testpunkte erzielt, betrachtet man die Wahrscheinlichkeit, ein solches Testergebnis zu erreichen, um zu beurteilen, wie überdurchschnittlich intelligent die Person ist. In dem Beispiel geht es also um die Wahrscheinlichkeit, maximal einen Punktwert von X = 113 zu erreichen, wenn die Verteilung aller Punktwerte einer Normalverteilung folgt, durchschnittlich einen Wert von $\mu = 100$ Punkten aufweist und die Varianz aller Punktwerte $\sigma^2 = 15^2 = 225$ beträgt.

In unserem Beispiel erzeugen wir Zufallszahlen entsprechend der Standardnormalverteilung, die innerhalb der Standardabweichung $\sigma = 1$ liegen. Wir greifen dazu auf die seit 1958 bekannte Box-Muller Transformation zurück.

Für den Fall das wir eine Gleichung haben, die unsere gewünschte Verteilungsfunktion beschreibt, ist es möglich einen mathematischen Trick anzuwenden, der auf den fundamentalen Transformationsgesetzen für Wahrscheinlichkeiten beruht, um an eine transformierte Funktion für die Verteilung zu gelangen. Die Transformation nimmt zufällige Variablen aus einer beliebigen Verteilung entgegen und generiert zufällige Variablen in einer neuen Verteilungsfunktion. Das erlaubt uns einheitlich verteilte Werte in einen neuen Satz normalverteilter Zufallszahlen zu transformieren.

Die gängigste Form der Transformation sieht wie folgt aus:

y1 = sqrt(-2 * ln(r1)) * cos(2 * PI * r2) y2 = sqrt(-2 * ln(r1)) * sin(2 * PI * r2)

Wir starten mit zwei unabhängigen Zufallszahlen, r1 und r2, welche von einer einheitlichen Verteilungsfunktion stammen (zwischen 0 und 1). Die dafür notwendige C-Funktion haben wir bereits auf der ersten Seite kennengelernt. Anschließend wenden wir die obige Transformation an um zwei unabhängige Zufallszahlen zu erzeugen die innerhalb der Standardnormalverteilung liegen, also die speziellen Parameterwerte $\mu = 0$ und $\sigma = 1$ aufweisen.

[code xml:lang="cpp"]#include #include #include #include #include #include double const PI = 4 * std::atan(1.0); // Functor for normal distribution class normal_distribution { public: normal_distribution(double m, double s) : mu(m), sigma(s) { } double operator()() { // Generates a equal distribution in the range [0, 1] double r1 =

(std::rand() + 1.0) / (RAND_MAX + 1.0); double r2 = (std::rand() + 1.0) / (RAND_MAX + 1.0); return mu + sigma * std::sqrt(-2 * std::log(r1)) * std::cos(2 * PI * r2); } private: double mu, sigma; }; int main() { double samples[100]; srand((unsigned)time(NULL)); // Assigns the normal distributed values generated by a function object to a // specified number of element in a range and returns to the position // one past the last assigned value. std::generate_n(samples, 100, normal_distribution(1.0, 0.5)); std::copy(samples, samples + 100, std::ostream_iterator(std::cout, "n")); return 0; }[/code]

Diese besondere Transformation hat aber zwei entscheidende Nachteile.

- Sie ist langsam, weil sie mehrmals trigonometrische Funktionen aufruft.
- Sie kann numerische Stabilitätsprobleme aufweisen, wenn r1 sehr nahe an der 0 liegt.

Das sind ernsthafte Probleme falls sie stochastische Modellierungen durchführen oder Millionen von Zufallszahlen generieren wollen.

Die Polarform der Box-Muller Transformation ist schneller und robuster. Der Algorithmus ist folgendermaßen definiert:

[code xml:lang="cpp"]#include extern float ranf(); // ranf() is uniform in 0..1 // mean m, standard deviation s float box_muller(float m, float s) { float x1, x2, w, y1; static float y2; static int use_last = 0; if (use_last) { y1 = y2; use_last = 0; } else { do { x1 = 2.0 * ranf() - 1.0; x2 = 2.0 * ranf() - 1.0; w = x1 * x1 + x2 * x2; } while (w >= 1.0); w = sqrt((-2.0 * log(w)) / w); y1 = x1 * w; y2 = x2 * w; use_last = 1; } return m + y1 * s; }[/code]

Die Funktion ranf() steht provisorisch für einheitlich verteilte Zufallszahlen im Bereich zwischen 0 und 1.

Es existieren neben den benannten Algorithmus noch weitere, wie der von Graeme West. Dabei handelt es sich um einen hochpräzisen (double) Algorithmus der im Artikel Better Approximations to Cumulative Normal Fuctions
näher beschrieben wird. Der Algorithmus basiert auf Hart's Algorithmus 5666 aus dem Jahre 1968. Wir wollen nicht näher auf diesen Algorithmus eingehen, da dieser bereits ausführlich in dem Dokument beschrieben wird.

Die Gammaverteilung

Wir haben uns nun mit der grundlegensten kontinuierlichen Wahrscheinlichkeitsverteilung, der Gaußschen Normalverteilung befasst und Methoden kennengelernt statistisch normalverteilte Zufallszahlen mit dem Rechner zu generieren. Die Art und Weise, wie unsere Zufallszahlen generiert worden sind, hat uns dabei allerdings nicht interessiert. Genauer gesagt haben wir in fast allen Fällen die allseits bekannte Funktion rand() für die Erzeugung der Zahlen verwendet.

Bevor wir uns im konkreten Fall mit bekannten Algorithmen aus Pseudozufallszahlengeneratoren beschäftigen, möchten wir uns noch mit einer letzten, aber wichtigen Wahrscheinlichkeitsverteilung auseinander setzen, der Gammaverteilung. Die Gammaverteilung ist ebenfalls eine kontinuierliche Wahrscheinlichkeitsverteilung. Sie ist definiert über der Menge der positiven reellen Zahlen. Einerseits ist die Gammaverteilung eine direkte Verallgemeinerung der Exponentialverteilung und andererseits eine Verallgemeinerung der Erlang-Verteilung für nichtganzzahlige Parameter. Sie findet unter anderem Anwendung in der Warteschlangentheorie, um die Bedienzeiten oder Reparaturzeiten zu beschreiben und in der Versicherungsmathematik zur Modellierung kleinerer bis mittlerer Schäden.

Die allgemeine Dichtefunktion der Gammaverteilung kann in Bezug auf die Gammafunktion folgendermaßen ausgedrückt werden:

$$f(x) = \frac{\left(\frac{x-\mu}{\beta}\right)^{\alpha-1} \exp\left(-\frac{x-\mu}{\beta}\right)}{\beta\Gamma(\alpha)} \qquad x \geq \mu \qquad \alpha, \beta > 0$$

Der Parameter α bezeichnet den sogenannten Formparameter, μ die Lokalisierung und β den Skalenparameter. Die Gammaverteilung leitet sich aus der Gammafunktion ab, die wie folgt definiert ist:

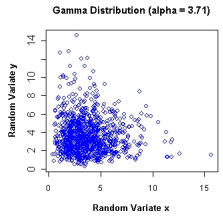
$$\Gamma(a) = \int_0^\infty t^{a-1} \mathrm{e}^{-t} \mathrm{d}t$$

Nachfolgend sehen sie ein Java-Applet das die Gammaverteilung in Abhängigkeit des Formparameters und des Skalenparameters darstellt. Dabei ist zu beachten das der Formparameter mit δ abgekürzt wurde und der Skalenparameter als Inverse λ = 1 / β definiert ist. Diese Art der Parametrisierung wird sehr oft verwendet.

Ähnlich zu der Standardnormalverteilung mit N(0; 1) sehen wir uns bei der Gammaverteilung den Fall μ = 0 und β = 1 etwas näher an. Wir erhalten mit diesen Parametern die folgende Dichtefunktion:

$$f(x) = \frac{x^{\alpha-1} \mathrm{e}^{-x}}{\Gamma(\alpha)} \qquad x \geq 0 \qquad \alpha > 0$$

Damit lässt sich ein Graph generieren, der die Verteilung von Zufallszahlen grafisch veranschaulicht. Dazu werden 1000 Zufallszahlen aus zwei unterschiedlichen Sequenzen generiert. Jedes Paar an Zahlen wird gegeneinander geplottet, um das Verhalten der Gammaverteilung bei diesem, nicht gleich verteilten, Zufallszahlengenerator zu demonstrieren.



Gamma-verteilte Zufallsvariablen eignen sich zur Modellierung von Prozessen, bei denen es um Wartezeiten zwischen ”zufälligen“ Ereignissen geht. In der Eisenbahnbetriebswissenschaft sind dies zum Beispiel Ankünfte der Züge an bestimmten Stellen im Eisenbahnnetz, oder aber die zeitliche Verteilung der Wunschtrassen bei der Trassenvergabe in der Fahrplanerstellung. Die weitgehend freie Wahl der Parameter bietet einen großen Spielraum bei der Beschreibung von stochastischen Vorgängen.

Wir wenden uns nun von kontinuierlichen Wahrscheinlichkeitsverteilungen für Zufallszahlen ab und gehen über zu den Algorithmen die von gängigen Zufallszahlengeneratoren verwendet werden um diese Zahlen zu generieren. Im nächsten Abschnitt gehen wir auf den Mersenne-Twister Pseudozufallszahlengenerator ein.

Mersenne Twister

Der Mersenne-Twister ist ein Pseudozufallszahlengenerator, der 1997 von Makoto Matsumoto und Takuji Nishimura entwickelt wurde. Er ermöglicht die schnelle Erzeugung hochwertiger Sequenzen von Pseudozufallszahlen und wurde extra darauf zugeschnitten, die Probleme älterer Algorithmen zu überwinden.

- Er hat die extrem lange Periode von $2^{19937}-1$ ($\approx 4{,}3\cdot10^{6001}$). Diese Periodenlänge erklärt auch den Namen des Algorithmus: Sie ist eine Mersenne-Primzahl, und einige Eigenschaften des Algorithmus resultieren daraus.

- Er liefert hochgradig gleichverteilte Sequenzen (bewiesene Gleichverteilung bis zur Dimension 623, siehe unten). Daraus folgt eine extrem kleine Korrelation zwischen aufeinanderfolgenden Wertefolgen der Ausgabesequenz.
- Er ist schneller als jeder andere bekannte (hinreichend gute) Algorithmus.

- Alle Bits der Ausgabesequenz sind für sich gleichverteilt.

Andererseits hat er den Nachteil, auf einer großen Datenmenge von 2,5 kB (624 Wörter mit je 32 Bits) zu arbeiten. Mit den heutigen Rechnerarchitekturen mit schnellem, aber relativ kleinem Cache und langsamerem Arbeitsspeicher kann sich daraus ein Geschwindigkeitsnachteil ergeben.

Der Mersenne-Twister ist ein verdrillter *Generalized Feedback Shift Register* Algorithmus, kurz (GFSR), von rationaler Normalenform (TGFSR(R)), mit Statusbit Reflection und Tempering, mit dem die Multiplikation einer Matrix T gemeint ist. Er wird charakterisiert durch die folgenden Einheiten:

- w: Wortgröße (in Anzahl an Bits)
- n: Grad der Rekursionen
- m: mittleres Wort, oder die Anzahl an parallelen Sequenzen, $1 \leq m \leq n$
- r: Trennpunkt eines Wortes, oder die Anzahl an Bits der niederen Bitmaske, $0 \leq r \leq w - 1$
- a: Koeffizienten der verdrillten Matrix in rationaler Normalenform
- b, c: TGFSR(R) Tempering Bitmasken
- s, t: TGFSR(R) Tempering Bit Shifts
- u, l: zusätzliche Mersenne Twister Tempering Bit Shifts

mit der Einschränkung das $2^{nw - r} - 1$ eine Mersenne Primzahl ist. Diese Wahl vereinfacht den Primitivitätstest und k-Verteilungstest, welche für die Parametersuche benötigt werden.

Für ein Wort x mit der Bitlänge w, wird es als rekursives Verhältnis, wie folgt ausgedrückt:

$$x_{k+n} := x_{k+m} \oplus ({x_k}^u \mid {x_{k+1}}^l)A \qquad k = 0, 1, \ldots$$

mit | als das bitweise ODER und ⊕ als das bitweise Exklusiv ODER (XOR), während $\boldsymbol{x}^u$, $\boldsymbol{x}^l$ eine Anwendung der höheren und niederen Bitmasken auf das x darstellen. Die Twist Transformation A ist in der rationalen Normalenform definiert.

$$A = R = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots a_0) \end{pmatrix}$$

mit I_{n-1} als die $(n - 1) \times (n - 1)$ Identitätsmatrix (und im Gegensatz zu der normalen Matrixmultiplikation, ersetzt das bitweise XOR die Addition).

Ähnlich wie der TGFSR(R), ist der Mersenne Twister in einer Kaskade geschalten mit einer Tempering Transformation um die reduzierte Dimensionsgröße der Gleichverteilung (Weylsche “Gleichverteilung von Zahlen mod 1”) zu kompensieren. Diese ist äquivalent zu der Transformation $A = R \rightarrow A = T^{-1}RT$, T umkehrbar. Das Tempering ist beim Mersenne Twister als

$$\begin{aligned} \boldsymbol{y} &:= \boldsymbol{x} \oplus (\boldsymbol{x} >> u) \\ \boldsymbol{y} &:= :\boldsymbol{y} \oplus ((\boldsymbol{y} << s) \,\&\, \boldsymbol{b}) \\ \boldsymbol{y} &:= :\boldsymbol{y} \oplus ((\boldsymbol{y} << t) \,\&\, \boldsymbol{c}) \\ \boldsymbol{z} &:= \boldsymbol{y} \oplus (\boldsymbol{y} >> l) \end{aligned}$$

definiert.

Zusammenfassend kann man den Mersenne Twister in 6 einfache Einzelschritte unterteilen. Zunächst wird eine Maske für die höheren und niederen Bits generiert. Anschließend wird ein Array x mit einem Seed initialisiert. Danach werden die höheren Bits y von x[i] mit den niederen von x[i+1] verknüpft. Nun wird y mit der Matrix A multipliziert. A wird dabei vorsichtig gewählt, damit es leicht mit Rechtsshifts und EXOR berechnet werden kann. Nun kommt das Tempering ins Spiel, bei dem mit T multipliziert wird um eine bessere Weylsche Gleichverteilung und Genauigkeit zu erreichen. Im letzten Schritt wird i um 1 inkrementiert und der Vorgang wiederholt.

Der Algorithmus ist im Detail auf der offiziellen Seite beschrieben. Dort findet sich auch der originale, in C implementierte Quellcode.

Die Kombination der gewonnenen Erkenntnisse

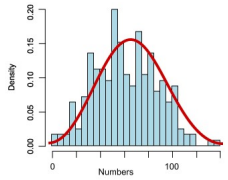
Um einen zuverlässigen PRNG zu schreiben sind die bisher beschriebenen Punkte von großer Bedeutung. Aus der Kombination von Wissen über die Wahrscheinlichkeitsverteilungen und moderne Algorithmen zur Generierung von Zufallszahlen, lässt sich das nun bewerkstelligen. Die gezeigten Methoden zur Berechnung statistisch verteilter Zufallszahlen werden bereits in vielen Bibliotheken vorimplementiert. So verfügt C++ Boost über eine eigene ” Boost Random Number Library “, die diverse Zufallszahlengeneratoren und Verteilungen zur Verfügung stellt.

Unter Nutzung dieser Bibliothek lässt sich beispielsweise ein Mersenne Twister Generator schreiben, der mithilfe der Gaußschen Normalverteilung Zufallszahlen erzeugt. Die folgende

Funktion mit dem Namen *SampleNormal* gibt Proben von Zufallszahlen aus einer Normalenverteilung zurück. Der Mittel- oder Erwartungswert und die Standardabweichung der Verteilung werden als Parameter an die Funktion übergeben.

[code xml:lang="cpp"]#include #include using namespace boost; double SampleNormal(double mean, double sigma) { // Create a Mersenne twister random number generator // that is seeded once with #seconds since 1970 static mt19937 rng(static_cast(std::time(0))); // select Gaussian probability distribution normal_distribution norm_dist(mean, sigma); // bind random number generator to distribution, forming a function variate_generator normal_sampler(rng, norm_dist); // sample from the distribution return normal_sampler(); }[/code]

Aus den Proben lässt sich ein sogenanntes Histogramm generieren, mit dem die Häufigkeitsverteilung der Zufallszahlen anschaulich graphisch dargestellt werden kann. Das folgende Histogramm zeigt sehr unregelmäßige Proben die eine Normalenverteilung aufweisen.



Bei einem echten Zufallszahlengenerator wären die hellblau gefärbten Proben über das gesamte Spektrum absolut gleichverteilt und gleich hoch. In unserem Fall wird die Verteilung der Werte, aufgrund der gewählten Verteilungsfunktion, sinngemäß ebenfalls der Gaußschen Glockenkurve nahe kommen.
Jeder PRNG erfüllt in keinem Fall das Kriterium eines einheitlichen Histogramms. Er kann diesem Kriterium zwar nahe kommen, allerdings wird er es nie zu 100 % erfüllen können. Nichts desto trotz liefert der Mersenne Twister sehr gleichverteilte Werte über das gesamte Spektrum.

Weitere Generatoren

Neben dem eingehend beschriebenen Mersenne Twister Zufallszahlengenerator existieren auch noch weitere bekannte Generatoren, die in der Praxis oft zum Einsatz kommen. Dazu zählen der Lagged Fibonacci Generator (LFG), der auf Fibonacci Sequenzen basiert, der Linear Congruential Generator, der zu den ältesten und bekanntesten Zufallszahlengeneratoren gehört bis hin zum alten Blum Blum Shub (BBS) Algorithmus, der sich heute aufgrund seiner Trägheit kaum noch für Simulationen, sondern hauptsächlich für Verschlüsselungen eignet.

In einem letzten Codebeispiel generieren wir einen Lagged Fibonacci Generator und wenden eine Normalenverteilung auf die Zufallszahlen an.

[code xml:lang="cpp"]#include #include #include #include using namespace boost; int main() { const double mean = 10.0; const double sigma = 1.0; normal_distribution norm_dist(mean, sigma); lagged_fibonacci44497 engine; // Make a histogram const int size = static_cast(mean) * 2; std::vector histo(size, 0); for(int i = 0; i != 1000000; ++i) { const double value = norm_dist.operator()((engine)); int index = static_cast(value); index = index size - 1 ? size - 1 : index; ++histo[index]; } // Output histogram for(int i = 0; i != size; ++i) { std::cout