1 Zufallszahlengenerator

Zufällig generierte Zahlen sind ein wichtiges und mächtiges Werkzeug in der numerischen Physik. Sie finden beispielsweise beim Integrieren von Funktionen mit mehreren Variablen mit sogenannten Monte Carlo Verfahren Anwendung.

Allerdings sind die am Computer genutzten Zufallszahlen nicht tatsächlich zufällig, sondern sie werden durch einen deterministischen Algorithmus erzeugt. Deshalb spricht man von Pseudozufallszahlen. In dieser Aufgabe werden wir einen Pseudozufallszahlengenerator schreiben. Dabei wird eine Reihe von Pseudozufallszahlen $I_0, I_1, I_2, \ldots$ erzeugt, die dann möglichst ähnlich zu einer Reihe echter Zufallszahlen ist. Die Reihe wird beispielsweise mit folgender Iteration erzeugt:

$$I_{j+1} = aI_j \,(\text{mod } m)\ , \quad j = 0, 1, 2, \ldots\ . \tag{1}$$

Diesen Instruktion erstellt eine neue Zufallszahl (I_{j+1}) aus der vorangegangenen Zahl (I_j). Die Qualität der Zufallszahlen hängt von den Eingangsparametern (a, m) ab. Ein guter Zufallszahlengenerator hat große Perioden, dass heißt, es braucht sehr viele Iterationen in j bis eine Zufallszahl I_k erneut auftritt. Die Periode ist deswegen so wichtig, weil sich die Reihe von da an exakt wiederholt. Park und Miller haben die folgenden Parameter für a und m gewählt:

$$a = 16807\,, \quad m = 2^{31} - 1 = 2147483647\,. \tag{2}$$

Leider ist die direkte Implementation dieses Zufallszahlengenerators mit diesen Parametern in `C` mit dem Typ `int` nicht möglich. Der Grund ist, dass wir keine Zahlen größer als m in einem `int` speichern können. (Mit Hilfe von `unsigned long int` kann man auf modernen Rechnern natürlich schon größere Zahlen darstellen, als m) Glücklicherweise gibt es eine Möglichkeit das Problem algorithmisch umzugehen. Wir faktorisieren m wie folgt:

$$m = aq + r\,; \quad r = m \,(\text{mod } a)\ ; \quad q = [m/a]\ . \tag{3}$$

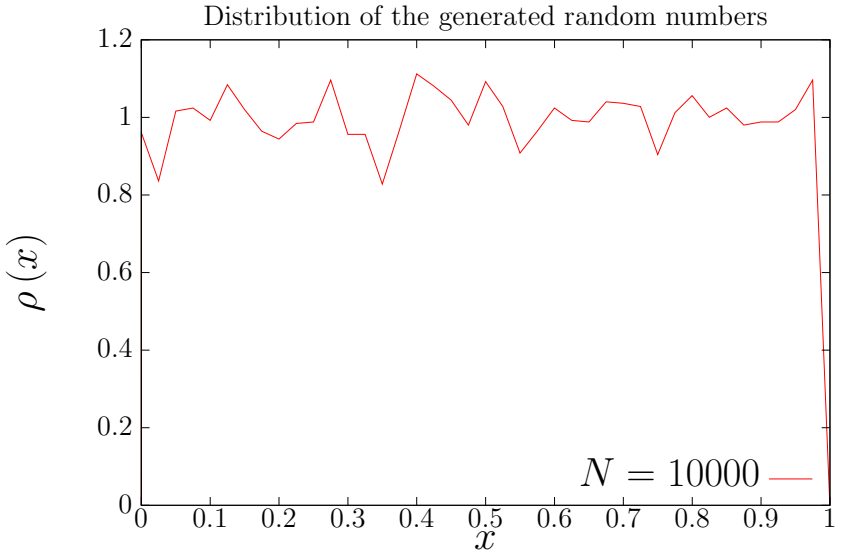
Damit können wir die Gleichung (1) auch mit (q, r) darstellen:

$$\begin{aligned} aI_j \,(\text{mod } m) &= (a\,(I_j \text{mod } q) - r\,[I_j/q] + m) \,(\text{mod } m) \\ &= \begin{cases} a\,(I_j \text{mod } q) - r\,[I_j/q] & \text{falls } > 0\,, \\ a\,(I_j \text{mod } q) - r\,[I_j/q] + m & \text{sonst}\,. \end{cases} \end{aligned} \tag{4}$$

Hierbei ist $r = 2836$ und $q = 127773$.

Implementieren Sie den Zufallszahlengenerator nach Gleichung 4. Ändern Sie den Algorithmus so ab, dass er gleichverteilte Fließkommazahlen u_j zwischen 0 und 1 erzeugt. Um das Programm auf Korrektheit zu überprüfen, können wir beispielsweise die Verteilung der Zufallszahlen testen. Da der Algorithmus gleichverteilte Zufallszahlen erzeugen soll, muss im Mittel jede reelle Zahl zwischen 0 und 1 gleichoft vorkommen. Um das zu überprüfen, kann man ein Histogramm der erzeugten Zufallszahlen erzeugen. Dafür unterteilt man das Interval $[0, 1]$ in n Abschnitte der Länge $\Delta = 1/n$. Dann zählt man, wir viele Zufallszahlen C_i im Interval $[x_i, x_i + \Delta]$ liegen, mit $x_i = i\Delta$ und $i = 0, \ldots, n-1$. Schließlich trägt man die Dichte $\rho(x_i) = C_i/N$ gegen x_i auf, wobei N die Gesamtzahl der gezogenen Zufallszahlen ist.

Bespielsweise könnte man das Ergebnis graphisch wie folgt darstellen:



Welchen Wert erwarten Sie für den Mittelwert

$$\bar{x} = \frac{1}{N} \sum_j u_j$$

Ihrer Zufallszahlen? Passt Ihre Erwartung zu Ihrer Implementation?

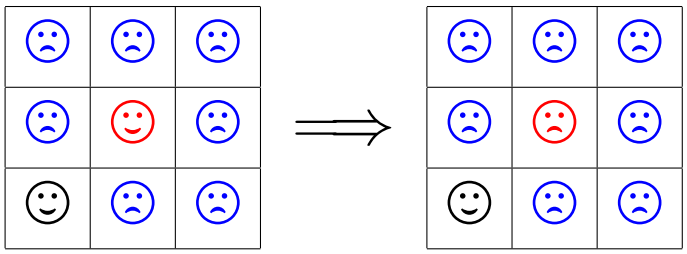
2 Conway's Spiel des Lebens

Betrachten wir ein „Universum", dass aus einem zweidimensionalen Gitter besteht. Jede Gitterzelle kann zwei Zustände einnehmen:

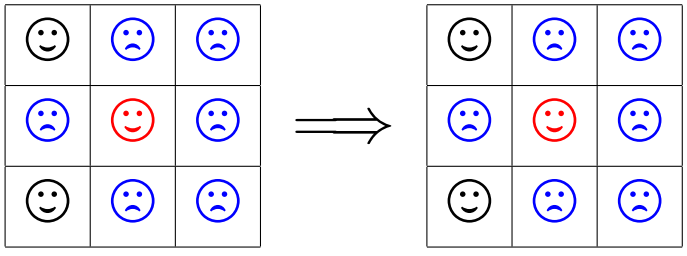
- lebendig,
- tot.

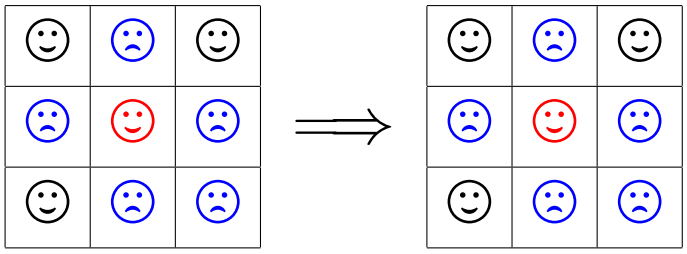
Im Spiel entwickelt sich das Universum nach bestimmten Regeln in diskreten Zeitschritten. Der Satz von Regeln R sieht wie folgt aus:

a) Eine lebendige Zelle stirbt, wenn sie weniger als zwei lebendige Nachbarzellen hat. (Einsamkeit)

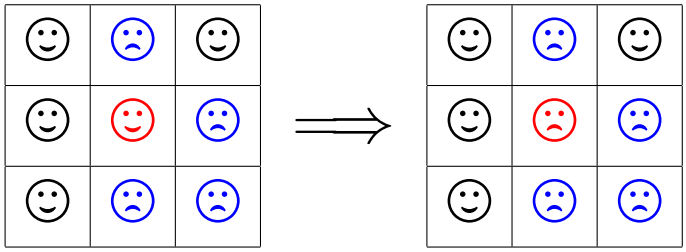


b) Eine lebendige Zelle mit zwei oder drei lebendigen Nachbarn lebt weiter.

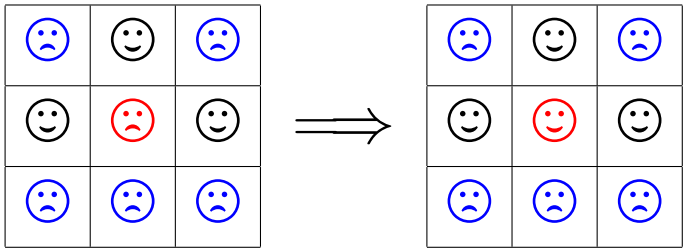




c) Eine lebendige Zelle mit mehr als drei lebenden Nachbarzellen stirbt. (Überbevölkerung)



d) Eine tote Zelle wird wiederbelebt, wenn sie genau drei lebende Nachbarzellen hat.



Dabei wird davon ausgegangen, dass diese Regeln auf alle Zellen gleichzeitig angewendet werden. Es gibt also einen Zustand des Universums $Z(\tau)$ zum Zeitpunkt τ. Dann wird jede Zelle nach obigen Regeln weiterentwickelt basierend auf dem Zustand $Z(\tau)$. Wir können also auch schreiben, dass die Entwicklung nach den Regeln $R(Z(\tau))$ stattfindet. Der neue Zustand ist dann $Z(\tau + 1)$. Der entsprechende Pseudocode könnte wie folgt aussehen

Algorithmus 1 Entwicklung $Z(\tau)$

Input: $Z(\tau)$
Output: $Z(\tau + 1)$
1: **for all** $z_i \in Z(\tau)$ **do**
2: **set** $z_i(\tau + 1) \xleftarrow{R(Z(\tau))} z_i(\tau)$
3: **end for**
4: **return** $Z(\tau + 1)$

Aufgabe ist ein Programm zu schreiben, das dieses Spiel implementiert. Jede Zelle kann zwei Zustände haben, die wir auf 0 (tot) und 1 lebendig

abbilden wollen. Es gilt also $z \in \{0,1\}\ \forall z \in Z$. Das zweidimensionale Gitter soll quadratisch sein und $L \times L$ Zellen z haben. D.h., insgesamt gibt es Zellen

$$z(x,y;\tau) = \begin{cases} 1 & \text{Zelle ist lebendig} \\ 0 & \text{Zelle ist tot} \end{cases}, \quad (x \in [0,1,\ldots L{-}1], y \in [0,1,\ldots L{-}1])\,. \tag{5}$$

Obiger Pseudocode sieht damit wie folgt aus

Algorithmus 2 Entwicklung von $Z(\tau)$ auf 2D Gitter

Input: $Z(\tau)$
Output: $Z(\tau+1)$

```
1: for all x ∈ 0, 1, ... L − 1 do
2:    for all y ∈ 0, 1, ... L − 1 do
3:       set z(x, y; τ + 1) ←R(Z(τ))— z(x, y; τ)
4:    end for
5: end for
6: return Z(τ + 1)
```

Es bietet sich an, das Gitter im Programm als ein zweidimensionales Array darzustellen. Der Datentyp des Arrays sollte es erlauben, die zwei Zustände darzustellen. Die Größe des Arrays ist durch L bestimmt. Wir haben zwar noch nicht über zweidimensionale Arrays gesprochen, trotzdem können wir diese Aufgabe erledigen. Wir beschränken uns auch zunächst auf *statische* arrays. Ein zweidimensionale $L \times L$ Array kann durch folgende Abbildung g auf ein eindimensionales Array der Größe L^2 abgebildet werden:

$$g(x,y) \;=\; xL + y\,, \qquad (x \in [0,1,\ldots L-1], y \in [0,1,\ldots L-1])\,. \tag{6}$$

Wir müssen noch sagen, wie wir am Rand des Gitters verfahren wollen. Hier werden wir periodische Randbedingungen annehmen, also

$$z(x+L,y) = z(x,y)\,, \quad (x \in [0,1,\ldots L-1], y \in [0,1,\ldots L-1]) \tag{7}$$

und

$$z(x,y+L) = z(x,y)\,, \quad (x \in [0,1,\ldots L-1], y \in [0,1,\ldots L-1])\,. \tag{8}$$

Im Quelltext benötigt man entsprechend zwei Arrays gleicher Größe für Zeitschritte τ und $\tau+1$. Zum Beispiel

```
const unsigned int L = 12;      // feste Groesse des Gitters
unsigned int Z1[L*L], Z2[L*L]; // statische Speicheranforderung
unsigned int *Ztau = Z1;        // Zeiger fuer den Zugriff
unsigned int *Ztaup1 = Z2;
```

Der Zugriff kann dann über die Zeiger `Z1` und `Z2` erfolgen. So kann man $Z(\tau)$ und $Z(\tau + 1)$ vertauschen, ohne die gesamten Inhalte kopieren zu müssen.

In zwei Dimensionen hat eine Zelle $z(x, y)$ acht Nachbarn:

- $z(x - 1, y)$
- $z(x - 1, y - 1)$
- $z(x - 1, y + 1)$
- $z(x, y - 1)$
- $z(x, y + 1)$
- $z(x + 1, y)$
- $z(x + 1, y - 1)$
- $z(x + 1, y + 1)$

Für die Implementation müssen wir (mindestens) drei verschiedenen Funktionen schreiben:

a) Entwicklung: Gibt ein neues Gitter zurück, nachdem es nach den obigen Regeln entwickelt wurde

b) Ausgabe: Gibt das Gitter auf Monitor aus

c) Spiel: Initialisiert die Zellen zufällig und „spielt“, bis alle Zellen den Zustand tot angenommen haben oder bis eine obere Schranke für die Zeit erreicht ist.

Für die zufällige Initialisierung der Zellen könne wir den Zufallszahlengenerator aus der vorigen Aufgabe verwenden. Für jede Zelle erzeugt man einen Zufallszahl u. Falls $u < \gamma$, wird die Zelle als tot initialisiert, sonst als lebendig. Wählen Sie zu Beginn $\gamma = 0,5$. Später können Sie untersuchen, wie die Überlebensrate der Population vom Wert von γ abhängt.