0.1 Exp(λ)-Generator Qualitäts und Geschwindgkeitsvergleich

Hier Vergleichen wir den Standardmäßigen Operator rand() und xorschrift182plus in der Qualität der Uniformen Zufallszahlen generiert und die dessen Geschwindgkeit ohne und mit Parallelisierung.

Definition 0.1: Pseudo Random Number Generator

Seien $s, k \in \mathbb{N}$, eine Abbildungsfunktion

$$h\colon \{0,1,\ldots,k\}^s \;\to\; \{0,1,\ldots,k\}^s$$

sowie Anfangswerte $x_1, x_2, \ldots, x_s \in \{0,1,\ldots,k\}$ gegeben. Wir definieren für $i \geq s$

$$x_{i+1} \;=\; h\big(x_i,\; x_{i-1},\; \ldots,\; x_{i-s+1}\big),$$

und setzen

$$u_i \;=\; \frac{x_i}{k}, \quad i \in \mathbb{N}.$$

Die Folge $(u_n)_{n\in\mathbb{N}}$ heißt dann *pseudozufällige Zahlenfolge* in $[0,1]$. Die Anfangswerte $u_1, \ldots, u_s$ (bzw. $x_1, \ldots, x_s$) nennt man *Seeds*, und die Abbildung h wird *Generatorfunktion* genannt.

- **Zustandsdimension s:** $s = 2$, wobei jedes "x_i" ein 64-Bit-Wort ist.
- **Werteraum $\{0,\ldots,k\}$:** Arbeitet hier in $\{0,1\}^{64}$ (also Bitfolgen der Länge 64). Man kann dies als Elemente aus $\{0,\ldots,2^{64}-1\}$ interpretieren.
- **Generatorfunktion h:** Diese Funktion h wird in `xorschrift182plus` durch Bitoperationen (XOR, Bit-Shifts) realisiert. Genauer:

  $$(x_i, x_{i-1}) \;\mapsto\; h\big(x_i, x_{i-1}\big) \;=\; x_{i+1}$$

  wobei die genaue Formel aus mehreren XOR- und Shift-Schritten besteht. Genaueres später.
- **Ausgabe u_i:** Um daraus eine Zufallszahl in $[0,1)$ zu erhalten, wird der (in 64 Bits codierte) Ausgabewert normiert, zum Beispiel durch Division durch 2^{64} oder $2^{64}-1$. Dies entspricht der Vorschrift $u_i = x_i/k$ in der obigen Definition.
- **Seeds:** Die Anfangswerte (x_1, x_2) – in Form von zwei 64-Bit-Zahlen – sind die Seeds. Sie bestimmen den Startpunkt der Folge.

Zufallszahlengenerator: xorschrift182plus:
Der gehört zu Familie der Xorsgift-Mehtoden, die auf dem Körper $\mathbb{F}_2$ (also modulo 2) definiert sind. Die idee ist wie folgt: Nehme zwei 64-Bit Zahlen $s_1, s_2 \in \{0,1\}^{64}$ und stelle diese als einen großen Vektor da $s = (s_1, s_2) \in \{0,1\}^{128}$. Dann ist offensichtlich

$$\{0,1\}^{128} = (\mathbb{F}_2)^{128}$$

Definition 0.2: Bit-shifts

Wir definieren für zwei Binären Vektor $s_1, s_2 \in \{0,1\}^b$

- Den XOR-Operator $\oplus$: $s_1 \oplus s_2$ als die komponentenweise addition auf $\mathbb{F}_2$, i.e.

$$s_1 \oplus s_2 = (\underbrace{s_1(i) + s_2(i)}_{\in \mathbb{F}_2})_{i=1,...,b}$$

- Linksshift $\ll$ um $a < b$ bits, als

$$s_1 \ll a = (s_1(a), ..., s_1(b), \underbrace{0, ..., 0}_{=a \text{ mal}})$$

- Rechtsshift $\gg$ um $a < b$ bits, als

$$s_1 \gg a = (\underbrace{0, ..., 0}_{=a \text{ mal}}, s_1(1), ..., s_1(b-a)).$$

Um die Zufallszahlen zu generieren, nehmen wir zwei 64-Bit Worte s_1, s_2 und bilden diesen wie folgt ab

$$(s_0, s_1) \mapsto \big(s_1,\ s_1 \oplus s_0 \oplus (s_1 \gg b) \oplus (s_0 \gg c)\big) = (s_0', s_1').$$

Die Ausgabe r wird schließlich als Summe der beiden Komponenten berechnet:

$$r = s_0' + s_1' \quad \text{(Addition im Ganzzahlbereich).}$$

Um unsere Zufallzahl zu erzeugen teilen wir

$$u_1 = \frac{r}{2^{64}}.$$

- **Effizienz:** Da alle Operationen in hardwareeffizienten Bit-Operationen umgesetzt werden, ist der Generator sehr schnell.
- **Periodenlänge:** Die Konstruktion ermöglicht eine sehr lange Periode, was insbesondere bei Simulationen und Monte-Carlo-Methoden von Vorteil ist. ($2^{128} - 1$.)

CAT-Test:
Wir betrachten eine Folge ganzzahliger Ausgaben

$$I_1, I_2, I_3, \ldots$$

eines Zufallszahlengenerators (PRNG), wobei jedes $I_n = \lceil 26u_n \rceil \in \{1, 2, \ldots, 26\}$. Wir definieren das *Wort*

$$(\mathrm{C}, \mathrm{A}, \mathrm{T}) = (3,\ 1,\ 20).$$

Dann setzen wir:

$$\tau_1 = \min\big\{\, n \geq 1 : (I_n, I_{n+1}, I_{n+2}) = (3, 1, 20)\big\}.$$

Für $m \geq 1$ definiert man rekursiv

$$\tau_{m+1} \ = \ \min\big\{\, n > \tau_m : (I_n, I_{n+1}, I_{n+2}) = (3, 1, 20)\big\}.$$

Im Mittel (unter der Annahme einer idealen, unabhängigen und gleichverteilten Erzeugung der I_n) gilt

$$\tau_{m+1} - \tau_m \approx 26^3 = 17576.$$

Liegt der beobachtete Abstand systematisch außerhalb eines sinnvollen Konfidenzbereichs, so ist der PRNG nicht geeignet.

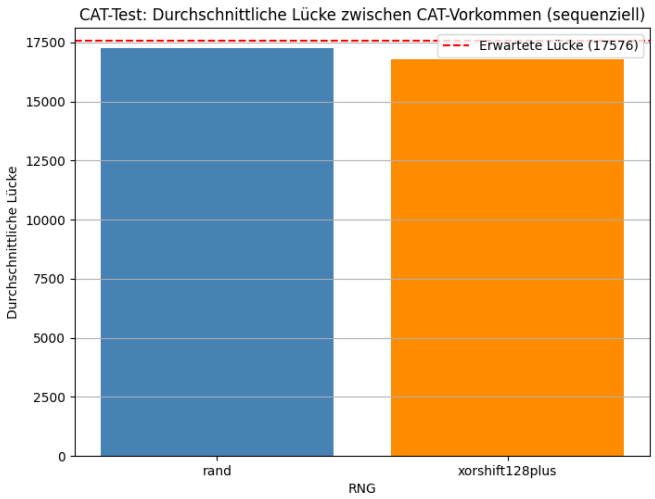

Figure 0.1: CAT-Test

Beide Generatoren bestehen den CAT test, jedoch ist rand() ein wenig besser. **χ^2-Test für Uniformität**:

Beim χ^2-Test wird das Intervall $[0, 1)$ in k gleich große Bins unterteilt. Für jeden Bin wird die Anzahl der Beobachtungen O_i ermittelt. Unter der Nullhypothese der Gleichverteilung ist die erwartete Anzahl pro Bin:

$$E = \frac{N}{k}.$$

Die Teststatistik berechnet sich als

$$T = \sum_{i=1}^{k} \frac{(O_i - E)^2}{E}.$$

Diese Größe folgt, unter der Nullhypothese, einer χ^2-Verteilung mit $k-1$ Freiheitsgraden. 95%-Konfidenzintervall für T etwa zwischen 3.33 und 16.92.

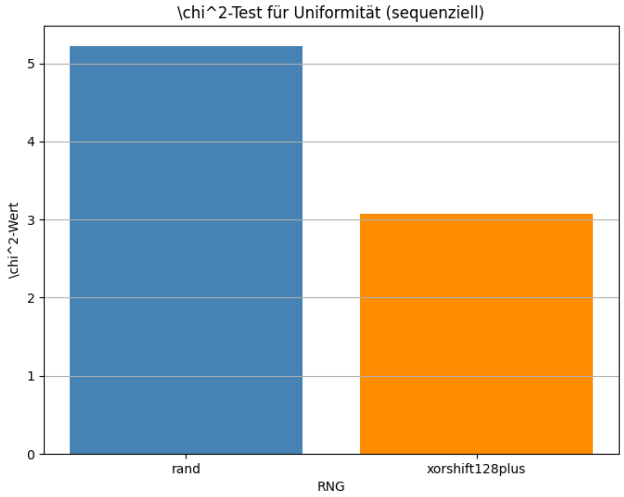

Figure 0.2: χ^2-Test

Hier ist also rand() gut, aber xorschrift182plus nicht signinfikant, also sind die Werte nicht Uniform verteilt. Insgesamt haben wir also das Ergebnis, dass xorschrift182plus kein guter Generator ist.

Und der Geschwindigkeitsvergleich ergeibt:

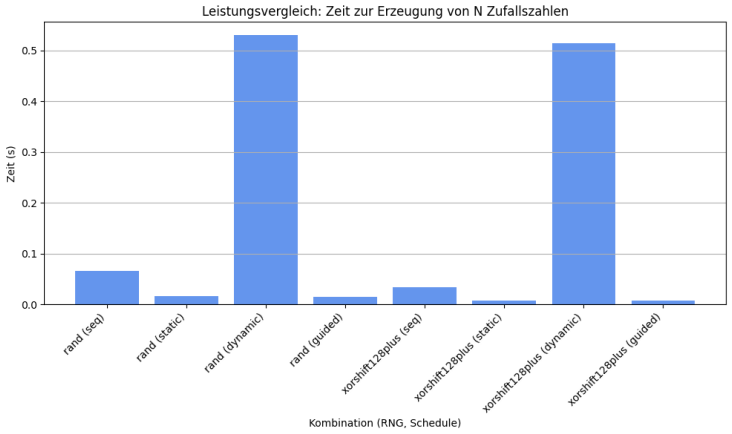

Figure 0.3: Geschwindigkeitsvergleich

In einem dynamischen Scheduling werden die Iterationen nicht vorab in große, feste Blöcke aufgeteilt. Stattdessen gibt es einen gemeinsamen "Aufgabenpool", aus dem jeder Thread, sobald er seine aktuellen Aufgaben (z.B. einen kleinen Chunk von Iterationen) abgearbeitet hat, neue Iterationen „abholt". Das bedeutet, dass jeder Thread – anstatt einen fest zugewiesenen Bereich zu bearbeiten – nach Fertigstellung seines aktuellen Blocks wieder den Pool abfragt, um weitere Aufgaben zu erhalten. Dieser wiederholte Zugriff auf den gemeinsamen Pool erfordert Synchronisationsmechanismen und führt zu Overhead, was insbesondere bei sehr kurzen Aufgaben, dominant wird.