Technische Universität Darmstadt
Fachbereich Informatik
Kryptographie und Computeralgebra

Diplomarbeit

April 2011

SWIFFT-Modifikationen, Korrektur von Operm5

Stephan Mönkehues

Technische Universität Darmstadt
Fachbereich Mathematik

Betreut durch Prof. Dr. Johannes Buchmann,
Dr. Richard Lindner

Danksagung

Ich bedanke mich bei Prof. Dr. Johannes Buchmann, der es versteht Studenten wie mich für die Kryptographie zu begeistern. Ein besonderer Dank geht an Dr. Richard Lindner für seine gute Betreuung. Ich danke ihm auch dafür, dass er es mir möglich machte meine Diplomarbeit über solch ein interessantes Thema zu schreiben.

Für das Suchen und Finden von Fehlern danke ich Patrick Schmidt und Dr. Michael Linker. Ein weiterer Dank geht an Prof. Robert G. Brown und Teeproduzenten in aller Welt.

Erklärung

Hiermit versichere ich, dass der Inhalt dieser Diplomarbeit Ergebnis meiner Arbeit ist und alle verwendeten Quellen angegeben sind. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Einleitung

In einer mehr und mehr vernetzten Welt wird die Kryptographie immer wichtiger. Nicht zuletzt durch vermehrtes Cloud-Computing wird ihre Anwendung auch in Zukunft weiter zunehmen. Wird sie z.B. in Smartcards, Autofunkschlüsseln usw. eingesetzt, wird sie oft gar nicht mehr wahrgenommen. Viele kryptographische Verfahren verwenden Hashfunktionen. Einsatzfelder für Hashfunktionen sind z.B. Integritätsprüfungen oder digitale Signaturen. Diese Arbeit beschäftigt sich mit ihrer Verwendung als Pseudozufallsgenerator. Für viele Kryptosysteme sind Zufallszahlen, oder meist Pseudozufallszahlen grundlegend. So wie das beste Türschloss nicht viel wert ist, falls jeder Fünfte das gleiche Türschloss mit demselben Schlüssel besitzt, sind die besten Kryptosysteme nicht viel wert, falls geheime Schlüssel nicht (pseudo-)zufällig gewählt werden. Um die Güte von Zufallszahlen beurteilen zu können, sind Zufallstests sehr hilfreich.

Der erste Teil dieser Arbeit beschäftigt sich mit SWIFFT, dem Hauptbaustein der Hashfunktion SWIFFTX. SWIFFTX wurde von Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert und Alon Rosen entworfen und war ein SHA-3-Kandidat, ist aber leider ausgeschieden. Allerdings nicht wie viele andere Kandidaten, weil sie gebrochen wurde, sondern weil sie im Vergleich zu den jetzigen Finalisten langsamer ist. Es wird gezeigt, wie man SWIFFT bei bleibender Effizienz noch flexibler und damit interessanter machen kann.

Dazu wird in Kapitel 1 zunächst eine kurze Einführung in die Gittertheorie gegeben. Kapitel 2 und 3 stellen den *Number-Theoretic Transform* (NTT) vor und wie man mit ihm Polynommultiplikationen und somit SWIFFT berechnen kann. Die Kapitel 4, 5 und 6 zeigen wie man den NTT geschickt und effizient berechnen kann.

Der zweite Teil behandelt Zufallstests. Hauptbestandteil hierbei ist die Korrektur von Operm5, einem der vorgestellten Zufallstests. Er ist in den Testsuiten *dieharder* von Robert G. Brown, und dem Vorgänger *diehard* von George Marsaglia enthalten. Ergebnis ist ein neuer Rechenweg, die tatsächliche Berechnung und daraus resultierend ein neuer Quellcode, der inzwischen in der aktuellen *dieharder*-Version enthalten ist.

Kapitel 7 enthält eine Einführung in die Wahrscheinlichkeitstheorie, stellt Test-Suiten und deren Zufallstests vor und wendet diese auf SWIFFTX an. Kapitel 8 enthält die Theorie und die zur Korrektur von Operm5 nötigen Berechnungen.

1 Gitter und Berechnungsprobleme

Dieses Kapitel soll lediglich eine knappe Einführung in die für diese Arbeit relevante Gittertheorie darstellen. Als Quellen dienten Kapitel 6 des Skripts von Prof. Alexander May zur Kryptoanalyse [14] und das Skript zur Vorlesung „Post-Quantum Cryptography“ von Prof. Buchmann, geschrieben von Patrick Schmidt [17].

Definition 1.1 (Gitter)**.** Ein Gitter ist eine diskrete abelsche Untergruppe des $\mathbb{R}^n$.

Definition 1.2 (Gitter,Gitterdimension)**.** Seien $b_1, b_2, ..., b_d \in \mathbb{R}^n$ linear unabhängige Vektoren. Dann ist

$$L = \left\{ v \in \mathbb{R}^n \;\middle|\; \sum_{i=1}^{d} a_i \cdot b_i, a_i \in \mathbb{Z} \right\}$$

ein Gitter mit Dimension d.

Die Matrix $B = \{b_1, b_2, ..., b_d\}$ wird auch als Basis von L bezeichnet. Ein mehrdimensionales Gitter hat unendlich viele Basen.

Definition 1.3 ($\lambda_i(L)$)**.** Sei L ein Gitter mit Dimension d. Das i-te sukzessive Minimum $\lambda_i(L)$ von L ist der minimale Radius des Balls (der Sphäre) um Null, der i linear unabhängige Vektoren enthält.

Es ist im Allgemeinen schwierig die sukzessiven Minima auszurechnen.

Definition 1.4 (SVP)**.** Das *shortest vector problem* (*SVP*) ist wie folgt definiert: Gegeben eine Basis B eines Gitters L. Finde $x \in L$ mit $\|x\| \leq \lambda_1(L)$ und $x \neq 0$.

Das Problem besteht also darin, einen kürzesten Vektor des Gitters zu finden. Hat man eine ‚gute‘ Basis, ist es einfach einen kürzesten Vektor abzulesen. Hat man allerdings eine ‚schlechte‘ Basis, ist es schwer.

Ajtais Reduktion Ajtai bewies, dass man das Problem einen kurzen Vektor in irgend einem Gitter der Dimension d zu finden auf das Problem einen kurzen Vektor in einem zufälligen Gitter der Dimension $m \gg d$ zu finden reduzieren kann.

Theorem 1.5. *Falls es einfach ist einen kurzen Vektor in einem zufälligen Gitter der Dimension $m \gg d$ zu finden, dann ist es einfach einen kurzen Vektor in allen Gittern der Dimension d zu finden.*

Beweis. Ajtai konstruierte eine Kompressionsfunktion f, die kolissionsresistent ist, falls das *SVP* in einigen Gittern der Dimension m hart ist. Wähle hierzu positive Zahlen $d, q, e, m \in \mathbb{N}$ mit d sehr klein und $m \gg \frac{d \cdot \log_2(q)}{\log_2(e)}$ und eine zufällige Matrix $A \in \mathbb{Z}_q^{d \times m}$. Die Kompressionsfuntion ist dann gegeben durch

$$f_A : \mathbb{Z}_e^m \to \mathbb{Z}_q^d : x \mapsto Ax \pmod q.$$

Da $x \in \mathbb{Z}_e^m$ die Bitlänge $m \cdot \log_2(e)$ und $Ax \pmod q \in \mathbb{Z}_q^d$ die Bitlänge $d \cdot \log_2(q) < m \cdot \log_2(e)$ hat, ist f eine Kompressionsfunktion.
Findet man nun eine Kollision von f_A, hat man $x_1, x_2 \in \mathbb{Z}_e^m$ mit $Ax_1 = Ax_2 \pmod q$ gefunden. Also gilt $A(x_1 - x_2) = 0 \pmod q$. Der Vektor $y = (y_1, y_2, ..., y_d) = x_1 - x_2$ ist kurz, da $|y_i| \leq e$ und somit (nach euklidischer Norm) $|y| \leq \sqrt{(e-1)^2 + (e-1)^2 + ... + (e-1)^2} = \sqrt{d \cdot (e-1)^2}$. Wählt man $e = 2$, erhält man also $|y| \leq 1$. Die Menge $L = \{y \in \mathbb{Z}^m | Ay = 0 \pmod q\}$ ist ein Gitter, da die Lösungen diskret sind und mit $y_1 \in L$ und $y_2 \in L$ auch $y_1 + y_2$ in L liegt. □

Definition 1.6 (Ideale Gitter)**.** Sei $R = \mathbb{Z}[x]/(f)$ ein Ring modulo einem monischen Polynom $f = a_0 \cdot x^0 + a_1 \cdot x^1 + ... + a_d \cdot x^d$ vom Grad d. Da R isomorph zu $\mathbb{Z}^d$ und eine additive Gruppe ist, ist R auch ein Gitter. Gitter dieser Form heißen Idealgitter.

Der Ring R ist isomorph zu $\mathbb{Z}^d$, also $R^{\frac{m}{d}}$ (m passend) isomorph zu $\mathbb{Z}^m$. Werden die Koeffizienten von R modulo q gerechnet, schreiben wir hierfür R_q. Für R_q gilt $R_q \cong \mathbb{Z}_q^d$. Überträgt man Ajtais Konstruktion einer Kompressionsfunktion auf ideale Gitter, so erhält man

$$\begin{array}{rcccl} f_{a_0,a_1,...,a_{\frac{m}{d}-1}} : & \mathbb{Z}_e^m \cong R_e^{\frac{m}{d}} & \longrightarrow & R_q \cong \mathbb{Z}_q^d \\ : & (x_0, x_1, ..., x_{\frac{m}{d}-1}) & \longmapsto & \sum_{i=0}^{\frac{m}{d}-1} a_i \cdot x_i \pmod q \end{array}$$

Ein Vorteil von $f_{a_0,a_1,...,a_{\frac{m}{d}-1}}$ gegenüber f_A ist, dass man zu seiner Repräsentation weniger Elemente und somit weniger Speicherplatz benötigt. Für f_A werden $m \cdot d$ Elemente aus $\mathbb{Z}_q$ benötigt. Für $f_{a_0,a_1,...,a_{\frac{m}{d}-1}}$ hingegen langen $\frac{m}{d}$ Elemente aus R_q, gleichzusetzen mit m Elementen aus $\mathbb{Z}_q$.

Wir möchten jedoch ein größeres Augenmerk auf die Geschwindigkeit der Berechnungen von f_A und $f_{a_0,a_1,...,a_{\frac{m}{d}-1}}$ legen.

Zur Erinnerung: $\mathcal{O}$ ist eines der Landau-Symbole („Big O Notation“). Seien f und g reellwertige Funktionen, dann ist $\mathcal{O}$ wie folgt definiert:

$$f \in \mathcal{O}(g) \Longleftrightarrow 0 \leq \limsup_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$$

Um f_A zu berechnen werden durch das Multiplizieren der Matrix A an x insgesamt $\mathcal{O}(m \cdot d)$ Rechenschritte gebraucht. Für $f_{a_0,a_1,...,a_{\frac{m}{d}-1}}$ fallen konservativ $\frac{m}{d}$ Polynommultiplikationen an. Pro Polynommultiplikation ergeben sich $(d \cdot d)$ Multiplikationen und Additionen, was ebenfalls zu einer Gesamtlaufzeit von $\mathcal{O}(m \cdot d)$ führt.

Im folgendem Kapitel wird der Number-Theoretic Transform vorgestellt, und wie man mit ihm Polynommultiplikationen der Form $(a_0, a_1, ..., a_{d-1}) \cdot (x_0, x_1, ..., x_{d-1}) \pmod q$ durchführen kann. In Kapitel 4 werden effiziente Algorithmen vorgestellt, mit denen es möglich ist, die Polynommultiplikation mit einer Laufzeit von $\mathcal{O}(d \cdot \log(d))$ zu berechnen.

2 Number-Theoretic Transform

Der Number-Theoretic Transform (NTT) ist dem Discrete Fourier Transform (DFT) ähnlich, einer Form der Fourier-Transformation, die ein wichtiges Werkzeug in der digitalen Signalverarbeitung ist. Im Gegensatz zum DFT operiert der NTT aber nicht auf komplexen Zahlen, sondern auf ganzen Zahlen modulo einer Zahl p (nicht notwendig prim).

Ziel dieses Kapitels ist zu zeigen, wie man Polynommultiplikationen der Form $g(\alpha) \cdot h(\alpha) = (g_0 \cdot \alpha^0, g_1 \cdot \alpha^1, ..., g_{d-1} \cdot \alpha^{d-1}) \cdot (h_0 \cdot \alpha^0, h_1 \cdot \alpha^1, ..., h_{d-1} \cdot \alpha^{d-1}) \pmod p$ mithilfe des NTTs berechnen kann. Es wird in $\mathbb{Z}_p$ gerechnet, für eine bessere Lesbarkeit und aus Platzgründen wird aber das ‚$(\text{mod } p)$' weggelassen.

Definition 2.1. (Number-theoretic transform (NTT)). Seien $p \in \mathbb{N}$ und γ so gewählt, dass $ord(\gamma) = d \pmod p$ und die Inverse $d^{-1} \pmod p$ existiert. Weiter sei $\boldsymbol{x} = (x_0, ..., x_{d-1}) \in \mathbb{Z}_p^d$. Dann hat $NTT(\boldsymbol{x})$ die Ordnung d und ist ein Vektor mit

$$NTT(\boldsymbol{x}) : \mathbb{Z}_p^d \to \mathbb{Z}_p^d : NTT(\boldsymbol{x})_i = \sum_{k=0}^{d-1} x_k \cdot \gamma^{i \cdot k}.$$

für $0 < i < d - 1$.

Theorem 2.2. *Der NTT ist bijektiv, für seine Inverse NTT^{-1} gilt:*

$$NTT^{-1}(\boldsymbol{y})_j = d^{-1} \cdot \sum_{i=0}^{d-1} y_i \cdot (\gamma^{-1})^{j \cdot i}$$

Beweis.

$$\begin{aligned}
(NTT^{-1}(NTT(\boldsymbol{x})))_j &= d^{-1} \cdot \sum_{i=0}^{d-1} \left(\sum_{k=0}^{d-1} x_k \cdot \gamma^{i \cdot k} \right) \cdot (\gamma^{-1})^{j \cdot i} \\
&= d^{-1} \cdot \sum_{i=0}^{d-1} \left(\sum_{k=0}^{d-1} x_k \cdot (\gamma^{k-j})^i \right) \\
&= d^{-1} \cdot \sum_{k=0}^{d-1} x_k \left(\sum_{i=0}^{d-1} (\gamma^{k-j})^i \right)
\end{aligned}$$

Für $k = j \pmod d$ gilt

$$\sum_{i=0}^{d-1} (\gamma^{k-j})^i = \sum_{i=0}^{d-1} 1^i = d.$$

Behauptung: Für $k \neq j \pmod{d}$ ist $\sum_{i=0}^{d-1}(\gamma^{k-j})^i = 0$.

$$
\begin{aligned}
1 + \gamma^{k-j} \cdot \sum_{i=0}^{d-1}(\gamma^{k-j})^i &= 1 + \cdot \sum_{i=1}^{d}(\gamma^{k-j})^i \\
1 + \gamma^{k-j} \cdot \sum_{i=0}^{d-1}(\gamma^{k-j})^i &= (\gamma^{k-j})^0 + \sum_{i=1}^{d-1}(\gamma^{k-j})^i + (\gamma^{k-j})^d \\
1 + \gamma^{k-j} \cdot \sum_{i=0}^{d-1}(\gamma^{k-j})^i &= \sum_{i=1}^{d-1}(\gamma^{k-j})^i + 1 \\
\gamma^{k-j} \cdot \sum_{i=0}^{d-1}(\gamma^{k-j})^i &= \sum_{i=1}^{d-1}(\gamma^{k-j})^i \\
(\gamma^{k-j} - 1) \cdot \sum_{i=0}^{d-1}(\gamma^{k-j})^i &= 0
\end{aligned}
$$

Aus $\gamma^{k-j} \neq 1$ folgt die Behauptung. Setzt man beides ein, folgt die Inverse:

$$
\begin{aligned}
(NTT^{-1}(NTT(\boldsymbol{x})))_j &= d^{-1} \cdot \sum_{k=0}^{d-1} x_k \left(\sum_{i=0}^{d-1}(\gamma^{k-j})^i \right) \\
&= d^{-1} \cdot x_j \cdot d \\
&= x_j
\end{aligned}
$$

□

Definition 2.3. (Cauchy-Produkt). Das Cauchy-Produkt ist ein spezieller Fall der zyklischen, diskreten Faltung, weswegen sie auch Cauchy-Faltung genannt wird. Seien $\boldsymbol{g} = \sum_{k=0}^{\infty} g_k$ und $\boldsymbol{h} = \sum_{k=0}^{\infty} h_k$ unendliche Reihen. Dann ist die Cauchy-Produktreihe definiert durch:

$$\boldsymbol{c} = \sum_{k=0}^{\infty} c_k \text{ mit } c_k = \sum_{l=0}^{k} g_l h_{k-l}$$

Man schreibt für die (Cauchy-)Faltung auch $\boldsymbol{c} = \boldsymbol{g} * \boldsymbol{h}$.

Der Fall, der uns interessiert, ist das Cauchy-Produkt von (endlichen) Potenzreihen. Seien g und h Polynome vom Grad kleiner gleich $(d-1)$, repräsentiert durch die Folgen ihrer Koeffizienten: $\boldsymbol{g} = (g_0, g_1, ..., g_{d-1})$ und $\boldsymbol{h} = (h_0, h_1, ..., h_{d-1})$. Die Folgenglieder sind nach ihrem ‚Grad' aufsteigend geordnet, also $g(\alpha) = g_0 \cdot \alpha^0 + g_1 \cdot \alpha^1 + ... + g_{d-1} \cdot \alpha^{d-1}$. Die Koeffizienten von $c = g \cdot h$ entsprechen der Folge von $\boldsymbol{c} = \boldsymbol{g} * \boldsymbol{h}$. Es gilt $grad(c) \leq (d-1) + (d-1) = 2(d-1)$, also langen uns die ersten $(2d-1)$ Folgenglieder von $\boldsymbol{c}$.

Sind durch Anforderungen an den NTT nur bestimmte Ordnungen möglich, nehmen wir die kleinstmögliche Ordnung die größer gleich $(2d-1)$ ist. Wir wählen die Ordnung $2d$.

Für den nächsten Satz benötigen wir, dass die Repräsentationen von g, h und c, also $\boldsymbol{g}, \boldsymbol{h}$ und $\boldsymbol{c}$, die gleiche Länge haben. Wir füllen $\boldsymbol{g}$ und $\boldsymbol{h}$ mit Nullen auf $2d$ Folgenglieder auf. Dies ergibt z.B. $\boldsymbol{g} = (g_0, g_1, ..., g_{d-1}, 0, ..., 0)$.

Theorem 2.4. *Sei $\boldsymbol{c} = \boldsymbol{g} * \boldsymbol{h}$, außerdem ω ein Element mit Ordnung $n = 2d$ und p eine Primzahl mit $(p-1)$ einem Vielfachen von n. Dann wird die Faltung unter dem NTT zu einer Multiplikation:*

$$NTT(\boldsymbol{c}) = NTT(\boldsymbol{g}) \odot NTT(\boldsymbol{h}),$$

wobei $\odot$ die elementweise Multiplikation ist.

Beweis.

Wir zeigen: $NTT(\boldsymbol{c})_i = NTT(\boldsymbol{g})_i \cdot NTT(\boldsymbol{h})_i$

$$\begin{aligned} NTT(\boldsymbol{c})_i &= \sum_{k=0}^{n-1} (c_k) \cdot \omega^{i \cdot k} \\ &= \sum_{k=0}^{n-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \right) \cdot \omega^{i \cdot k} \end{aligned}$$

$$\begin{aligned} NTT(\boldsymbol{g})_i \cdot NTT(\boldsymbol{h})_i &= \left(\sum_{l=0}^{n-1} g_l \cdot \omega^{i \cdot l} \right) \cdot \left(\sum_{k=0}^{n-1} h_k \cdot \omega^{i \cdot k} \right) \\ &= \sum_{k=0}^{n-1} h_k \cdot \omega^{i \cdot k} \cdot \left(\sum_{l=0}^{n-1} g_l \cdot \omega^{i \cdot l} \right) \\ &= \sum_{k=0}^{n-1} \left(\sum_{l=0}^{n-1} g_l \cdot h_k \cdot \omega^{i \cdot (l+k)} \right) \end{aligned}$$

Umsortieren liefert

$$= \sum_{k=0}^{n-1} \left(\sum_{l=0}^{n-1} g_l \cdot h_{k-l \pmod n} \cdot \omega^{i \cdot (l + (k-l \pmod n))} \right)$$

$$NTT(\boldsymbol{g})_i \cdot NTT(\boldsymbol{h})_i = \sum_{k=0}^{n-1} \left(\sum_{l=0}^{n-1} g_l \cdot h_{k-l \pmod n} \cdot \omega^{i \cdot k} \right)$$

Behauptung: Wir können annehmen, dass $l \leq k$.

$$= \sum_{k=0}^{n-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l \pmod n} \cdot \omega^{i \cdot k} \right)$$
$$= NTT(\boldsymbol{c})_i$$

Beweis zur Behauptung: Falls $l > k$ ist, dann ist $k - l < 0$. Für $-(d-1) < k - l < 0$ ist $h_{k-l \pmod d} = 0$. Für $k - l < (d-1)$ ist $l > (d-1)$ und somit $g_l = 0$. Wir können also annehmen $l \leq k$. □

Folglich gilt $\boldsymbol{c} = NTT^{-1}(NTT(\boldsymbol{g}) \odot NTT(\boldsymbol{h})) = (c_0, c_1, ..., c_{2d-1})$, was unser gesuchtes Polynom $\boldsymbol{c}(\alpha) = c_0 \cdot \alpha^0 + ... + c_{2d-1} \cdot \alpha^{2d-1} = g(\alpha) \cdot h(\alpha)$ repräsentiert.

Anmerkung: Möchte man nicht, dass bei der Polynommultiplikation modulo p gerechnet wird, kann man dies umgehen, indem man p groß genug wählt.

Rechnet man zusätzlich noch modulo $\alpha^d + 1$, ist es ausreichend mit NTTs der Ordnung d zu rechnen. Dazu benötigen wir eine andere Repräsentation der Polynome. Sei ω weiterhin ein Element mit Ordnung $2d$, dann repräsentieren wir $g(\alpha) = g_0\alpha^0 + g_1\alpha^1 + ... + g_{d-1}\alpha^{d-1}$ statt mit $\boldsymbol{g} = (g_0, ..., g_{d-1}, 0, ..., 0)$ mit $\hat{\boldsymbol{g}} = (g_0 \cdot \omega^0, ..., g_{d-1} \cdot \omega^{d-1}, 0, ..., 0)$ (und analog $\boldsymbol{h}$).

Theorem 2.5. *Für die zyklische Faltung von $\hat{\boldsymbol{g}}$ und $\hat{\boldsymbol{h}}$ gilt*

$$\hat{\boldsymbol{g}} * \hat{\boldsymbol{h}} = \sum_{k=0}^{d-1} \left(\sum_{l=0}^{d-1} \hat{g}_l \cdot \hat{h}_{k-l \pmod d} \right)$$

Beweis.

$$\hat{\boldsymbol{g}} * \hat{\boldsymbol{h}} = \sum_{k=0}^{n-1} \left(\sum_{l=0}^{k} \hat{g}_l \cdot \hat{h}_{k-l} \right)$$
$$= \sum_{k=0}^{n-1} \left(\sum_{l=0}^{k} g_l \cdot \omega^l \cdot h_{k-l} \cdot \omega^{k-l} \right)$$
$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^k \right) + \sum_{k=d}^{2d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^k \right)$$

$$\hat{\boldsymbol{g}} * \hat{\boldsymbol{h}} = \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^k \right) + \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k+d} g_l \cdot h_{k+d-l} \cdot \omega^{k+d} \right)$$

Für $l \leq k$ ist $h_{k+d-l} = 0$.

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^k \right) + \sum_{k=0}^{d-1} \left(\sum_{l=k+1}^{k+d} g_l \cdot h_{k+d-l} \cdot \omega^{k+d} \right)$$

Und für $l \geq d$ ist $g_l = 0$.

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^k \right) + \sum_{k=0}^{d-1} \left(\sum_{l=k+1}^{d-1} g_l \cdot h_{k+d-l} \cdot \omega^{k+d} \right)$$

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l} \cdot \omega^l \omega^{k-l} \right) + \sum_{k=0}^{d-1} \left(\sum_{l=k+1}^{d-1} g_l \cdot h_{k+d-l} \cdot \omega^l \omega^{k+d-l} \right)$$

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{k} g_l \cdot h_{k-l \pmod d} \cdot \omega^l \omega^{k-l \pmod d} \right)$$

$$+ \sum_{k=0}^{d-1} \left(\sum_{l=k+1}^{d-1} g_l \cdot h_{k-l \pmod d} \cdot \omega^l \omega^{k-l \pmod d} \right)$$

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{d-1} g_l \cdot h_{k-l \pmod d} \cdot \omega^l \omega^{k-l \pmod d} \right)$$

$$= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{d-1} \hat{g}_l \cdot \hat{h}_{k-l \pmod d} \right)$$

□

Seien $c(\alpha) = g(\alpha) \cdot h(\alpha) \pmod{\alpha^d + 1}$ und $\hat{\boldsymbol{c}} = \hat{\boldsymbol{g}} * \hat{\boldsymbol{h}}$. Da $\omega^d = -1$ ist, gilt $\hat{\boldsymbol{c}} = (c_0\omega^0, c_1\omega^1, ..., c_{d-1}\omega^{d-1})$. Da ω invertierbar ist (siehe unten), können wir daraus die Koeffizienten von $c(\alpha)$ erhalten.

$$\begin{aligned} & \omega^{2d} && = && 1 \\ \iff & \omega \cdot \omega^{2d-1} && = && 1 \\ \iff & \omega^{-1} && = && \omega^{2d-1} \end{aligned}$$

Und damit ist ω invertierbar!

Da p nicht prim sein muss, ist nicht gegeben, dass alle Zahlen Inversen haben.

Nun können wir zeigen, dass die Faltung $\hat{\boldsymbol{g}} * \hat{\boldsymbol{h}}$ unter einem nur halb so großen NTT wie in Theorem 2.5 zur Multiplikation wird.

Theorem 2.6. *Sei ω ein Element mit Ordnung $2d$ und γ ein Element mit Ordnung d (z.B. $\gamma = \omega^2$). Seien weiter*

$$\hat{\boldsymbol{g}} = (g_0\omega^0, g_1\omega^1, ..., g_{d-1}\omega^{d-1}), \quad \hat{\boldsymbol{h}} = (h_0\omega^0, h_1\omega^1, ..., h_{d-1}\omega^{d-1})$$

$$\textit{und} \quad \hat{\boldsymbol{c}} = \hat{\boldsymbol{g}} * \hat{\boldsymbol{h}} \quad \textit{mit} \quad \hat{c_k} = \sum_{l=0}^{d-1} \hat{g}_l \cdot \hat{h}_{k-l \mod d}.$$

Dann wird die Faltung $\hat{\boldsymbol{g}} * \hat{\boldsymbol{h}}$ unter einem NTT mit Ordnung d zu einer Multiplikation:

$$NTT(\hat{\boldsymbol{c}}) = NTT(\hat{\boldsymbol{g}}) \odot NTT(\hat{\boldsymbol{h}})$$

Beweis. Wir zeigen

$$NTT(\hat{\boldsymbol{c}})_i = NTT(\hat{\boldsymbol{g}})_i \cdot NTT(\hat{\boldsymbol{h}})_i.$$

$$\begin{aligned}
NTT(\hat{\boldsymbol{g}})_i \cdot NTT(\hat{\boldsymbol{h}})_i &= \sum_{k=0}^{d-1} \hat{g}_k \cdot \gamma^{ik} \cdot \sum_{l=0}^{d-1} \hat{h}_l \cdot \gamma^{il} \\
&= \sum_{l=0}^{d-1} \left(\sum_{k=0}^{d-1} \hat{h}_l \cdot \hat{g}_k \cdot \gamma^{ik} \cdot \gamma^{il} \right) \\
&= \sum_{l=0}^{d-1} \left(\sum_{k=-l}^{d-1-l} \hat{g}_{k \pmod d} \cdot \hat{h}_l \cdot \gamma^{i(k+l \pmod d)} \right) \\
&= \sum_{l=0}^{d-1} \left(\sum_{k=0}^{d-1} \hat{g}_{k-l \pmod d} \cdot \hat{h}_l \cdot \gamma^{i(k+l-l \pmod d)} \right) \\
&= \sum_{k=0}^{d-1} \left(\sum_{l=0}^{d-1} \hat{g}_{k-l \pmod d} \cdot \hat{h}_l \cdot \gamma^{ik} \right) \\
&= \sum_{k=0}^{d-1} \left(\hat{c}_k \cdot \gamma^{ik} \right) \\
&= NTT(\hat{\boldsymbol{c}})_i
\end{aligned}$$

□

Wir erhalten

$$\hat{\boldsymbol{c}} = NTT^{-1}(NTT(\hat{\boldsymbol{g}}) \odot NTT(\hat{\boldsymbol{h}})) = (\omega^0 c_0, \omega^1 c_1, ..., \omega^{d-1} c_{d-1}).$$

Und da ω invertierbar ist erhalten wir daraus c mit $c = g \cdot h \pmod{\alpha^d + 1}$.

3 SWIFFT

Die Existenz des NTTs und Ajtais Idee machten sich Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert und Alon Rosen zu eigen und entwarfen die Kompressionsfunktion SWIFFT. Aufbauend auf SWIFFT entwarf die Gruppe, erweitert um Yuriy Arbitman und Gil Dogon, die Hashfunktion SWIFFTX. Die Bewerbungsunterlagen von SWIFFTX für die SHA3-Stelle mit umfangreicher Dokumentation sind öffentlich [2]. Die Autoren von SWIFFT hatten die Idee Ajtais Kompressionsfunktion mit der Hilfe des NTTs zu berechnen. Mit effizienten NTT-Algorithmen ist es so möglich gegenüber der normalen Berechnung deutlich schneller zu sein.

Sei d eine Potenz von 2, $m > 0$ eine ganze Zahl und p eine Primzahl. Wir definieren den Ring $R_p = \mathbb{Z}_p[\alpha]/(\alpha^d + 1)$. Seien weiter $\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_m$ zufällig gewählte Elemente aus R. Dann ist eine SWIFFT-Funktion durch

$$\begin{aligned} f_{m,d,p} : R_2^m &\longrightarrow R_p \\ : (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_m) &\longmapsto \boldsymbol{c} := \sum_{i=1}^{m} (\boldsymbol{a}_i \cdot \boldsymbol{x}_i) \end{aligned}$$

definiert. Die $\boldsymbol{x}_i$ sind Elemente aus R_p mit *binären* Koeffizienten. Die SWIFFT-Funktionen sind also Kompressionsfunktionen für ideale Gitter. Die Idee war die m Polynommultiplikationen $\boldsymbol{a}_i \cdot \boldsymbol{x}_i$ mithilfe des vorgestellten NTTs durchzuführen. Sei dazu ω ein Element mit Ordnung $2d$. Dann schreiben wir wieder $\hat{c} = \omega^0 c_0 + \omega^1 c_1 + ... + \omega^{d-1} c_{d-1}$ für das Polynom, dessen Koeffizienten mit Potenzen von ω multipliziert werden. Für den NTT verwenden wir ein beliebiges γ mit Ordnung d, z.B. ω^2. Dann gilt

$$\sum_{i=1}^{m} (\boldsymbol{a}_i \cdot \boldsymbol{x}_i) \cong \hat{\boldsymbol{c}} = \sum_{i=1}^{m} NTT^{-1} \left(NTT(\hat{\boldsymbol{a}}_{\boldsymbol{i}}) \odot NTT(\hat{\boldsymbol{x}}_{\boldsymbol{i}}) \right)$$

Wir erinnern uns, dass die Schwierigkeit darin besteht ein Urbild für gegebenes $\boldsymbol{c}$ zu finden. Da der NTT bijektiv ist, ist $\boldsymbol{a}_i \cdot \boldsymbol{x}_i$ schon eindeutig durch $(NTT(\boldsymbol{a}_i) \odot NTT(\boldsymbol{x}_i))$ bestimmt. Da uns dies langt, können wir den inversen NTT weglassen.

$$\sum_{i=1}^{m} (\boldsymbol{a}_{\boldsymbol{i}} \cdot \boldsymbol{x}_{\boldsymbol{i}}) \cong \sum_{i=1}^{m} \left(NTT(\hat{\boldsymbol{a}}_{\boldsymbol{i}}) \odot NTT(\hat{\boldsymbol{x}}_{\boldsymbol{i}}) \right)$$

Mit $\widetilde{\boldsymbol{a}}_i = NTT(\hat{\boldsymbol{a}}_{\boldsymbol{i}})$ wird daraus

$$\sum_{i=1}^{m} (\boldsymbol{a}_i \cdot \boldsymbol{x}_i) \cong \sum_{i=1}^{m} (\widetilde{\boldsymbol{a}}_i \odot NTT(\hat{\boldsymbol{x}}_{\boldsymbol{i}})).$$

Da jedes Polynom $\boldsymbol{a}_i$ zufällig gewählt werden soll, ist auch $NTT(\hat{\boldsymbol{a}}_{\boldsymbol{i}})$ zufällig. Daher kann man genauso gut $\widetilde{\boldsymbol{a}}_i$ zufällig wählen. Wir erhalten die Funktion

$$\begin{aligned} g_{m,d,p} : R_2^m &\longrightarrow R_p \\ : (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_m) &\longmapsto \sum_{i=1}^{m} (\widetilde{\boldsymbol{a}}_i \odot NTT(\hat{\boldsymbol{x}}_{\boldsymbol{i}})). \end{aligned}$$

3.1 SWIFFT in SWIFFTX

In SWIFFTX wurden die Parameter $d = 64, m = 32$ und $p = 257$ gewählt. Gerechnet wird in $R_p = \mathbb{Z}_p[\alpha]/(\alpha^d + 1)$. Eingabe ist eine $m \cdot d$ -Bit-Matrix, die die m $\boldsymbol{x}_i$-Polynome repräsentiert. Da unsere Koeffizienten binär sind, entspricht dies 2048 Bit.

Eingabematrix: $\begin{bmatrix} x_{0,0} & \cdots & x_{0,31} \\ \vdots & \ddots & \vdots \\ x_{63,0} & \cdots & x_{63,31} \end{bmatrix}$

Um einen NTT mit Ordnung d verwenden zu können, benötigen wir ein Element ω mit $ord(\omega) = 2d = 128$. In SWIFFTX ist dies $\omega = 42$ (mehr zur Wahl von ω im Kapitel 6 über Vorberechnungen). Es folgt die Multiplikation der Zeilen mit ω^i in $\mathbb{Z}_{257}$:

$$\begin{bmatrix} \hat{x}_{0,0} & \cdots & \hat{x}_{0,31} \\ \hat{x}_{1,0} & \cdots & \hat{x}_{1,31} \\ \vdots & \ddots & \vdots \\ \hat{x}_{63,0} & \cdots & \hat{x}_{63,31} \end{bmatrix} = \begin{bmatrix} x_{0,0} \cdot \omega^0 & \cdots & x_{0,31} \cdot \omega^0 \\ x_{1,0} \cdot \omega^1 & \cdots & x_{1,31} \cdot \omega^1 \\ \vdots & \ddots & \vdots \\ x_{63,0} \cdot \omega^{63} & \cdots & x_{63,31} \cdot \omega^{63} \end{bmatrix} \pmod p$$

Wir bezeichnen die Spalte $(x_{0,i} \cdot \omega^0,\ x_{1,i} \cdot \omega^1,\ ..., x_{63,i} \cdot \omega^{63})^T$ mit $\hat{\boldsymbol{x}}_{\boldsymbol{i}}$.

Der NTT-Part: Für den NTT wird $\gamma = \omega^2$ mit $ord(\gamma) = 64$ verwendet. Der NTT-Algorithmus wird auf die Spalten der Matrix angewendet. Für $j = 0, ..., 31$ sei

$$F^{(j)} = NTT(\hat{\boldsymbol{x}}_{\boldsymbol{j}}).$$

Das heißt

$$F_i^{(j)} = NTT(\hat{\boldsymbol{x}}_{\boldsymbol{j}})_i = \sum_{k=0}^{63} \hat{x}_{j,k} \cdot \gamma^{ik} = \sum_{k=0}^{63} \omega^k \cdot x_{j,k} \cdot \omega^{2 \cdot ik} = \sum_{k=0}^{63} x_{j,k} \cdot \omega^{(2i+1)k}.$$

Im letzten Schritt wird aufsummiert und man erhält

$$z = \sum_{j=0}^{31} \widetilde{a_j} \odot F^{(j)} \quad \text{also} \quad z_i = \sum_{j=0}^{31} \widetilde{a_{i,j}} \cdot F_i^{(j)}.$$

Die $\widetilde{a_{i,j}}$ wurden in SWIFFTX aus den ersten Stellen von π gewonnen.

4 NTT-Algorithmen

Der Fast Fourier Transform (FFT) ist ein effizienter Algoritmus zur Berechnung des DFTs. Die bekannteste Variante wurde im Jahre 1965 von James Cooley und John Wilder Tukey in [5] veröffentlicht. Allerdings wurde er schon 1805 von Carl Friedrich Gauß verwendet um Flugbahnen von Asteroiden zu berechnen [7]. Möglicherweise war ihm das Konzept zu simpel und zu selbstverständlich um es zu veröffentlichen [8].

Wir möchten die Architektur des FFTs nutzen und auf den NTT anwenden. Ziel ist eine Laufzeit von $\mathcal{O}(\log(d)d)$ statt $\mathcal{O}(d^2)$. In diesem Kapitel steht γ für ein Element der Ordnung d in $\mathbb{Z}_p$. Der Übersicht halber lassen wir das $\pmod p$ bei den Berechnungen weg.
Alle FFT-Algorithmen haben gemein, dass sie ‚Teile-und-herrsche'-Verfahren verwenden. Es wird hierbei ausgenutzt, dass beim Berechnen eines NTTs (des NTT-Vektors) viele Rechnungen mehrmals anfallen.

4.1 Cooley-Tukey-Algorithmus

Der Standard FFT ist der Cooley-Tukey-Algorithmus, der auch als Radix-2 bezeichnet wird. Im Cooley-Tukey-Algorithmus wird der NTT rekursiv in halb so große NTTs zerlegt. Sei d durch zwei teilbar:

$$\begin{aligned} NTT(\boldsymbol{x})_i &= \sum_{k=0}^{d-1} x_k \cdot \gamma^{i\cdot k} \\ &= \sum_{k=0}^{(d/2)-1} x_{2k} \cdot \gamma^{i\cdot 2k} \quad + \quad \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot \gamma^{i\cdot(2k+1)} \\ &= \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i\cdot k} \quad + \quad \gamma^i \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i\cdot k} \end{aligned}$$

Anmerkung: γ ist ein Element mit Ordnung d, also ist γ^2 ein Element mit Ordnung $\frac{d}{2}$.
Ist nun $d = 2^e$ lässt sich dieser Schritt e mal wiederholen. Wir erhalten $e+1$ ‚Ebenen'. Sei hierbei E^e die Ebene mit dem NTT mit Ordnung d. In der Ebene E^{e-r} sind dann 2^r NTTs der Ordnung $2^{(e-r)}$. Nun lässt sich $NTT(\boldsymbol{x})_i$ berechnen, indem man nacheinander $E^0, E^1, ..., E^e$ berechnet. Wie groß ist dann der Rechenaufwand um $NTT(\boldsymbol{x})_i$ zu berechnen?

Cooley-Tukey für einen Koeffizienten von NTT Ein NTT wird ersetzt durch zwei halb so große NTTs, eine Addition (der zwei NTTs) und eine Multiplikation (Vorfaktor vor dem zweiten NTT). Es ergeben sich (falls $d > 4$) für $d = 2^e$ insgesamt $e+1$ Ebenen, also e Übergänge von einer Ebene E^r zu E^{r+1} mit insgesamt

$$
\begin{aligned}
&2^0 \cdot (1\,\text{Add.}, 1\,\text{Mult.}) + 2^1 \cdot (1\,\text{Add.}, 1\,\text{Mult.}) + 2^2 \cdot (1\,\text{Add.}, 1\,\text{Mult.}) \\
&+ ... + 2^{e-1} \cdot (1\,\text{Add.}, 1\,\text{Mult.}) \\
&= \sum_{k=0}^{e-1} 2^k (1\,\text{Add.}, 1\,\text{Mult.}) \\
&= \frac{1-2^e}{1-2} (1\,\text{Add.}, 1\,\text{Mult.}) \\
&= \left(2^{e-1}\right) (1\,\text{Add.} + 1\,\text{Mult.}) \\
&= \left(2^{\log_2(d)} - 1\right) (1\,\text{Add.} + 1\,\text{Mult.}) \\
&= (d-1) (1\,\text{Add.} + 1\,\text{Mult.})
\end{aligned}
$$

Additionen und Multiplikationen. Man kann je nach i von NTT_i noch Multiplikationen sparen. Zum einen lässt sich $\gamma^{d/2}$ durch ein Minus ersetzen und zum anderen können Faktoren ganz wegfallen. So fallen für $i = 0$ alle Faktoren weg, während für $i = 1$ nur ein Faktor durch ein (-1) ersetzt werden kann. Interessanter ist allerdings der Fall, in dem alle Koeffizienten berechnet werden sollen.

Ausnutzen der NTT-Perioden Möchte man den ganzen Vektor $NTT(\boldsymbol{x})$ berechnen, so lassen sich viele Rechenschritte sparen. Um nun zu sehen, wo gleiche Zwischenergebnisse anfallen, vergleichen wir für $i \leq (\frac{d}{2} - 1)$ die Aufspaltung von $NTT(\boldsymbol{x})_i$ und $NTT(\boldsymbol{x})_{i+\frac{d}{2}}$:

$$
NTT(\boldsymbol{x})_i = \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i \cdot k} \quad + \quad \gamma^i \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i \cdot k} \tag{1}
$$

$$
\begin{aligned}
NTT(\boldsymbol{x})_{i+\frac{d}{2}} &= \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{(i+\frac{d}{2}) \cdot k} + \gamma^{i+\frac{d}{2}} \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{(i+\frac{d}{2}) \cdot k} \\
&= \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i \cdot k} \quad + \gamma^{i+\frac{d}{2}} \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i \cdot k}
\end{aligned} \tag{2}
$$

Die Unter-NTTs sind exakt dieselben, da γ^2 Ordnung $d/2$ hat. Der Vorfaktor des zweiten Terms lässt sich zu $\gamma^{i+d/2} = \gamma^i \cdot \gamma^{d/2} = \gamma^i \cdot (-1)$ umschreiben. Haben wir die zweite Summe $\sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i \cdot k}$ berechnet, können wir sie mit γ^i multiplizieren und einmal zur ersten Summe $\sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i \cdot k}$ addieren und einmal subtrahieren, um $NTT(\boldsymbol{x})_i$ und $NTT(\boldsymbol{x})_{i+\frac{d}{2}}$ zu erhalten.

Um die Bedeutung der Vorfaktoren besser nachzuvollziehen, spalten wir $NTT(\boldsymbol{x})_i$ noch weiter auf:

$$
\begin{aligned}
NTT(\boldsymbol{x})_i = & \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i \cdot k} + \gamma^i \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i \cdot k} \\
= & \left(\sum_{k=0}^{(d/4)-1} x_{4k} \cdot (\gamma^2)^{i \cdot 2k} + \sum_{k=0}^{(d/4)-1} x_{4k+2} \cdot (\gamma^2)^{i \cdot (2k+1)} \right) \\
& + \gamma^i \left(\sum_{k=0}^{(d/4)-1} x_{4k+1} \cdot (\gamma^2)^{i \cdot (2k)} + \sum_{k=0}^{(d/4)-1} x_{4k+3} \cdot (\gamma^2)^{i \cdot (2k+1)} \right) \\
= & \left(\sum_{k=0}^{(d/4)-1} x_{4k} \cdot (\gamma^4)^{i \cdot k} + \gamma^{2i} \sum_{k=0}^{(d/4)-1} x_{4k+2} \cdot (\gamma^4)^{i \cdot (k)} \right) \\
& + \gamma^i \left(\sum_{k=0}^{(d/4)-1} x_{4k+1} \cdot (\gamma^4)^{i \cdot k} + \gamma^{2i} \sum_{k=0}^{(d/4)-1} x_{4k+3} \cdot (\gamma^4)^{i \cdot k} \right)
\end{aligned}
$$

Analog zu der vorherigen Aufspaltung lässt sich feststellen, dass für $i \leq \frac{d}{4} - 1$ die Unter-NTTs für $i, i + \frac{d}{4}, i + \frac{d}{2}$ und $i + \frac{3d}{4}$ übereinstimmen.

4.2 Konstruktion des Cooley-Tukey-Netzwerks

Wir benötigen hierfür einige Notationen. Sei wie gewohnt $i \in \{0, 1, ..., d - 1\} = \{0, 1, ..., 2^e - 1\}$. Dann sei $i_{②}$ die Binärdarstellung von i mit Länge e, also gegebenenfalls mit führenden Nullen (Bsp.: für $e = 5$ gilt $9_{②} = 01001$). Wir fassen $i_{②}$ auch als Vektor auf (Bsp.: $9_{②}(0) = 0$, $9_{②}(1) = 1$).

Wir definieren für r und $j < 2^r$

$$\boldsymbol{x}_j^r := (x_{j+0 \cdot 2^r}, x_{j+1 \cdot 2^r}, ..., x_{j+(2^{e-r}-1) \cdot 2^r})$$

Beispiele:

$$\boldsymbol{x}_0^0 = (x_0, x_1, ..., x_{d-1})$$
$$\boldsymbol{x}_0^1 = (x_0, x_2, ..., x_{d-2})$$
$$\boldsymbol{x}_1^1 = (x_1, x_3, ..., x_{d-1})$$
$$\boldsymbol{x}_3^{e-1} = (x_3, x_{3+(2^{e-(e-1)}-1) \cdot 2^{e-1}}) = (x_3, x_{3+2^{e-1}}) = (x_3, x_{3+\frac{d}{2}})$$

Wir können also 2^r als Sprungweite und j als Startwert ansehen. Die Zahl der Elemente von $\boldsymbol{x}_j^r$ ist demzufolge 2^{e-r}.

Weiter ist $NTT(\boldsymbol{x}_j^r)$ ein NTT der Ordnung 2^{e-r}, welcher $\gamma^{(2^r)}$ als Element der Ordnung 2^{e-r} verwendet.

Sei $i < 2^{e-(r+1)}$ und $j < 2^r$. Dann folgt analog zu (1) und (2) für das Paar $\left(i, i + 2^{e-(r+1)}\right)$

$$NTT(\boldsymbol{x}_j^r)_i = NTT(\boldsymbol{x}_j^{r+1})_i + \left(\gamma^{(2^r)}\right)^i NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i \tag{3}$$

und

$$\begin{aligned} NTT(\boldsymbol{x}_j^r)_{i+2^{e-(r+1)}} &= NTT(\boldsymbol{x}_j^{r+1})_i + \left(\gamma^{(2^r)}\right)^{i+2^{e-(r+1)}} NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i \\ &= NTT(\boldsymbol{x}_j^{r+1})_i + \gamma^{(2^r)\cdot 2^{e-(r+1)}} \cdot \left(\gamma^{(2^r)}\right)^i NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i \\ &= NTT(\boldsymbol{x}_j^{r+1})_i + \gamma^{(2^{e-1})} \cdot \left(\gamma^{(2^r)}\right)^i NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i \\ &= NTT(\boldsymbol{x}_j^{r+1})_i + \gamma^{\frac{d}{2}} \cdot \left(\gamma^{(2^r)}\right)^i NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i \\ &= NTT(\boldsymbol{x}_j^{r+1})_i - \left(\gamma^{(2^r)}\right)^i NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i. \end{aligned} \tag{4}$$

Falls $NTT(\boldsymbol{x}_j^{r+1})_i$ und $NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i$ gegeben sind, sind die Berechnung des Vorfaktors $\left(\gamma^{(2^r)}\right)^i$ sowie eine Addition und eine Subtraktion nötig um $NTT(\boldsymbol{x}_j^r)_i$ und $NTT(\boldsymbol{x}_j^r)_{i+2^{e-(r+1)}}$ zu berechnen. Die Konstruktion wird mit $e+1$ Ebenen arbeiten, die aber in der Praxis nicht alle nebeneinander existieren, sondern ersetzt werden, so dass immer nur eine Ebene existiert. Es bietet sich an, $NTT(\boldsymbol{x}_j^r)_i$ mit $NTT(\boldsymbol{x}_j^{r+1})_i$ und $NTT(\boldsymbol{x}_j^r)_{i+2^{e-(r+1)}}$ mit $NTT(\boldsymbol{x}_{j+2^r}^{r+1})_i$ zu ersetzen. Dies führt zu:

$$\begin{aligned} E^e &= (NTT(\boldsymbol{x})_0, NTT(\boldsymbol{x})_1, ..., NTT(\boldsymbol{x})_{d-1}) \\ E^e &= (NTT(\boldsymbol{x}_0^e)_0, NTT(\boldsymbol{x}_0^e)_1, ..., NTT(\boldsymbol{x}_0^e)_{d-1}) \\ E^{e-1} &= \big(NTT(\boldsymbol{x}_0^{e-1})_0, NTT(\boldsymbol{x}_0^{e-1})_1, ..., NTT(\boldsymbol{x}_0^{e-1})_{\frac{d}{2}-1}, \\ &\quad NTT(\boldsymbol{x}_1^{e-1})_0, NTT(\boldsymbol{x}_1^{e-1})_1, ..., NTT(\boldsymbol{x}_1^{e-1})_{\frac{d}{2}-1}\big) \\ E^{e-2} &= \big(NTT(\boldsymbol{x}_0^{e-2})_0, NTT(\boldsymbol{x}_0^{e-2})_1, ..., NTT(\boldsymbol{x}_0^{e-1})_{\frac{d}{4}-1}, \\ &\quad NTT(\boldsymbol{x}_2^{e-2})_0, NTT(\boldsymbol{x}_2^{e-2})_1, ..., NTT(\boldsymbol{x}_2^{e-1})_{\frac{d}{4}-1}, \\ &\quad NTT(\boldsymbol{x}_1^{e-2})_0, NTT(\boldsymbol{x}_1^{e-2})_1, ..., NTT(\boldsymbol{x}_1^{e-2})_{\frac{d}{4}-1}, \\ &\quad NTT(\boldsymbol{x}_3^{e-2})_0, NTT(\boldsymbol{x}_3^{e-2})_1, ..., NTT(\boldsymbol{x}_3^{e-1})_{\frac{d}{4}-1}\big) \\ &\vdots \qquad \vdots \end{aligned}$$

Wie sieht dann die Ebene E^0 aus? Sie enthält NTTs der Länge eins, also Elemente von $\boldsymbol{x}$. Um zu sehen welche, muss man die Veränderungen der $\boldsymbol{x}_r^j$ von $NTT(\boldsymbol{x}_j^r)_i$ und $NTT(\boldsymbol{x}_j^r)_{i+2^{e-(r+1)}}$ betrachten. Es wird beim Übergang von Ebene E^{e-r} zu $E^{e-(r+1)}$ das $\boldsymbol{x}_j^r$ von $NTT(\boldsymbol{x}_j^r)_i$ zu $\boldsymbol{x}_j^{r+1}$

und das $\boldsymbol{x}_j^r$ von $NTT(\boldsymbol{x}_j^r)_{i+2^{e-(r+1)}}$ zu $\boldsymbol{x}_{j+2^r}^{r+1}$. Zur Unterscheidbarkeit schreiben wir j_k^{e-r} für den Index von $\boldsymbol{x}$, also das ‚j', an der k-ten Stelle von E^{e-r}. Das heißt ausgehend von $j_k^e = 0$ werden Zweier-Potenzen dazuaddiert. Aus der Bedingung $i < 2^{e-(r+1)}$, gleichbedeutend mit $k \pmod{2^{e-r}} < 2^{e-(r+1)}$ für das Paar $\left(i, i + 2^{e-(r+1)}\right)$, folgt $j_k^{e-(r+1)} = j_k^{e-r}$ und $j_{k+2^{e-(r+1)}}^{e-(r+1)} = j_{k+2^{e-(r+1)}}^{e-r} + 2^r$. Schreiben wir k in Binärschreibweise wie oben beschrieben, so folgt aus $k \pmod{2^{e-r}} < 2^{e-(r+1)}$, dass $k_{②}(r) = 0$ ist und analog, dass $(k + 2^{e-(r+1)})_{②}(r) = 1$ ist. Daraus folgt also, dass genau dann 2^r zu j_k^{e-r} addiert wird, falls $k_{②}(r) = 1$ ist. Es ergibt sich daraus:

$$
\begin{aligned}
&(j_k^0)_{②}(0) = k_{②}(e-1),\\
&(j_k^0)_{②}(1) = k_{②}(e-2),\\
&(j_k^0)_{②}(2) = k_{②}(e-3),\\
&\vdots\\
&(j_k^0)_{②}(e-1) = k_{②}(0)
\end{aligned}
$$

Bezeichnen wir das umgekehrte $k_{②}$ als $rev(k_{②})$, erhalten wir

$$
\begin{aligned}
E^0 &= \left(NTT(\boldsymbol{x}^0_{rev(0_{②})})_0, NTT(\boldsymbol{x}^0_{rev(1_{②})})_0, ..., NTT(\boldsymbol{x}^0_{rev((d-1)_{②})})_0\right)\\
E^0 &= \left(x_{rev(0_{②})}, x_{rev(1_{②})}, ..., x_{rev((d-1)_{②})}\right).
\end{aligned}
$$

Java-Implementierungen des Cooley-Tukey-Algorithmus mit und ohne Permutation der Indizes befinden sich im Anhang A.

4.3 Beispiele von Cooley-Tukey-NTT-Netzwerken

Um den Cooley-Tukey-Algorithmus besser verstehen zu können folgen drei Beispiele. An ihnen wird auch ersichtlich, warum es vorteilhaft ist, die Elemente einer Ebene so wie in Kapitel 4.2 anzuordnen. Dadurch, dass wir in der Ebene E^0 die Indizes i der x_i zu $rev(i_{②})$ permutieren, können viele Operationen *am Platz* durchgeführt werden (engl. *in-place operations*). Die Beispiele sind ausführlich für den Fall $d = 4$ in Tabelle 1, gekürzt für die Fälle $d = 8$ in Tabelle 2 und $d = 16$ in den Tabellen 3 und 4, beschrieben.

E^2	$E_0^2 = NTT(\boldsymbol{x}_0^2)_0$	$E_1^2 = NTT(\boldsymbol{x}_0^2)_1$	$E_2^2 = NTT(\boldsymbol{x}_0^2)_2$	$E_3^2 = NTT(\boldsymbol{x}_0^2)_3$
	$= NTT(\boldsymbol{x}_0^1)_0 + \gamma^0 NTT(\boldsymbol{x}_1^1)_0$	$= NTT(\boldsymbol{x}_0^1)_1 + \gamma^1 NTT(\boldsymbol{x}_1^1)_1$	$= NTT(\boldsymbol{x}_0^1)_0 - \gamma^0 NTT(\boldsymbol{x}_1^1)_0$	$= NTT(\boldsymbol{x}_0^1)_1 - \gamma^1 NTT(\boldsymbol{x}_1^1)_1$
	$= E_0^1 + \gamma^0 E_2^1$	$= E_1^1 + \gamma^1 E_3^1$	$= E_0^1 - \gamma^0 E_2^1$	$= E_1^1 - \gamma^1 E_3^1$
E^1	$E_0^1 = NTT(\boldsymbol{x}_0^1)_0$	$E_0^1 = NTT(\boldsymbol{x}_0^1)_1$	$E_0^1 = NTT(\boldsymbol{x}_1^1)_0$	$E_0^1 = NTT(\boldsymbol{x}_1^1)_1$
	$= NTT(\boldsymbol{x}_0^0)_0 + \gamma^0 NTT(\boldsymbol{x}_2^0)_0$	$= NTT(\boldsymbol{x}_0^0)_0 - \gamma^0 NTT(\boldsymbol{x}_2^0)_0$	$= NTT(\boldsymbol{x}_1^0)_1 + \gamma^0 NTT(\boldsymbol{x}_3^0)_0$	$= NTT(\boldsymbol{x}_1^0)_1 - \gamma^0 NTT(\boldsymbol{x}_3^0)_0$
	$= E_0^0 + \gamma^0 E_1^0$	$= E_0^0 - \gamma^0 E_1^0$	$= E_2^0 + \gamma^0 E_3^0$	$= E_2^0 - \gamma^0 E_3^0$
E^0	$NTT(\boldsymbol{x}^0_{rev(0_{(2)})})_0 = \boldsymbol{x}_0$	$NTT(\boldsymbol{x}^0_{rev(1_{(2)})})_0 = \boldsymbol{x}_2$	$NTT(\boldsymbol{x}^0_{rev(2_{(2)})})_0 = \boldsymbol{x}_1$	$NTT(\boldsymbol{x}^0_{rev(3_{(2)})})_0 = \boldsymbol{x}_3$

Tabelle 1: NTT-Beispiel mit d=4

E^3	$E_0 + E_4$	$E_1 + E_5$	$E_2 + E_6$	$E_3 + E_7$	$E_0 - E_4$	$E_1 - E_5$	$E_2 + E_6$	$E_3 - E_7$
					$\cdot\gamma^0$	$\cdot\gamma^1$	$\cdot\gamma^2$	$\cdot\gamma^3$
E^2	$E_0 + E_2$	$E_1 + E_3$	$E_0 - E_2$	$E_1 - E_3$	$E_4 + E_6$	$E_5 + E_7$	$E_4 - E_7$	$E_6 - E_7$
			$\cdot\gamma^0$	$\cdot\gamma^2$			$\cdot\gamma^0$	$\cdot\gamma^2$
E^1	$E_0 + E_1$	$E_0 - E_1$	$E_2 + E_3$	$E_2 - E_3$	$E_4 + E_5$	$E_4 - E_5$	$E_6 + E_7$	$E_6 - E_7$
E^0	$\boldsymbol{x}_0$		$\boldsymbol{x}_4$	$\boldsymbol{x}_2$	$\boldsymbol{x}_6$	$\boldsymbol{x}_1$	$\boldsymbol{x}_5$	$\boldsymbol{x}_3$	$\boldsymbol{x}_7$

Tabelle 2: NTT mit d=8, gekürzt

E^4	$E_0 + E_8$	$E_1 + E_9$	$E_2 + E_{10}$	$F_E + E11$	$E_4 + E_{12}$	$E_5 + E_{13}$	$E_6 + E_{14}$	$E_7 + E_{15}$
E^3	$E_0 + E_4$	$E_1 + E_5$	$E_2 + E_6$	$E_3 + E7$	$E_0 - E_4$	$E_1 - E_5$	$E_2 - E_6$	$E_3 - E_7$
E^2	$E_0 + E_2$	$E_1 + E_3$	$E_0 - E_2$	$E_1 - E_3$	$\cdot\gamma^0$ $E_4 + E_6$	$\cdot\gamma^2$ $E_5 + E_7$	$\cdot\gamma^4$ $E_4 - E_6$	$\cdot\gamma^6$ $E_5 - E_7$
E^1	$E_0 + E_1$	$E_0 - E_1$	$E_2 + E_3$	$E_2 - E_3$	$E_4 + E_5$	$E_4 - E_5$	$E_6 + E_7$	$E_6 - F_7$
E^0	$\boldsymbol{x}_0$	$\boldsymbol{x}_8$	$\boldsymbol{x}_4$	$\boldsymbol{x}_{12}$	$\boldsymbol{x}_2$	$\boldsymbol{x}_{10}$	$\boldsymbol{x}_6$	$\boldsymbol{x}_{14}$

Tabelle 3: NTT mit d=16, linke Hälfte

E^4	$F_0 - F_8$	$F_1 - F_9$	$F_2 - F_{10}$	$F_3 - F11$	$F_4 - F_{12}$	$F_5 - F_{13}$	$F_6 - F_{14}$	$F_7 - F_{15}$
E^3	$\cdot\gamma^0$ $E_8 + E_{12}$	$\cdot\gamma^1$ $E_9 + E_{13}$	$\cdot\gamma^2$ $E_{10} + E_{14}$	$\cdot\gamma^3$ $E_{11} + E15$	$\cdot\beta^4$ $E_8 - E_{12}$	$\cdot\beta^5$ $E_9 - E_{13}$	$\cdot\beta^6$ $E_{10} - E_{14}$	$\cdot\beta^7$ $E_{11} - E_{15}$
E^2	$E_8 + E_{10}$	$E_9 + E_{11}$	$E_8 - E_{10}$	$E_9 - E_{11}$	$\cdot\gamma^0$ $E_{12} + E_{14}$	$\cdot\gamma^2$ $E_{13} + E_{15}$	$\cdot\gamma^4$ $E_{12} - E_{14}$	$\cdot\gamma^6$ $E_{13} + -E_{15}$
E^1	$E_8 + E_9$	$E_8 - E_9$	$\cdot\gamma^0$ $E_{10} + E_{11}$	$\cdot\gamma^4$ $E_{10} - E_{11}$	$E_{12} + E_{13}$	$E_{12} - E_{13}$	$\cdot\gamma^0$ $E_{14} + E_{15}$	$\cdot\gamma^4$ $E_{14} - E_{15}$
E^0	$\boldsymbol{x}_1$	$\boldsymbol{x}_9$	$\boldsymbol{x}_5$	$\boldsymbol{x}_{13}$	$\boldsymbol{x}_3$	$\boldsymbol{x}_{11}$	$\boldsymbol{x}_7$	$\boldsymbol{x}_{15}$

Tabelle 4: NTT mit d=16, rechte Hälfte

4.4 Rechenaufwand für Cooley-Tukey

Wie groß ist nun also der Aufwand um mit dem Cooley-Tukey-Algorithmus ausgehend von $\boldsymbol{x}$ den Vektor $NTT(\boldsymbol{x})$ zu berechnen? Grob lässt sich schon sagen, dass wir pro Ebene (außer der Startebene E^0) $\mathcal{O}(d)$ Rechenoperationen und somit insgesamt $\mathcal{O}(e \cdot d) = \mathcal{O}(\log(d) \cdot d)$ Rechenoperationen benötigen. Genauer benötigen wir für eine Ebene $\frac{d}{2}$ Additionen, $\frac{d}{2}$ Subtraktionen und höchstens $\frac{d}{2}$ Multiplikationen. Die Berechnung der Vorfaktoren lassen wir außer Acht, da sie für festes d fix und somit vorberechenbar sind. Die Zahl der Multiplikationen hängt davon ab, wie oft der Vorfaktor zu 1 oder (-1) wird. Im Übergang von Ebene E^{e-r} zu $E^{e-(r+1)}$ erhalten wir die Vorfaktoren aus (3) und (4). Für $i = 0, 1, ..., 2^{e-(r+1)} - 1$ und $j = 0, 1, ..., 2^r - 1$ sind dies die (von j unabhängigen) Vorfaktoren $\left(\gamma^{(2^r)}\right)^i$. Da $i < 2^{e-(r+1)}$ ist, kann $\left(\gamma^{(2^r)}\right)^i$ nicht (-1) und nur für $i = 0$ den Wert 1 annehmen. Da j insgesamt 2^r Werte annimmt, erhalten wir 2^r Einsen zwischen E^{e-r} zu $E^{e-(r+1)}$. Ziehen wir in jeder Ebene die Einsermultiplikationen von den $\frac{d}{2}$ Multiplikationen ab, bleiben

$$
\begin{aligned}
&\left(\frac{d}{2} - 2^0\right) + \left(\frac{d}{2} - 2^1\right) + \left(\frac{d}{2} - 2^2\right) + ... + \left(\frac{d}{2} - 2^{e-1}\right) \\
&= (e-1)\frac{d}{2} - (2^0 + 2^1 + ... + 2^{e-2}) \\
&= (e-1)\frac{d}{2} - \sum_{k=0}^{e-2} 2^k \\
&= (e-1)\frac{d}{2} - \frac{1 - 2^{e-1}}{1-2} \\
&= (e-1)\frac{d}{2} - \left(2^{e-1} - 1\right) \\
&= (e-1)\frac{d}{2} - \frac{1}{2}d + 1 \\
&= (e-2)\frac{d}{2} + 1 \\
&= \left(\frac{d}{2}\right)\log_2(d) - d + 1
\end{aligned}
$$

Multiplikationen.

Die Zahl der Rechenoperationen für unsere Beispielgrößen aus Kapitel 4.3 und für $d = 32$ finden sich in Tabelle 5. NTT steht hierbei für die einzelne Berechnung jedes Koeffizienten $NTT(x)_i$ von $NTT(x)$. In dem NTT-Fall erhalten wir für jedes $NTT(\boldsymbol{x})_i = \sum_{k=0}^{d-1} x_k \cdot \gamma^{i \cdot k}$ insgesamt $d-1$ Additionen und bis zu $d-1$ Multiplikationen.

	$d = 4$	$d = 8$	$d = 16$	$d = 32$
NTT Add./Subtr.	12	56	240	992
NTT Mult.	8	44	204	912
Cooley-Tukey-NTT Add./Sub.	8	24	64	160
Cooley-Tukey-NTT Mult.	1	5	17	49

Tabelle 5: Zahl der Rechenoperationen

4.5 Radix-4, Split-Radix

Andere bekannte NTT-Algorithmen für $d = 2^e$ sind z.B. Radix-4 (allgemeiner Radix-2^e) und Split-Radix: Sei d durch 4 teilbar, dann folgt Radix-4 ähnlich wie der Cooley-Tukey-Algorithmus (der Radix-2 entspricht) der Aufspaltung

$$\begin{aligned}
NTT(\boldsymbol{x})_i = & \sum_{k=0}^{(d/4)-1} x_{4k} \cdot \gamma^{i \cdot 4k} + \sum_{k=0}^{(d/4)-1} x_{4k+1} \cdot \gamma^{i \cdot (4k+1)} \\
+ & \sum_{k=0}^{(d/4)-1} x_{4k+2} \cdot \gamma^{i \cdot (4k+2)} + \sum_{k=0}^{(d/4)-1} x_{4k+3} \cdot \gamma^{i \cdot (4k+3)} \\
= & \sum_{k=0}^{(d/4)-1} x_{4k} \cdot \gamma^{i \cdot 4k} + \gamma^i \sum_{k=0}^{(d/4)-1} x_{4k+1} \cdot \gamma^{i \cdot 4k} \\
+ & \gamma^{2i} \sum_{k=0}^{(d/4)-1} x_{4k+2} \cdot \gamma^{i \cdot 4k} + \gamma^{3i} \sum_{k=0}^{(d/4)-1} x_{4k+3} \cdot \gamma^{i \cdot 4k}.
\end{aligned}$$

Der Split-Radix-Algorithmus ist eine Mischung aus dem Radix-2 und Radix-4-Algorithmus:

$$NTT(\boldsymbol{x})_i = \sum_{k=0}^{(d/2)-1} x_{2k} \cdot \gamma^{i \cdot 2k} + \gamma^i \sum_{k=0}^{(d/4)-1} x_{4k+1} \cdot \gamma^{i \cdot 4k} + \gamma^{3i} \sum_{k=0}^{(d/4)-1} x_{4k+3} \cdot \gamma^{i \cdot 4k}$$

Würden wir mit komplexen Zahlen rechnen, würden wir durch ihre Verwendung gegenüber dem Cooley-Tukey-Algorithmus Multiplikationen einsparen. Da wir aber nur ganze (reelle) Zahlen verwenden, sind sie für uns nicht von Nutzen.

4.6 Rechenaufwand für einen 3-er NTT

Bisher hatten wir an d die Bedingung geknüpft, dass es eine Zweierpotenz ist. Deswegen wird hier, stellvertretend auch für andere Primzahlen, der 3-er-NTT gezeigt. Gleichsam dem 2-er NTT spaltet auch der 3-er-NTT auf. Sei $d = 3^e$:

$$
\begin{aligned}
NTT(\boldsymbol{x})_i &= \sum_{k=0}^{(d/3)-1} x_{3k} \cdot (\gamma^3)^{i \cdot k} + \gamma^i \sum_{k=0}^{(d/3)-1} x_{3k+1} \cdot (\gamma^3)^{i \cdot k} \\
&\quad + \gamma^{2i} \sum_{k=0}^{(d/3)-1} x_{3k+2} \cdot (\gamma^3)^{i \cdot k} \\
NTT(\boldsymbol{x})_{i+d/3} &= \sum_{k=0}^{(d/3)-1} x_{3k} \cdot (\gamma^3)^{i \cdot k} + \gamma^{d/3} \cdot \gamma^i \sum_{k=0}^{(d/3)-1} x_{3k+1} \cdot (\gamma^3)^{i \cdot k} \\
&\quad + \gamma^{2d/3} \cdot \gamma^{2i} \sum_{k=0}^{(d/3)-1} x_{3k+2} \cdot (\gamma^3)^{i \cdot k} \\
NTT(\boldsymbol{x})_{i+2d/3} &= \sum_{k=0}^{(d/3)-1} x_{3k} \cdot (\gamma^3)^{i \cdot k} + \gamma^{2d/3} \cdot \gamma^i \sum_{k=0}^{(d/3)-1} x_{3k+1} \cdot (\gamma^3)^{i \cdot k} \\
&\quad + \gamma^{d/3} \cdot \gamma^{2i} \sum_{k=0}^{(d/3)-1} x_{3k+2} \cdot (\gamma^3)^{i \cdot k}
\end{aligned}
$$

Wir benötigen demzufolge insgesamt $2d \cdot \log_3(d)$ Additionen. Da $\log_3(x) = \log_2(x)/\log_2(3)$ sind dies $2d \cdot \frac{1}{\log_2(3)} \log_2(x)$ Additionen. Verglichen mit Cooley-Tukey (mit Add. = Subt.) sind das $\frac{2}{\log_2(3)} \approx 1,26$ mal so viele Additionen.

Da es kein γ mit $\gamma^{(d/3)} = -1$ oder $\gamma^{(2d/3)} = -1$ gibt, fallen deutlich mehr Multiplikationen als bei Cooley-Tukey an. Wir erhalten zwischen den Ebenen E^{e-r} und $E^{e-(r+1)}$ fix $\frac{4}{3}d$ Multiplikationen (die $\gamma^{d/3}$-Mult. und die $\gamma^{2d/3}$-Mult.) plus weitere $\frac{2}{3}d - 2 \cdot 3^r = 2 \cdot 3^r \cdot \left(3^{e-(r+1)} - 1\right)$ Multiplikationen (die γ^i-Mult. und die γ^{2i}-Mult.). Dies führt zu

$$
\begin{aligned}
&\frac{2}{3}d \cdot 2 \cdot \log_3(d) + 2 \cdot 3^0 \cdot \left(3^{e-1} - 1\right) + 2 \cdot 3^1 \cdot \left(3^{e-2} - 1\right) + 2 \cdot 3^2 \cdot \left(3^{e-2} - 1\right) \\
&\quad + ... + 2 \cdot 3^{e-1} \cdot \left(3^{e-(e-1)} - 1\right) \\
&= \frac{2}{3}d \cdot 2 \cdot \log_3(d) + 2 \cdot \Big(3^0 \cdot \left(3^{e-1} - 1\right) + 3^1 \cdot \left(3^{e-2} - 1\right) + 3^2 \cdot \left(3^{e-2} - 1\right) \\
&\quad + ... + 3^{e-1} \cdot \left(3^{e-(e-1)} - 1\right)\Big)
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{2}{3} d \cdot 2 \cdot \log_3(d) + 2 \cdot \Big(\left(3^{e-1} - 3^0\right) + \left(3^{e-1} - 3^1\right) + \left(3^{e-1} - 3^2\right) \\
&\quad + ... + \left(3^{e-1} - 3^{e-1}\right) \Big) \\
&= \frac{2}{3} d \cdot 2 \cdot \log_3(d) + 2 \cdot \left(e \cdot 3^{e-1} - \sum_{k=0}^{e-1} 3^k \right) \\
&= \frac{2}{3} d \cdot 2 \cdot \log_3(d) + 2 \cdot \left(\frac{1}{3} \log_3(d) \cdot d - \frac{3^e - 1}{3 - 1} \right) \\
&= \frac{2}{3} d \cdot 2 \cdot \log_3(d) + 2 \cdot \left(\frac{1}{3} \log_3(d) \cdot d - \frac{d - 1}{2} \right) \\
&= 2 \cdot d \cdot \log_3(d) - d + 1
\end{aligned}
$$

Multiplikationen und somit deutlich mehr als im Cooley-Tukey-Fall.

Möchte man die Indizes der $\boldsymbol{x}$ in der E^0-Ebene eines 3er-NTTs berechnen, so muss man im Gegensatz zu Cooley-Tukey die Indizes im Tenärsystem und nicht im Binärsystem umkehren!

4.7 Primfaktor-NTT

Der Primfaktor-NTT arbeitet durch geschicktes Umsortieren der Indizes. Sei $d = d_1 \cdot d_2$ mit $gcd(d_1, d_2) = 1$, also d_1 und d_2 teilerfremd. Sei $i_1, k_1 \in \{0, ..., d_1 - 1\}$ und $i_2, k_2 \in \{0, ..., d_2 - 1\}$. Dann ist i darstellbar durch

$$i = i_1 (d_2)_{d_1}^{-1} d_2 + i_2 (d_1)_{d_2}^{-1} d_1 \pmod{d}.$$

(Hierbei bezeichnet $(d_2)_{d_1}^{-1}$ die Inverse von d_2 modulo d_1.) Das k lässt sich eindeutig durch

$$k = k_1 d_2 + k_2 d_1 \pmod{d}$$

beschreiben. Zur Vereinfachung:

$$NTT(\boldsymbol{x})_{i_1, i_2} := \left(NTT(\boldsymbol{x})\right)_{i_1 (d_2)_{d_1}^{-1} d_2 + i_2 (d_1)_{d_2}^{-1} d_1}$$

Dann gilt für $NTT(\boldsymbol{x})_{i_1,i_2}$:

$$
\begin{aligned}
&NTT(\boldsymbol{x})_{i_1,i_2} \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{i \cdot k_1 d_2 + k_2 d_1} \right) \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{i \cdot k_1 d_2} \cdot \gamma^{i \cdot k_2 d_1} \right) \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{i \cdot k_2 d_1} \right) \cdot \gamma^{i \cdot k_1 d_2} \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{(i_1 (d_2)_{d_1}^{-1} d_2 + i_2 (d_1)_{d_2}^{-1} d_1) \cdot k_2 d_1} \right) \cdot \gamma^{(i_1 (d_2)_{d_1}^{-1} d_2 + i_2 (d_1)_{d_2}^{-1} d_1) \cdot k_1 d_2} \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{(i_2 (d_1)_{d_2}^{-1} d_1) \cdot k_2 d_1} \right) \cdot \gamma^{(i_1 (d_2)_{d_1}^{-1} d_2) \cdot k_1 d_2} \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot \gamma^{i_2 \cdot k_2 d_1} \right) \cdot \gamma^{i_1 \cdot k_1 d_2} \\
&= \sum_{k_1=0}^{d_1-1} \left(\sum_{k_2=0}^{d_2-1} x_{k_1 d_2 + k_2 d_1} \cdot (\gamma^{d_1})^{i_2 \cdot k_2} \right) \cdot (\gamma^{d_2})^{i_1 \cdot k_1}
\end{aligned}
$$

Ist beispielsweise $d_1 = 2$, so erhält man zwei NTTs der Größe $d_2 = \frac{d}{2}$. Natürlich ist der Primfaktor-NTT auch rekursiv anwendbar und lässt sich mit Cooley-Tukey kombinieren.

5 NTT in mehreren Dimensionen

Es ist sehr einfach einen NTT auf mehrere Dimensionen anzuwenden. Für den zweidimensinalen Fall mit

$$\begin{aligned}
\boldsymbol{x} &\in \mathbb{Z}_p^{d_1 \times d_2} \\
ord(\gamma_{d_1}) &= d_1 \\
ord(\gamma_{d_2}) &= d_2 \\
i &= 0, ..., d_1 - 1 \\
j &= 0, ..., d_2 - 1
\end{aligned}$$

gilt

$$NTT(\boldsymbol{x})_{i,j} = \sum_{k=0}^{d_1-1} \left(\sum_{l=0}^{d_2-1} x_{k,l} \cdot \gamma_{d_2}^{j \cdot l} \right) \gamma_{d_1}^{i \cdot k}.$$

Seien

$$\begin{aligned}
\boldsymbol{x} &\in \mathbb{Z}_p^{d_1 \times ... \times d_e} \\
ord(\gamma_{d_1}) &= d_1 \\
&\vdots \\
ord(\gamma_{d_e}) &= d_e \\
i_1 &= 0, ..., d_1 - 1 \\
&\vdots \\
i_e &= 0, ..., d_e - 1.
\end{aligned}$$

Dann lautet der e-dimensionale NTT dazu

$$NTT(\boldsymbol{x})_{i_1,...,i_e} = \sum_{k_1=0}^{d_1-1} \left(... \left(\sum_{k_e=0}^{d_e-1} x_{k_1,...,k_e} \cdot \gamma_{d_e}^{i_e \cdot k_e} \right) ... \right) \gamma_{d_1}^{i_1 \cdot k_1}.$$

Die Reihenfolge ist dabei beliebig. Die Inverse ist durch

$$\begin{aligned}
&NTT^{-1}(\boldsymbol{y})_{i_1,...,i_e} \\
&= (d_1 \cdot ... \cdot d_e)^{-1} \sum_{k_1=0}^{d_1-1} \left(... \left(\sum_{k_e=0}^{d_e-1} y_{k_1,...,k_e} \cdot \gamma_{d_e}^{-i_e \cdot k_e} \right) ... \right) \gamma_{d_1}^{-i_1 \cdot k_1}
\end{aligned}$$

definiert. Betrachten wir erneut die Aufspaltung beim Cooley-Tukey-Alg.

$$NTT(\boldsymbol{x})_i = \sum_{k=0}^{(d/2)-1} x_{2k} \cdot (\gamma^2)^{i \cdot k} \qquad + \qquad \gamma^i \sum_{k=0}^{(d/2)-1} x_{2k+1} \cdot (\gamma^2)^{i \cdot k}$$

wird deutlich, wie wir von einem eindimensionalen NTT zu einem zweidimensionalen NTT kommen können.

$$\begin{aligned} NTT(\boldsymbol{x})_i &= \sum_{k_1=0}^{2-1} \left(\sum_{k_2=0}^{(d/2)-1} x_{2k_2+k_1} \cdot \gamma^{i\cdot(2k_2+k_1)} \right) \\ &= \sum_{k_1=0}^{2-1} \left(\sum_{k_2=0}^{(d/2)-1} x_{2k_2+k_1} \cdot \gamma^{i\cdot 2k_2} \right) \gamma^{i\cdot k_1} \\ &= \sum_{k_1=0}^{2-1} \left(\sum_{k_2=0}^{(d/2)-1} x_{2k_2+k_1} \cdot (\gamma^2)^{i\cdot k_2} \right) \gamma^{i\cdot k_1} \end{aligned}$$

Nun teilen wir i noch so auf, dass wir zwei NTTs mit jeweils einem ‚Gamma' mit der richtigen Ordnung bekommen.

$$\begin{aligned} NTT(\boldsymbol{x})_{i_1,i_2} &= \sum_{k_1=0}^{2-1} \left(\sum_{k_2=0}^{(d/2)-1} x_{2k_2+k_1} \cdot (\gamma^2)^{(\frac{d}{2}i_1+i_2)\cdot k_2} \right) \gamma^{(\frac{d}{2}i_1+i_2)\cdot k_1} \\ &= \sum_{k_1=0}^{2-1} \left(\gamma^{i_2\cdot k_1} \sum_{k_2=0}^{(d/2)-1} x_{2k_2+k_1} \cdot (\gamma^2)^{i_2\cdot k_2} \right) (\gamma^{\frac{d}{2}})^{i_1\cdot k_1} \end{aligned}$$

Das heißt, der Unterschied zum Cooley-Tukey-Algorithmus besteht in den Vorfaktoren. Setzt man andere Vorfaktoren in das Cooley-Tukey-Netzwerk ein, so lassen sich mit ihm auch mehrdimensionale NTTs berechnen.
Ersetzen wir nochmal einen NTT durch zwei NTTs, erhalten wir einen dreidimensionalen NTT usw.
Möchten wir vom mehrdimesionalem Fall zum eindimensionalen (oder kleinerdimensionalen) Fall, so kann man analog zum folgenden Beispiel vorgehen.

$$NTT(\boldsymbol{x})_{i_1,i_2} = \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} x_{k_1,k_2} \cdot {\gamma_4}^{i_2\cdot k_2} \right) {\gamma_3}^{i_1\cdot k_1}$$

Wir setzen $\gamma = \gamma_3 \cdot \gamma_4^3$. Dann gilt $\gamma^4 = \gamma_3$ und $\gamma^3 = \gamma_4$ und $ord(\gamma) = 12$. Die ersten zwei Behauptungen sind direkt ersichtlich. Da $\gamma^{12} = 1$, gilt $ord(\gamma) \leq 12$. Wir nehmen an, dass $a = ord(\gamma) < 12$. Dann erhalten wir

$$\begin{aligned} \gamma^a &= 1 \\ {\gamma_3}^a \cdot {\gamma_4}^{3a} &= 1 \\ {\gamma_3}^{a \pmod 3} \cdot {\gamma_4}^{3a \pmod 4} &= 1. \end{aligned}$$

Da $a < 12$ ist, muss $a \pmod 3$ oder $3 \cdot a \pmod 4$ ungleich Null sein. Sei $a \pmod 3 \neq 0$, dann erhalten wir mit

$$\left({\gamma_3}^{a \pmod 3} \cdot {\gamma_4}^{3a \pmod 4}\right)^4 = 1$$
$${\gamma_3}^{a \pmod 3} = 1 \quad \text{einen Widerspruch!}$$

Für $3a \pmod 4 \neq 0$, ergibt sich mit

$$\left({\gamma_3}^{a \pmod 3} \cdot {\gamma_4}^{3a \pmod 4}\right)^3 = 1$$
$${\gamma_4}^{9a \pmod 4} = {\gamma_4}^{3a \pmod 4} = 1$$

ebenfalls ein Widerspruch. Somit gilt $ord(\gamma) = 12$. Setzen wir γ ein, erhalten wir

$$\begin{aligned}
NTT(\boldsymbol{x})_{i_1,i_2} &= \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} x_{k_1,k_2} \cdot \gamma^{3i_2 \cdot k_2} \right) \gamma^{4i_1 \cdot k_1} \\
&= \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} x_{k_1,k_2} \cdot \gamma^{3i_2 \cdot k_2 + 4i_1 \cdot k_1} \right) \\
&= \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} \gamma^{i_1 k_2} \cdot \gamma^{-i_1 k_2} \cdot x_{k_1,k_2} \cdot \gamma^{3i_2 \cdot k_2 + 4i_1 \cdot k_1 + 12 i_2 \cdot k_1} \right) \\
&= \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} \gamma^{-i_1 k_2} \cdot x_{k_1,k_2} \cdot \gamma^{i_1 \cdot k_2 + 3i_2 \cdot k_2 + 4i_1 \cdot k_1 + 12 i_2 \cdot k_1} \right) \\
&= \sum_{k_1=0}^{3-1} \left(\sum_{k_2=0}^{4-1} (\gamma^{-i_1 k_2} \cdot x_{k_1,k_2}) \cdot \gamma^{(i_1 + 3i_2)(4k_1 + k_2)} \right).
\end{aligned}$$

Setze nun $i = i_1 + 3i_2$ und $k = 4k_1 + k_2$, dann ist

$$NTT(\boldsymbol{x})_i = \sum_{k=0}^{12-1} (\gamma^{-i_1 k_2} \cdot x_k) \cdot \gamma^{i \cdot k}$$

ein eindimensionaler NTT mit Ordnung 12.

5.1 Multidimensionaler NTT vs. Cooley Tukey

Sei $NTT_{(1)}$ ein eindimensionaler NTT mit Ordnung $d = 2^e$ für ein $e \in \mathbb{N}$ und sei $NTT_{(e)}$ ein e-dimensionaler NTT, dessen e Ordnungen $d_i = 2$ sind. Wir betrachten also die zwei Abbildungen

$$NTT_{(1)} : \mathbb{Z}^d \longrightarrow \mathbb{Z}^d : \boldsymbol{x} \longmapsto NTT_{(1)}(\boldsymbol{x}) \quad \text{mit } \boldsymbol{x} \in \mathbb{Z}^d = \mathbb{Z}^{2^e}$$

und

$$NTT_{(e)} : (\mathbb{Z}^2)^e \longrightarrow (\mathbb{Z}^2)^e : \boldsymbol{z} \longmapsto NTT_{(e)}(\boldsymbol{z}) \quad \text{mit } \boldsymbol{z} \in (\mathbb{Z}^2)^e.$$

Uns interessiert der Rechenaufwand. Für $NTT_{(1)}$ erhalten wir laut Kapitel 4.4 insgesamt $\frac{1}{2}d \cdot \log_2(d)$ Additionen und Subtraktionen und $\left(\frac{d}{2}\right)\log_2(d) - d + 1$ Multiplikationen. Mit $d = 2^e$ erhalten wir $\frac{1}{2}2^e \cdot e = 2^{e-1} \cdot e$ Additionen bzw. Subtraktionen und $(e-2) \cdot \frac{1}{2}2^e + 1 = (e-2) \cdot 2^{e-1} + 1$ Multiplikationen. In $NTT_{(e)}$ verwenden wir verschachtelte NTTs der Ordnung 2. Für einen eindimensionalen NTT der Ordnung 2 benötigen wir lediglich eine Addition und eine Subtraktion. Was gilt für den mehrdimensionalen Fall? Hier ist schon gegeben, wie wir vom e-dimensionalen zu zwei $(e-1)$-dimensionalenen Fällen gelangen:

$$NTT(\boldsymbol{x})_{(e)_{i_e,i_{e-1},\ldots,i_1}}$$

$$= \sum_{k_e=0}^{1} \left(\sum_{k_{e-1}=0}^{1} \left(\ldots \left(\sum_{k_1=0}^{1} x_{k_e,k_{e-1},\ldots,k_1} \cdot \gamma^{i_1 \cdot k_1} \right) \ldots \right) \cdot \gamma^{i_{e-1} \cdot k_{e-1}} \right) \cdot \gamma^{i_e \cdot k_e}$$

$$= \left(\sum_{k_{e-1}=0}^{1} \left(\ldots \left(\sum_{k_1=0}^{1} x_{0,k_{e-1},\ldots,k_1} \cdot \gamma^{i_1 \cdot k_1} \right) \ldots \right) \cdot \gamma^{i_{e-1} \cdot k_{e-1}} \right) \cdot \gamma^0$$

$$+ \left(\sum_{k_{e-1}=0}^{1} \left(\ldots \left(\sum_{k_1=0}^{1} x_{1,k_{e-1},\ldots,k_1} \cdot \gamma^{i_1 \cdot k_1} \right) \ldots \right) \cdot \gamma^{i_{e-1} \cdot k_{e-1}} \right) \cdot \gamma^{i_e}$$

$$= NTT(\boldsymbol{x}|x_1 = 0)_{(e-1)_{i_{e-1},\ldots,i_1}} + \gamma^{i_e} \cdot NTT(\boldsymbol{x}|x_1 = 1)_{(e-1)_{i_{e-1},\ldots,i_1}}$$

Man erkennt, dass sich die Berechnungen von $NTT(\boldsymbol{x})_{(e)}$ für $i_e = 0$ und $i_e = 1$ im Vergleich zum Cooley-Tukey-Fall nur durch einen Vorfaktor unterscheiden. Da wir hier mit $\gamma = -1$ arbeiten können, ändert sich also lediglich das Vorzeichen des zweiten Terms. Wir erhalten demnach die gleiche Anzahl von Additionen und Subtraktionen wie bei Cooley-Tukey, allerdings keine Multiplikationen.

6 NTT-Ebenen vorbechnen

Bisher wurde noch nicht auf die besondere Form der SWIFFT-Eingabepolynome eingegangen. Um Ajtais Theorem 1.5 ausnutzen zu können, sind alle Koeffizienten der Eingabepolynome 0 oder 1. Dies ermöglicht es Teilschritte auf Kosten von Speicherplatz vorzuberechnen.

Verkleinern von NTTs durch Vorberechnungen Seien wie gewohnt $p \in \mathbb{N}$ und ω so gewählt, dass $ord(\omega) = 2d \pmod p$ und $d^{-1} \pmod p$ existiert. Sei $\gamma = \omega^2$ und $\boldsymbol{x} = (x_0, ..., x_{d-1}) \in \mathbb{Z}_2^d$ und $\hat{\boldsymbol{x}} = (\hat{x}_0, ..., \hat{x}_{d-1}) = (\omega^0 x_0, ..., \omega^{d-1} x_{d-1}) \in \mathbb{Z}_p^d$ und somit $NTT(\hat{\boldsymbol{x}})$ ein NTT mit Ordnung d. Seien a und b Teiler von d. Dann sind durch

$$\begin{aligned} &i = i_0 + ai_1 \text{ mit } i_0 \in \{0, 1, ..., a-1\} \text{ und } i_1 \in \{0, 1, ..., d/a - 1\} \text{ und} \\ &k = k_0 + bk_1 \text{ mit } k_0 \in \{0, 1, ..., b-1\} \text{ und } k_1 \in \{0, 1, ..., d/b - 1\} \end{aligned}$$

alternative Darstellungen von i und k gegeben. Gerechnet wird wieder modulo p. Wir ersetzen i und k und erhalten

$$\begin{aligned} NTT(\hat{\boldsymbol{x}})_i &= \sum_{k=0}^{d-1} \hat{x}_k \cdot \gamma^{i \cdot k} \\ &= \sum_{k=0}^{d-1} (x_k \cdot \omega^k) \cdot (\omega^2)^{i \cdot k} \\ &= \sum_{k=0}^{d-1} x_k \cdot \omega^{(2i+1) \cdot k} \\ NTT(\hat{\boldsymbol{x}})_{i_0 + ai_1} &= \sum_{k=0}^{d-1} x_{k_0 + bk_1} \cdot \omega^{(2(i_0 + ai_1)+1) \cdot (k_0 + bk_1)} \\ &= \sum_{k_0=0}^{b-1} \omega^{(2(i_0 + ai_1)+1) \cdot k_0} \sum_{k_1=0}^{d/b-1} x_{k_0 + bk_1} \cdot \omega^{(2(i_0 + ai_1)+1) \cdot (bk_1)} \\ &= \sum_{k_0=0}^{b-1} (\omega^{2a})^{i_1 \cdot k_0} \cdot \underbrace{\omega^{(2i_0+1) \cdot k_0}}_{=m_{k_0, i_0}} \cdot \underbrace{\sum_{k_1=0}^{d/b-1} x_{k_0 + bk_1} \cdot \omega^{(2(i_0 + ai_1)+1) \cdot (bk_1)}}_{=t_{k_0, i_0, i_1}}. \end{aligned}$$

Die Bezeichnungen m und t sind hierbei wie von den SWIFFT-Autoren gewählt. Nun kann m_{k_0, i_0} nur noch $a \cdot b$ Werte annehmen. Allerdings hängt t noch von drei Variablen und x ab. Da $x_{k_0 + bk_1}$ nur 0 oder 1 sein kann, kann t noch $a \cdot d/a \cdot 2^b = d \cdot 2^{d/b}$ Werte annehmen. Schreibt man noch etwas um, erhält man

$$NTT(\boldsymbol{x})_{i_0+ai_1} = \sum_{k_0=0}^{b-1} (\omega^{2a})^{(i_1 \cdot k_0)} \cdot m_{k_0,i_0} \cdot \underbrace{\sum_{k_1=0}^{d/b-1} x_{k_0+bk_1} \cdot \omega^{((2i_0+1)\cdot bk_1)+2ai_1 \cdot bk_1}}_{=t_{k_0,i_0,i_1}}.$$

Seien a und b so gewählt, dass $a \cdot b$ ein Vielfaches von d ist. Dann ist t nicht mehr von i_1 abhängig, da $ord(\omega) = 2d$. Somit kann t nur noch $a \cdot 2^{d/b}$ Werte annehmen.

$$\begin{aligned} NTT(\boldsymbol{x})_{i_0+ai_1} &= \sum_{k_0=0}^{b-1} (\omega^{2a})^{i_1 \cdot k_0} \cdot m_{k_0,i_0} \cdot \underbrace{\sum_{k_1=0}^{d/b-1} x_{k_0+bk_1} \cdot \omega^{((2i_0+1)\cdot bk_1)}}_{=t_{k_0,i_0}} \\ &= \sum_{k_0=0}^{b-1} (\omega^{2a})^{i_1 \cdot k_0} \cdot m_{k_0,i_0} \cdot t_{k_0,i_0} \end{aligned}$$

Nun ähnelt $NTT(\boldsymbol{x})_{i_0+ai_1}$ mehreren kleineren NTTs mit Größe b. Wir benötigen aber noch ein Element mit Ordnung b. Nehmen wir an, dass $ab = d$ ist, dann erhalten wir mit

$$\begin{aligned} NTT(\boldsymbol{x})_{i_0+ai_1} &= \sum_{k_0=0}^{b-1} (m_{k_0,i_0} \cdot t_{k_0,i_0}) \cdot (\omega^{2a})^{(i_1 \cdot k_0)} \\ NTT(\boldsymbol{x})_{i_0+ai_1} &= \sum_{k_0=0}^{b-1} (m_{k_0,i_0} \cdot t_{k_0,i_0}) \cdot (\omega^{2\frac{d}{b}})^{(i_1 \cdot k_0)} \end{aligned} \tag{5}$$

für jedes i_0 einen NTT mit Größe b.
Zusammenfassend: Wir möchten a und b so wählen, dass $a \cdot b$ Vielfaches von d und $a \cdot b = d$ ist, also langt uns die Bedingung $a \cdot b = d$.
Wir erhalten je nach Wahl von a und b einen Speicherplatzbedarf von Ordnung $\mathcal{O}(a \cdot 2^a)$ für die t und von Ordnung $\mathcal{O}(d)$ für die m und müssen a NTTs mit Ordnung b berechnen.

Im Anhang A befindet sich mit `precompNTT` eine Implementierung eines NTTs mit Vorberechnungen.

Die Aufspaltung in SWIFFTX: In SWIFFTX wurden für SWIFFT die Parameter $a = b = 8$ gewählt:

$$NTT(\boldsymbol{x})_{i_0+8i_1} = \sum_{k_0=0}^{8-1} (m_{k_0,i_0} \cdot t_{k_0,i_0}) \cdot (\omega^{16})^{i_1 \cdot k_0}$$

In diesem Fall sind also 256 Werte für m und $8 \cdot 2^8 = 2048$ Werte für t vorzuberechnen. Hier wird auch der Grund für die Wahl von ω ersichtlich. Für $\omega = 42$ gilt $\omega^{16} = 4 \pmod{257}$. Die Multiplikationen in den Cooley-Tukey-Netzwerken entsprechen dann lediglich Bitshifts um zwei Stellen.

Permutation der Indizes Sei $d = a \cdot b$, wobei $b = 2^e$ eine Zweierpotenz ist und wir möchten unseren NTT mit der Hilfe von Vorberechnungen berechnen. Außerdem möchten wir, dass die Rechnungen am Platz geschehen, so dass wir weder während noch nach unserer Berechnung der äußeren NTTs Permutationen vornehmen müssen. Ziel ist es, alle Permutationen, die für die äußeren NTTs notwendig sind, in den Vorberechnungen vorzunehmen. Der Formel (5) entnehmen wir, dass dazu k_0 permutiert werden muss. Zur Erinnerung: $k = k_0 + bk_1$ wobei $k_0 \in \{0, 1, ..., b-1\}$ und $k_1 \in \{0, 1, ..., a-1\}$ sind. Für den Fall, dass a eine Zweierpotenz ist, heißt das, dass wir, k im Binärsystem geschrieben, die letzten e Stellen von k umkehren müssen. Es muss außerdem darauf geachtet werden, dass auch das k_0 der Faktoren $(\omega^{2\frac{d}{b}})^{(i_1 \cdot k_0)}$ umgekehrt wird. Für den Fall, dass a keine Zweierpotenz ist, ist die Permutation nicht so einfach möglich, da die Bitdarstellungen von k_0 und k_1 konkateniert nicht k ergeben. Eine Implementierung (`reverseParts`) und eine Beschreibung der dann benötigten Permutation ist im Anhang A.

6.1 Geschwindigkeitsvergleich

In den Tabellen 6 und 7 ist ein Geschwindigkeitsvergleich verschiedener NTT-Algorithmen. Testrechner war ein Notebook mit Intel® Core 2 Duo Prozessor T7100 (2MB Cache, 1.80 GHz, 800 MHz FSB) und 2 GB Arbeitsspeicher.

Für jedes d bekamen die Algorithmen die gleiche Eingabe $x \in \{0, 1\}^d$. Jeder Algorithmus musste das Ergebnis $NTT(\hat{x})$ mit $\hat{x} = (x_0\omega^0, x_1\omega^1, ..., x_{d-1}\omega^{d-1}$ insgesamt 10 000 mal berechnen. Dabei wurden Berechnungen die für jedes x gleich wären (z.B. Potenzen von ω oder Vorberechnungen wie in (5)) nur einmal durchgeführt.

Ordnung d :	**64**		**96**		**128**	
ω, γ, p :	$42, 222, 257$		$52, 2, 193$		$3, 9, 257$	
	ms	$Vorb.$	ms	$Vorb.$	ms	$Vorb.$
NTT	652		1872		2595	
Cooley-Tukey-NTT	81				171	
Precomp-NTT	150	4	240	6	262	4

Tabelle 6: Geschwindigkeitsvergleich Teil 1

Ordnung d :	**160**		**192**		**256**	
ω, γ, p :	$7, 49, 641$		$2, 4, 769$		$62, 3844, 7681$	
	ms	$Vorb.$	ms	$Vorb.$	ms	$Vorb.$
NTT	4582		6524		10241	
Cooley-Tukey-NTT					431	
Precomp-NTT	331	5	421	$3,6$	641	8

Tabelle 7: Geschwindigkeitsvergleich Teil 2

Im Falle des Cooley-Tukey-Algorithmus mit Vorberechnungen wurde die Vorberechnungsgröße mit angegeben. Sie entspricht der Zahl b in (5). Es wurde jeweils die Vorberechnungsgröße gewählt, für die der Algorithmus am schnellsten war.

Wie man erkennt, sind die Algorithmen, die das Cooley-Tukey-Netzwerk benutzen deutlich schneller als die simple Berechnung des NTT. Auch ist der Zuwachs der benötigten Zeit des einfachen Algorithmus NTT deutlich größer. Braucht der NTT-Algorithmus für $d = 64$ noch ca. acht mal länger als der Cooley-Tukey-NTT, braucht er für $d = 256$ schon ca. 24 mal so lang. Dagegen ist der Zeitzuwachs des Cooley-Tukey-NTTs mit Vorberechnungen langsamer als der des Cooley-Tukey-NTTs. Auch ist der Cooley-Tukey mit Vorberechnungen deutlich flexibler, da er nicht darauf angewiesen ist, dass d eine Zweierpotenz ist.

7 Tests auf Zufälligkeit

Eine der wichtigen Eigenschaften von in der Kryptographie verwendeten Hashfunktionen ist die Ununterscheidbarkeit zu einer (deterministischen) gleichverteilten Zufallsvariable. Tests können nicht *beweisen*, ob die Ausgaben eines Zufallgenerators fehlerfrei sind! Auch können sie echte Zufälle für ‚unecht' halten. So hätten sicherlich die meisten Zufallstests die Lottozahlen der israelischen Lotterie der Monate September und Oktober 2010 als wenig zufällig angesehen, nachdem am 21. September und 16. Oktober genau die gleichen Zahlen gezogen wurden [16]. Zufallstests können jedoch Auffälligkeiten und Unregelmäßigkeiten aufdecken.

In dieser Arbeit wird nicht zwischen ‚Zufallsgenerator' und ‚Pseudozufallsgenerator' unterschieden!

7.1 Wahrscheinlichkeitstheorie

Um die Zufallstests beschreiben, verstehen und verändern zu können benötigen wir etwas Wahrscheinlichkeitstheorie.

Definition 7.1 (Zufallsvariable)**.** Eine Zufallsvariable ist eine Abbildung, die Ereignissen eines Zufallsexperiments Werte zuordnet.

Definition 7.2 (Verteilungsfunktion)**.** Sei X eine Zufallsvariable und sei $x \in \mathbb{R}$. Bezeichne die Funktion P die Wahrscheinlichkeit, dann heißt die Funktion F mit

$$F(x) = P(X \leq x)$$

die Verteilungsfunktion von X. Sie gibt an, wie wahrscheinlich es ist, dass X kleiner als x ist.

Definition 7.3 (Empirische Verteilungsfunktion)**.** Sei X eine Zufallsvariable und $x_1, x_2, ..., x_n$ eine Stichprobe von X. Für $x \in \mathbb{R}$ bezeichne $H_n(x)$ die Anzahl der Werte aus der Stichprobe, die kleiner oder gleich x sind. Dann ist

$$F_n(x) = \frac{1}{n} H_n(x)$$

die empirische Verteilungsfunktion dieser Stichprobe [18].

Definition 7.4 (Erwartungswert)**.** Der Erwartungswert $E[X]$ einer Zufallsvariablen X ist der Wert, der sich bei häufigem Wiederholen des der Zufallsvariablen zugrunde liegenden Experiments als Mittelwert ergeben sollte.

Definition 7.5 (Binomial-Verteilung)**.** Sei X eine Zufallsvariable mit Wertebereich $0, 1, ..., n$. Falls

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \quad \text{für k = 0,1,...,n,}$$

heißt X binomialverteilt mit den Parametern p und n. Ihr Erwartungswert ist pn.

Definition 7.6 (Poisson-Verteilung)**.** Sei $\lambda > 0$ und X eine Zufallsvariable mit Wertebereich $\mathbb{N}$. Falls

$$P(X = k) = \frac{\lambda^k}{k!} e^{-k} \quad \text{für k = 0,1,2,...,}$$

heißt X Poisson-verteilt mit Parameter λ. Der Erwartungswert einer Poisson-verteilten Zufallsvariable mit Parameter λ ist λ [18],[21].

Theorem 7.7 (Grenzwertsatz von Poisson)**.** *Es sei $\lambda \geq 0$ und $(p_n)_{n\in\mathbb{N}}$ eine Folge im Intervall $[0, 1]$. Falls*

$$\lim_{n\to\infty} np_n = \lambda,$$

gilt

$$\lim_{n\to\infty} \binom{n}{k} p_n^k (1 - p_n)^{n-k} = \frac{\lambda^k}{k!} e^{-\lambda}$$

für $k \in \mathbb{N}$. Für große n und kleine p lässt sich also die Binomialverteilung durch die Poisson-Verteilung approximieren. Eine Abschätzung des Fehlers ist z.B. durch

$$\frac{1}{2} \sum_{k=0}^{\infty} \left| \binom{n}{k} p_n^k (1 - p_n)^{n-k} - \frac{\lambda^k}{k!} e^{-\lambda} \right| \leq \frac{\lambda}{n} \min(2, \lambda)$$

gegeben [18].

Definition 7.8 (Chi-Quadrat-Verteilung)**.** Sei X_n die Summe der Quadrate von n voneinander unabhängigen, standardnormalverteilten Zufallsvariablen Z_i.

$$X_n = Z_1^2 + Z_2^2 + ... + Z_n^2.$$

Die zugehörige Verteilungsfunktion ist

$$F_{\chi_n^2}(x) = \frac{1}{\Gamma(\frac{n}{2})} \Gamma(\frac{n}{2}, \frac{x}{2}).$$

Dabei ist $\Gamma(x)$ die Eulersche Gammafunktion und $\Gamma(a, x)$ die unvollständige Eulersche Gammafunktion der unteren Grenze. Man spricht bei F auch von einer χ^2-Verteilung mit n Freiheitsgraden [18].

Definition 7.9 (Chi-Quadrat-Anpassungstest)**.** Sei X eine Zufallsvariable und F ihre Verteilung. Sei F_0 eine bekannte Verteilungsfunktion, dann ist der χ^2-Anpassungstest ein Signifikanztest zur Prüfung der Hypothese

$$H_0 : F = F_0$$

mit Alternative

$$H_0 : F \neq F_0.$$

Sei $x_1, x_2, ..., x_n$ eine Stichprobe von X. Seien m Kategorien gegeben und sei n_i die Zahl der x_i, die in die i-te Kategorie fallen. Sei p_j die Wahrscheinlichkeit, dass ein x_i zur Kategorie j gehört. Dann ist np_j die erwartete Zahl von x_i in der Kategorie j. Die χ^2-Testgröße ist

$$\chi_n^2 = \sum_{k=0}^{m} \frac{(n_k - np_k)^2}{np_k}.$$

Die Testgröße ist für ausreichend große n annähernd χ^2-verteilt mit Freiheitsgrad $(n-1)$ [21],[18].

Theorem 7.10 (Hauptsatz der mathematischen Statistik)**.** *Die Differenz*

$$D_n := \sup_{x \in \mathbb{R}} \|(F_n(x) - F(x))\|$$

geht für $n \to \infty$ mit Wahrscheinlichkeit 1 gegen 0.

Definition 7.11 (Kolmogorow-Verteilung)**.** Die Kolmogorow-Verteilung ist durch

$$K(x) := \begin{cases} \sum_{i=-\infty}^{\infty} (-1)^i e^{-2i^2x^2} & \text{für x} > 0 \\ 0 & \text{für x} \leq 0 \end{cases}$$

definiert

Kolmogorow hat gezeigt, dass für eine beliebige stetige Verteilungsfunktion F die Größe $T_n = \sqrt{n} D_n$ gegen die Kolmogorow-Verteilung konvergiert.

Definition 7.12 (Kolmogorow-Smirnow-Test)**.** Der Kolmogorow-Smirnow-Test (KS-Test) ist ein Hypothesentest zum Testen, ob die Verteilungsfunktionen zweier Zufallsvariabler übereinstimmen, oder ob die Verteilungsfunktion einer Zufallsvariablen einer angenommenen Wahrscheinlichkeitsverteilung gleicht. Sei X eine Zufallsvariable mit der Verteilungsfunktionen F. Sei F_0 die angenommene Verteilungsfunktion. Die Nullhypothese lautet dann

$$H_0 : F_X = F$$

und die Alternativhypothese

$$H_1 : F_X \neq F$$

Falls H_0 zutrifft, strebt $T_n = \sqrt{n}D_n$ gegen die Kolmogorow-Verteilung. Sei $P(K \leq K_\alpha) = 1 - \alpha$. Die Hypothese H_0 wird dann mit Signifikanzniveau α verworfen, falls $T_n > K_\alpha$ gilt, ansonsten wird sie angenommen [18].

Definition 7.13 (Kuiper-Test)**.** Der Kuiper-Test nimmt gegenüber dem KS-Test nur eine kleine Veränderung vor. Während der KS-Test die maximale Abweichung von $F(x)$ und $F_n(x)$, also die Teststatistik D_n und somit entweder $D_n^+ := \sup_{x\in\mathbb{R}}(F_n(x) - F(x))$ oder $D_n^- := \sup_{x\in\mathbb{R}}(F(x) - F_n(x))$ betrachtet, addiert der Kuiper-Test beide Abweichungen von $F(x)$ über und unter $F_n(x)$:

$$V_n := \sup_{x\in\mathbb{R}}(F_n(x) - F(x)) + D_n := \sup_{x\in\mathbb{R}}(F(x) - F_n(x)).$$

Der kritische Wert zum Signifikanzniveau α, bei dem $H_0 : F_X = F_0$ verworfen wird, wird Tabellen entnommen [9].

Definition 7.14 (Varianz)**.** Sei X eine Zufallsvariable mit $E[X^2] < \infty$. Dann ist

$$\mathrm{Var}(X) := E((X - E[X])^2)$$

die Varianz von X.

Definition 7.15 (Kovarianz)**.** Seien X und Y zwei Zufallsvariablen für die die Erwartungswerte $E[X], E[Y]$ und $E[XY]$ existieren. Dann ist

$$Cov(X, Y) := E\left[(X - E[X])(Y - E[Y])\right]$$

die Kovarianz von X und Y.

Theorem 7.16 (Verschiebungssatz von Steiner)**.** *Mit Steiners Verschiebungssatz lässt sich die Kovarianz durch Erwartungswerte ausdrücken:*

$$Cov(X, Y) = E[X \cdot Y] - E[X] \cdot E[Y]$$

Definition 7.17 (Kovarianzmatrix)**.** Die Kovarianzmatrix enthält die paarweisen Kovarianzen eines Zufallsvektors. Ist $\boldsymbol{X} = X_1, X_2, ..., X_n$ ein Zufallsvektor, für den die Varianzen $\mathrm{Var}(X_i)$ existieren, ist

$$Cov(\boldsymbol{X}) = \begin{bmatrix} Cov(X_1, X_1) & \cdots & Cov(X_1, X_n) \\ \vdots & \ddots & \vdots \\ Cov(X_n, X_1) & \cdots & Cov(X_n, X_n) \end{bmatrix}$$

die Kovarianzmatrix von $\boldsymbol{X}$. Sie ist symmetrisch und positiv semidefinit.

Definition 7.18 (Pseudoinverse)**.** Die Pseudoinverse einer Matrix A ist die Verallgemeinerng einer inversen Matrix für singuläre und nichtquadratische Matrizen. Eine Matrix B ist eine Pseudoinverse von A, falls

$$ABA = A \quad \text{und} \quad BAB = B.$$

7.2 Testbeschreibungen

Die Grundlage für die Testbeschreibungen stammt aus den Dokumentationen der Testsuiten *diehard* [11] und *dieharder* [4]. Tests, deren Namen mit ‚RGB' anfangen, sind von Robert G. Brown entworfen und aus *dieharder*. Fangen sie mit ‚Diehard' an, sind sie von George Marsaglia. Brown programmierte die meisten *diehard*-Tests nach und so befinden sie sich in der Regel in *diehard* und *dieharder*. ‚STS'-Tests sind Tests aus der Statistical Test Suite von NIST, die Brown für *dieharder* neu programmierte.

Anwendung des Kuiper-KS-Tests in der *dieharder*-Testsuite Die nachfolgenden Testbeschreibungen beschreiben jeweils einen Testdurchgang. Die Anzahl der Testdurchläufe ist aber wählbar und als Standard sind 100 gewählt. Auf diese 100 Testergebnisse wird dann der Kuiper-KS-Test angewandt.
Vorab ein Beispiel zur Erklärung des Kuiper-KS-Tests:
Im Diehard-Bitstream-Test 7.2.7 wird erwartet, dass die Anzahl j der fehlenden Wörter normalverteilt ist mit Erwartungswert 141909 und Standardabweichung 428. Damit ist j die Realisierung einer Zufallsvariablen mit der Verteilungsfunktion $F_{\mathcal{N}(141909,428)}(j) \in [0,1]$. Führen wir den Diehard Bitstream 100 mal durch, erhalten wir durch Anwenden der Verteilungsfunktion auf die j_i die 100 p_i-Werte $p_1 = F_{\mathcal{N}(141909,428)}(j_1)$, $p_2 = F_{\mathcal{N}(141909,428)}(j_2), ..., p_{100} = F_{\mathcal{N}(141909,428)}(j_{100})$. Falls die Annahme über die Normalverteilung stimmt, liegen diese p_i gleichverteilt im Intervall $[0,1]$. Das heißt, wir interpretieren die p_i als eine Stichprobe einer gleichverteilten Zufallsvariable und wenden den Kuiper-KS-Test auf sie an.

Für eine ausreichend große Anzahl (laut Brown mindestens 100) von Wiederholungen erhalten wir so ein aussagekräftiges Ergebnis. In *diehard* war die Zahl der Wiederholungen nicht variabel und der anschließende KS-Test arbeitete oft mit lediglich ein bis zwanzig p-Werten. So gibt es mehrere Zufallsgeneratoren, die in *diehard* bestanden, aber in *dieharder* durchfielen, da hier durch eine größere Zahl von p-Werten offensichtlich wurde, dass die p-Werte nicht gleichverteilt waren. [3]

Den Testbeschreibungen vorgreifend rufen wir *dieharder* mit `./dieharder -g 11 -r 3 -n 3` auf und erhalten folgende Ausgabe:

```
#=================================================================
#                 RGB Bit Distribution Test
# Accumulates the frequencies of all n-tuples of bits in a list
# of random integers and compares the distribution thus generated
# with the theoretical (binomial) histogram, forming chisq and the
# associated p-value.  In this test n-tuples are selected without
# WITHOUT overlap (e.g. 01|10|10|01|11|00|01|10) so the samples
# are independent.  Every other sample is offset modulus of the
# sample index and ntuple_max.
#
#                       Run Details
# Random number generator tested: minstd
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
#################################################################
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     14|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |****|****|    |    |
#     12|    |    |    |    |****|    |****|****|    |    |
#       |    |    |****|    |****|****|****|****|    |    |
#     10|    |    |****|    |****|****|****|****|    |****|
#       |    |    |****|    |****|****|****|****|    |****|
#      8|****|****|****|****|****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|    |****|
#      6|****|****|****|****|****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
Kuiper KS: p = 0.23228283
Assessment: PASSED at > 5% for RGB Bit Distribution Test
```

Die Ausgabe ist Resultat des RGB-Bit-Distribution-Tests für 3-Tupel, angewandt auf *minstd*. Der Zufallsgenerator *minstd* ist u.A. die Zufallsfunktion von MATLAB.

7.2.1 RGB Bit Persistance Test

Dies ist ein sehr einfacher Test. Er erhält als Eingabe 256 Zahlen mit 32 Bit. Der Test erstellt aus ihnen eine Maske in der er festhält, an welchen Stellen sich ein Bit nicht verändert, also in allen 256 Zahlen gleich ist.

7.2.2 RGB Bit Distribution Test

Gegeben sei eine Eingabe von n Bit. Der Test zählt, wie oft sich nicht überlappende, also voneinander unabhängige t-Tupel auftreten. So gibt es beispielsweise für $t = 3$ acht verschiedene 3-Tupel. Jedes Tupel sollte in diesem Fall eine Wahrscheinlichkeit von $\frac{1}{8}$ haben und binomialverteilt sein. Auf die Anzahlen der t-Tupel wird der Chi-Quadrat-Test angewandt.

7.2.3 Diehard Birthday-Spacings-Test

Der Birthday-Spacings-Test ist eine Erweiterung des Iterated-Spacings-Tests, beschrieben z.B. in [10]. Sind n Geburtstage eines Jahres mit m Tagen zufällig gewählt und sortiert, sollte die Zahl der übereinstimmenden Abstände asymptotisch Poisson-verteilt sein mit Parameter $\lambda = \frac{n^3}{4m}$. Vorgehen: Der zu testende Generator produziert n b-Bit-Zahlen $I_1, I_2, ..., I_n$ (Geburtstage). Die Zahlen werden dann so sortiert, dass $I_{1'} \leq I_{2'} \leq ... \leq I_{n'}$. Das Jahr hat dann $m = 2^b$ Tage. Sei X die Anzahl der Werte, die unter den Abständen $I_{1'} - 0, I_{2'} - I_{1'}, ..., I_{n'} - I_{(n-1)'}$ mehrfach vorkommen. X sollte Poisson-verteilt sein mit Parameter $\lambda = \frac{n^3}{4m}$. Siehe auch [13].

7.2.4 Diehard Overlapping 5-Permutations-Test

(Operm5-Test). Gegeben n 32-Bit Zahlen $I_1, I_2, ..., I_n$. Sortiert man fünf aufeinanderfolgende Zahlen, kann es 5! Anordnungen geben. Wir erhalten also für $J_1 = (I_1, I_2, I_3, I_4, I_5)$, $J_2 = (I_2, I_3, I_4, I_5, I_6), ...$, jeweils eine Sortierung von 120 möglichen Sortierungen. Das Auftreten dieser Sortierungen wird gezählt und überprüft. Da die J_i sich überlappen, müssen die Abhängigkeiten der sich überlappenden Sortierungen mit einbezogen werden. Dies wird mithilfe der Kovarianzmatrix und einem angepassten Chi-Quadrat-Test mit geringerem Freiheitsgrad bewerkstelligt. Der Test existiert in *dieharder* auch in einer nichtüberlappenden Version. In diesem Fall ist ein normaler Chi-Quadrat-Test möglich.

Anmerkung: Die in den *dieharder*-Versionen 3.29.4beta und früher (einschließlich aller 2.xx.x-Versionen) verwendete Pseudoinverse ist falsch, siehe Kapitel 8. Ab Version 3.31.0 verwendet *dieharder* die Pseudoinverse von der im Zuge dieser Arbeit berechneten Kovarianzmatrix.

7.2.5 Diehard 32x32 Binary Rank Test

Eingabe sind 32-Bit Zufallszahlen. Der Test nimmt die 32 Bit jeder Zufallszahl und formt so eine 32x32-Matrix über dem Körper $\mathbb{Z}_2$. Von dieser Matrix wird der Rang bestimmt, der zwischen 0 und 32 liegen kann. Dies wird für 40 000 solcher Matrizen durchgeführt und gezählt, wie oft welcher Rang auftaucht, wobei Ränge ≤ 29 zusammengezählt werden, da diese sehr selten sind (zumindest, falls der Zufallsgenerator gut ist). Anschließend wird ein Chi-Quadrat-Test durchgeführt.

7.2.6 Diehard 6x8 Binary Rank Test

Ähnlich wie der vorherige Test. Aus sechs 32-Bit-Zufallszahlen, wird ein bestimmtes Byte ausgewählt. Diese bilden dann eine 6x8-Matrix. Wieder wird der Rang berechnet, Ränge kleiner gleich vier werden zusammengefasst. So werden 100 000 Ränge bestimmt, auf denen ein Chi-Quadrat-Test angewandt wird.

7.2.7 Diehard Bitstream Test

Die Eingabe wird interpretiert als 20-Bit-Wörter. Es gibt 2^{20} verschiedene 20-Bit-Wörter. Der Test geht 2^{21} Wörter durch und überprüft wieviele der möglichen Wörter nicht auftauchen. Der Erwartungswert der fehlenden Wörter ist 141909 mit Standardabweichung 428.

Anmerkung: In *diehard* werden überlappende Wörter benutzt. Die Standardabweichung ist daher anders.

7.2.8 Diehard Overlapping Pairs Sparse Occupance

Aus jeder 32-Bit Zahl werden zehn aufeinanderfolgende Bit gewählt. Die zehn Bit repräsentieren einen Buchstaben. Aus zwei aufeinanderfolgenden Buchstaben wird ein Wort. Auf diese Weise werden 2^{21} Wörter generiert. Es gibt insgesamt 1024^2 Zweibuchstabenwörter. Die Zahl der fehlenden Wörter sollte annähernd normalverteilt mit Erwartungswert 142909 und Standardabweichung 290 sein.

7.2.9 Diehard Overlapping Quadruples Sparse Occupancy Test

Dieser Test ähnelt dem vorhergenden Test. Die Buchstaben werden jeweils durch fünf Bit repräsentiert. Ein Wort besteht nun aus vier, statt aus zwei Buchstaben. Wieder werden 2^{21} Wörter generiert und die fehlenden Wörter gezählt. Der Erwartungswert ist auch 141909 und die Standardabweichung 295.

7.2.10 Diehard DNA Test

Der DNA-Test ähnelt den vorherigen Tests. Es gibt vier Buchstaben, jeder durch zwei Bits bestimmt. Ein Wort besteht aus zehn Buchstaben, so dass wir $4^{10} = 2^{20}$ mögliche Wörter erhalten. Wie gewohnt werden 2^{21} Wörter erstellt und die fehlenden gezählt. Erwartungswert ist wieder 141909 und die Standardabweichung 339.

7.2.11 Diehard Count the 1s (stream) Test

Dieser Test betrachtet die Eingabe als Bytes und zählt, wieviele Einser in ihnen vorkommen. Die Wahrscheinlichkeiten für die Zahl der Einser in einem Byte sind $\frac{1}{256}, \frac{8}{256}, \frac{28}{256}, \frac{56}{256}, \frac{70}{256}, \frac{56}{256}, \frac{28}{256}, \frac{8}{256}$ und $\frac{1}{256}$. Falls null, eins oder zwei Einser vorkommen, wird dem Byte der Buchstabe A zugeordnet, für drei Einser B, für vier C, für fünf D, für sechs, sieben oder acht E. Marsaglia spielt gerne mit dem Bild des Zufallsaffen, der mit den Wahrscheinlichkeiten $\frac{37(=1+8+28)}{128}, \frac{56}{128}, \frac{70}{128}, \frac{56}{128}$ und $\frac{37}{128}$ die Buchstaben A,B,C,D und E tippt. Bezeichnen wir die Abfolge der Buchstaben mit $Y = Y_0, Y_1, Y_2, ...$ Dann ist S_i mit $S_i = Y_i, Y_{i+1}, Y_{i+2}, Y_{i+3}, Y_{i+4}$ ein Fünfbuchstabenwort. Dabei überlappen sich die Wörter S_i, sind also voneinander abhängig. Es gibt 5^5 verschiedene Wörter, deren Auftauchen gezählt wird. Alleine das Zählen der Wörter würde aber außer Acht lassen, wie sie ‚verteilt' sind. Es wird also wieder mithilfe einer Pseudoinversen von der Kovarianzmatrix ein Chi-Quadrat-Test möglich gemacht.

7.2.12 Diehard Count the 1s (byte) Test

Dieser Test funktioniert wie der stream-Test. Es wird allerdings nicht die komplette Ausgabe des Zufallsgenerators genutzt.

7.2.13 Diehard Parking Lot Test

Dieser Test simuliert einen Parkplatz der Größe 100×100. Auf diesem werden quadratische Autos mit Kantenlänge eins geparkt. Dabei vergleicht der Test die Zahl der erfolgreich geparkten Autos (k) und der Parkversuche (n). Wird ein neues Auto geparkt, wird getestet, ob es an der Stelle eine Überlappung mit vorhandenen Autos gibt. Falls nicht, wird das Auto erfolgreich geparkt, falls doch, wird das Auto zurückgewiesen. Nach Marsaglias Untersuchungen mit nach seiner Einschätzung sehr guten Zufallsgeneratoren sollte der Erwartungswert von k für $n = 12000$ bei 3523 liegen und eine Standardabweichung von $21,9$ haben. Außerdem soll k (fast) normalverteilt und somit $(k - 3523)/21,9$ standardnormalverteilt sein [12]. Eine Arbeit von Stefan C. Agapie und Paula A. Whitlock beschäftigt sich mit dem Parking-Lot-Test [1]. Sie errechneten Erwarungswert und Standardabweichung für drei Generatoren. Sie stimmten überein mit denen Marsaglias. Weiter bestimmten

sie Erwartungswert und Standardabweichung für den Fall von kreisförmigen Autos (also der Hubschrauberfall) anstelle von quadratischen für den zwei-, drei-, und vierdimensionalen Fall.

7.2.14 Diehard Minimum Distance (2d Circle) Test

Dieser Test erzeugt 8000 Punkte in einem Quadrat mit Kantenlänge 10000. Die kürzeste Distanz zweier Punkte sei d. Die Distanz zum Quadrat sollte exponentialverteilt mit Erwartungswert $0,995$ sein. Also sollte $p = 1-e^{-\frac{d^2}{0,995}}$ gleichverteilt sein. Auf diese p wird ein KS-Test angewendet.

7.2.15 Diehard 3d-Sphere (Minimum Disctance) Test

Gegeben sei ein Würfel mit Kantenlänge 1000. In ihm werden 4000 Punkte zufällig gewählt. Blasen wir um jeden Punkt eine Hülle langsam auf, berührt irgendwann die Hülle eines Punktes einen anderen Punkt (und umgekehrt). Der Radius r dieser Hülle ist die kürzeste Distanz zweier Punkte. Das Volumen der Hülle sollte exponentialverteilt sein mit Erwartungswert $120\frac{\pi}{3}$. Folglich sollte r^3, der Radius hoch drei, exponentialverteilt mit Erwartungswert 30 sein. Dann können wir mit $p = 1 - e^{-\frac{r^3}{30}}$ gleichverteilte p-Werte generieren, auf die der KS-Test angewendet wird.

7.2.16 Diehard Squeeze Test

Der Squeeze-Test generiert aus der Eingabe Zahlen $u_0, u_1, u_2, ... \in [0, 1)$. Sei $x_0 = 2^{31} - 1$. Eine Iteration ist definiert als $x_j = \lceil x_{j-1}u_{j-1} \rceil$ (wobei $\lceil \rceil$ die Aufrundungsfunktion ist). Dann wird getestet, wieviele Iterationen nötig sind, bis $x_j = 1$ ist. Der Test wird 100000 mal wiederholt. Die j-Werte werden anschließend mit einem Chi-Quadrat-Test überprüft. Brown sieht den Test als „wenig hilfreich“ an.

7.2.17 Diehard Sums Test

Es werden wie beim Squeeze-Test Zahlen $u_0, u_1, u_2, ... \in [0, 1)$ generiert. Aus ihnen werden in *diehard* die sich überlappenden Summen $S_0 = u_0 + u_1 + ... + u_{99}, S_1 = u_1 + u_2 + ... + u_{100}, ...$ berechnet. Unter anderem mithilfe einer Kovarianzmatrix werden standartnormalverteilte Größen berechnet. In *dieharder* überlappen sich die Summen nicht, die Berechnung der p-Werte erfolgt dann durch die Annnahme der Normalverteilung, da sich die Summe von vielen gleichverteilten Zufallsvariablen der Normalverteilung nähert.

7.2.18 Diehard Runs Test

Eingabe sind wieder Zahlen $u_0, u_1, u_2, ... \in [0, 1)$. Der Test zählt Aufstiege, Abstiege und deren Längen. Beispiel: Die Folge 0,002; 0,13; 0,842; 0,52; 0,9;

0,42; 0,325; ... enthält einen Aufstieg der Länge drei, einen Abstieg der Länge zwei, einen Aufstieg der Länge zwei und einen Abstieg mit einer Länge von mindestens drei. Mithilfe von Pseudoinversen der Kovarianzmatrizen lassen sich Chi-Quadrat-Tests durchführen.

7.2.19 Diehard Craps Test

Craps ist ein Würfelspiel, bei dem der Schütze (der Würfelnde) mit zwei Würfeln spielt. Es wird jeweils die Augensumme gebildet. Würfelt der Schütze in der ersten Runde eine sieben oder elf, hat er gewonnen. Würfelt er eine zwei, drei oder zwölf, hat er verloren. Würfelt er eine andere Augensumme, so bildet diese seinen *Point* und das Spiel geht weiter. Ziel für den Schützen ist nun, seinen *Point*, also seine Augensumme aus der ersten Runde zu wiederholen. Schafft er dies, bevor er eine sieben würfelt, hat er gewonnen, sonst verloren. Die Chance zu gewinnen liegt bei $p = \frac{244}{495}$. Der Craps-Test produziert aus der Eingabe Zahlen $u_0, u_1, u_2, ..., u_{n-1} \in [0, 1)$. Der Wert eines einzelnen Würfels W_i wird simuliert durch $W_i = \lfloor u_i \cdot 6 \rfloor + 1$. Es werden n Spiele gespielt, die Zahl der gewonnen Spiele sollte normalverteilt sein mit Erwartungswert $E = n \cdot p$ und Varianz $V = n \cdot p \cdot (p-1)$. Des Weiteren zählt der Test, wieviele der n Spiele nach 1,2,3,...,20,20+ Würfen beendet sind, und vergleicht sie mit dem zu erwartenden Wert.

7.2.20 STS Monobit Test

Dieser einfache Test zählt, wieviele Nullen (n_0) und Einser (n_1) in der Bit-Eingabe vorkommen und bildet die Differenz $n_1 - n_0 (= n_1 - (n - n_1) = 2n_1 - n)$. Transformiert mit $\sqrt{n}$ sollte sie (annähernd) standard-normalverteilt sein.

7.2.21 STS Runs Test

Als Eingabe dienen dem Test Bitstrings, die zyklisch betrachtet werden. Der Test zählt, wieviele ‚Einserruns' und ‚Nullerruns' es gibt. Ein Einserrun beginnt mit 01 und endet mit 10, ein Nullerrun beginnt mit 10 und hört mit 01 auf. Man kann also gleichermaßen das Auftauchen von 10 und 01 in dem Bitstring zählen. Die Anzahlen werden mit den Erwartungswerten verglichen ($p = 0,25$). So gesehen ist der Test, wie auch Brown schreibt, in dem RGB-Bit-Distribution-Test enthalten, der zusätzlich die 00er und 11er zählt.

7.3 Marsaglias Diehard Testsuite

Die *diehard*-Testsuite [11] ist eine Testsuite für Zufallsgeneratoren, entwickelt von George Marsaglia. Lange Zeit war sie sehr populär und ein Standard zum Testen von Zufallsgeneratoren. Eine ihrer Stärken ist die Vielseitigkeit ihrer

Tests, eine ihrer Schwächen, dass die Parameter der einzelnen Tests nicht veränderlich sind. Auch die Eingabegröße:
„The tests in DIEHARD require that you provide a large binary file of random integers to be tested.“
- erwartet wurde eine Größe von 10-11 MB - ist für heutige Verhältnisse zu klein. Die Testsuite Marsaglias beinhaltet fünfzehn Tests auf Zufälligkeit:

- Birthday Spacings
- Overlapping Permutations
- Ranks of 31x31 and 32x32 matrices
- Ranks of 6x8 matrices
- Monkey Tests on 20-bit words
- Monkey Tests OPSO,OQSO,DNA
- Count the 1's in a Stream of Bytes
- Count the 1's in Specific Bytes
- Parking Lot Test
- Minimum Distance Test
- Random Spheres Test
- The Squeeze Test
- Overlapping Sums Test
- Runs Test
- Craps Test

Als Eingabe fordert die Diehard-Testsuite circa 250 000 Zeilen mit je zehn 32-Bit-Zahlen. Die Testsuite wurde dreimal angewandt:

Erster Test Der Input für SWIFFTX war ein char-Vektor der Länge 1000, dessen chars in 250 000 Iterationen ‚aufgefüllt' wurden. So wurden 250 000 Hashwerte gebildet und jeweils die ersten 320 Bit gewählt.

Zweiter Test Als Startvektor diente (0...0). Der zweite Hashwert wurde von dem ersten gebildet usw. Die iterative Anwendung von Hashfunktionen ist in der Kryptographie üblich. Es wurden wieder 250 000 Hashwerte gebildet und jeweils die ersten 320 Bit ausgewählt.

Dritter Test Wieder wurde iteriert mit Startvektor (0...0), diesmal wurden aber nicht nur die ersten 320 Bit, sondern die komplette Ausgabe der Hashfunktion gewählt.

SWIFFTX bestand in allen drei Fällen alle oben genannten Tests.

7.4 Dieharder Testsuite

Dieharder ist eine umfangreiche Testsuite von Robert G. Brown, Professor der Duke University, Durham. Sie enthält zum einen die Tests aus *diehard*. Die Tests wurden allerdings neu programmiert, so dass sie nicht nur mit größeren Datenmengen arbeiten können, sondern auch deren Parameter variabel sind. Viele Tests enthielten keinen anschließenden Kuiper-KS-Test (Definition 7.13) oder nur mit wenigen Testwiederholungen. Durch die größere (variable) Anzahl von Testwiederholungen wurden viele Tests schärfer und ihr anschließender Kuiper-KS-Wert aussagekräftiger. Viele Zufallsgeneratoren, die *diehard* bestehen, fallen bei *dieharder* durch.

Des Weiteren enthält *dieharder* zwei Tests aus der Statistical Testsuite, entwickelt von NIST (National Institute for Standards and Technology) und mehrere von Brown selbst entworfene Tests.

Die Testsuite ist unter der GNU Public License (mit einer interessanten ‚Beverage'-Modifikation) veröffentlicht.

Zum Testen wurde die *dieharder*-Version 3.31.0 verwendet. Einige Tests der Suite sind noch nicht vollständig und korrekt implementiert und wurden hier nicht verwendet.

Die Testdatei: Die 32-Bit-Zufallszahlen für *dieharder* wurden aus den Hashwerten von SWIFFTX gewonnen. Es wurden jeweils 100 000 Hashwerte durch Iterationen mit den Startwerten 0,1 und 2 gebildet und jeweils 500 000 Hashwerte mit den Startwerten 3 und 4. Zusammen ergab dies eine Testdatei mit 20,8 Millionen 32-Bit-Zahlen, Resultat der 1,3 Millionen 512-Bit-Hashwerte. Die Ergebnisse gekürzt:

```
#=============================================================================#
#        dieharder version 3.31.0 Copyright 2003 Robert G. Brown              #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                       zufall.zuf|  7.07e+05  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.79113746|  PASSED
      diehard_operm5|   0|   1000000|     100|0.21003838|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.37414877|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.32195284|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.85361408|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.41232343|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.16310495|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.65687896|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.14978046|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.81092162|  PASSED
     diehard_squeeze|   0|    100000|     100|0.01312136|  PASSED
        diehard_runs|   0|    100000|     100|0.04061280|  PASSED
```

```
       diehard_runs|   0|    100000|     100|0.91885261|  PASSED
      diehard_craps|   0|    200000|     100|0.07625406|  PASSED
      diehard_craps|   0|    200000|     100|0.12793724|  PASSED
 marsaglia_tsang_gcd|   0|     50000|     100|0.32481897|  PASSED
 marsaglia_tsang_gcd|   0|     50000|     100|0.08362886|  PASSED
        sts_monobit|   1|    100000|     100|0.25365114|  PASSED
           sts_runs|   2|    100000|     100|0.85432696|  PASSED
         sts_serial|   1|    100000|     100|0.25365114|  PASSED
         sts_serial|   2|    100000|     100|0.81068282|  PASSED
         sts_serial|   3|    100000|     100|0.73969862|  PASSED
         sts_serial|   3|    100000|     100|0.65325912|  PASSED
         sts_serial|   4|    100000|     100|0.50258146|  PASSED
         sts_serial|   4|    100000|     100|0.96331871|  PASSED
         sts_serial|   5|    100000|     100|0.17605859|  PASSED
         sts_serial|   5|    100000|     100|0.48918103|  PASSED
         sts_serial|   6|    100000|     100|0.65130794|  PASSED
         sts_serial|   6|    100000|     100|0.96900072|  PASSED
         sts_serial|   7|    100000|     100|0.98897576|  PASSED
         sts_serial|   7|    100000|     100|0.80302439|  PASSED
         sts_serial|   8|    100000|     100|0.98043104|  PASSED
         sts_serial|   8|    100000|     100|0.84849437|  PASSED
         sts_serial|   9|    100000|     100|0.91878554|  PASSED
         sts_serial|   9|    100000|     100|0.98468808|  PASSED
         sts_serial|  10|    100000|     100|0.32771873|  PASSED
         sts_serial|  10|    100000|     100|0.37535462|  PASSED
         sts_serial|  11|    100000|     100|0.27147737|  PASSED
         sts_serial|  11|    100000|     100|0.06247486|  PASSED
         sts_serial|  12|    100000|     100|0.42795783|  PASSED
         sts_serial|  12|    100000|     100|0.54928317|  PASSED
         sts_serial|  13|    100000|     100|0.54764445|  PASSED
         sts_serial|  13|    100000|     100|0.72799337|  PASSED
         sts_serial|  14|    100000|     100|0.98172159|  PASSED
         sts_serial|  14|    100000|     100|0.42884838|  PASSED
         sts_serial|  15|    100000|     100|0.65129478|  PASSED
         sts_serial|  15|    100000|     100|0.51813304|  PASSED
         sts_serial|  16|    100000|     100|0.71260107|  PASSED
         sts_serial|  16|    100000|     100|0.88050438|  PASSED
        rgb_bitdist|   1|    100000|     100|0.88369447|  PASSED
        rgb_bitdist|   2|    100000|     100|0.62385639|  PASSED
        rgb_bitdist|   3|    100000|     100|0.64992198|  PASSED
        rgb_bitdist|   4|    100000|     100|0.92011759|  PASSED
        rgb_bitdist|   5|    100000|     100|0.62359679|  PASSED
        rgb_bitdist|   6|    100000|     100|0.92059620|  PASSED
        rgb_bitdist|   7|    100000|     100|0.18713195|  PASSED
        rgb_bitdist|   8|    100000|     100|0.41499849|  PASSED
        rgb_bitdist|  12|    100000|     100|0.60788110|  PASSED
#=============================================================================#
```

Keiner der Tests fand eine Schwachstelle von SWIFFTX!

8 Korrektur von Operm5

Schon die Beschreibung auf Browns Internetseite zu seiner Testsuite machte auf einen Test neugierig:
„Note that a few tests appear to have stubborn bugs. In particular, the diehard operm5 test seems to fail all generators in dieharder. Several users have attempted to help debug this problem, and it tentatively appears that the problem is in the original diehard code and not just dieharder. There is extensive literature on overlapping tests, which are highly non-trivial to implement and involve things like forming the weak inverse of covariance matrices in order to correct for overlapping (non-independent) statistics.“
Im Quellcode fanden sich unter anderem folgende Kommentare zu Operm5:
„Note – this test almost certainly has errors. It has been suggested that the actual rank is 5!-4!=96, not 99. However, ”good” generators still fail this test with the lower rank. I really think that the covariance matrix is going to have to recomputed...“,
„Good test. Just about everything fails it“,
„It would be nice, so nice, to have SOME clue how to actually generate the matrices and other numbers since even a simple sign error on a single number could make the test useless and (incidentally) cause it to sometimes return a negative chisq.“
Zur Beschreibung des Operm5-Tests siehe Kapitel 7.2.4.

8.1 Abhängigkeiten von sich überlappenden Zufallsvariablen

Dieser Teil richtet sich sehr nach „The Mathematics of the Overlapping Chi-square Test“ von Tsang und Pang [19].

Seien $0, 1, ..., d-1$ die möglichen Ausgabewerte eines Experiments mit Wahrscheinlichkeiten $P(0), P(1), ..., P(d-1)$. Wiederholt man das Experiment n mal, erhält man als Ausgabe $Y_1, Y_2, ..., Y_n$ mit $0 \leq Y_i < d$. Sei $N(i)$ die Anzahl, dass i ausgegeben wird. Der Chi-Quadrat-Test lautet dann

$$V = \sum_{i=0}^{d-1} \frac{(N(i) - nP(i))^2}{nP(i)}.$$

Asymptotisch ist V chi-quadrat-verteilt.
Wie Marsaglia schon beobachtete, untersucht dieser Test nur die Anzahl der Ausgaben, nicht aber deren Abhängigkeit untereinander. Er schlug daher vor, die Ausgaben $S_1 = (Y_1, Y_2, ..., Y_t)$, $S_2 = (Y_2, Y_3, ..., Y_{t+1})$,...,$S_n = (Y_n, Y_1, ..., Y_{t-1})$ zu betrachten. Sei $\alpha = a_1, a_2, ..., a_t$ mit $0 \leq \alpha < d$. Wir nennen α ein Wort. Bezeichne $N(\alpha)$, wie oft $S_i = \alpha$ ist. Dann ist

$$V = \sum_{|\alpha|=t} \frac{(N(\alpha) - nP(\alpha))^2}{nP(\alpha)} - \sum_{|\alpha|=t-1} \frac{(N(\alpha) - nP(\alpha))^2}{nP(\alpha)}$$

asymptotisch chi-quadrat-verteilt.
Um dies zu beweisen, wurden Tsang und Pang allgemeiner.

Sei $X = (X_1, X_2, ..., X_n)^T$ ein Vektor mit n standardnormalverteilten Zufallsvariablen. Es gilt also $E(X_i) = 0$ und $\mathrm{Var}(X_i) = 1$. Sei

$$N = \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_m \end{bmatrix} = AX + \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_m \end{bmatrix}. \tag{6}$$

Dann gilt $E[N] = (\mu_1, \mu_2, ..., \mu_m)$. Sei C die Kovarianzmatrix von N.

Theorem 8.1. *Seien N und A wie in (6). Dann ist*

$$C = AA^T$$

die Kovarianzmatrix von N.

Beweis. (aus [19])

$$\begin{aligned} c_{ij} &= E[(N_i - \mu_i)(N_j - \mu_j)] \\ &= E\left[\sum_{k=1}^{n} a_{ik}X_k \sum_{l=1}^{n} a_{jl}X_l\right] \\ &= \sum_{k=1}^{n} a_{ik}X_k \sum_{l=1}^{n} a_{jl}E[X_kX_l] \\ &\quad X_k \text{ und } X_l \text{ sind für } k \neq l \text{ unabhängig} \\ &= \sum_{k=1}^{n} a_{ik}a_{jk} \end{aligned}$$

□

Theorem 8.2. *Seien N und μ wie in (6). Sei C die Kovarianzmatrix von N und P_C eine Pseudoinverse von C (also $CP_CC = C$). Dann ist*

$$(N - \mu)^T P_C (N - \mu)$$

chi-quadrat verteilt.

Beweis. (siehe [19]) □

Es gilt also $(N - \mu)^T P_C (N - \mu) = X_1^2 + X_2^2 + ... + X_n^2 = X^T X$.

Bestimmung der Sortierungsnummer Eingabe für den Operm5-Test sind n 32-Bit-Zahlen $I_1, I_2, ..., I_n$. Setzen wir $M := 2^{32} - 1$ gilt folglich $I_i \in \{0, 1, ..., M\}$. Mit J_i bezeichnen wir die fünf aufeinander folgenden Zahlen, die mit I_i beginnen: $J_i = (I_i, I_{i+1}, I_{i+2}, I_{i+3}, I_{i+4})$. Jedem J_i wird durch die Funktion f eine Zahl zugeordnet, die eine Sortierung repräsentiert.

Beschreibung von f: Sei $J_i = (a, b, c, d, e)$ und sei $k_4 \in \{0, 1, 2, 3, 4\}$ die Stelle, an der die größte Zahl in J_i steht, beginnend mit der Null. Ist sie nicht schon an der letzten (vierten Stelle), so tauscht sie mit der vierten den Platz. Sei dann $k_3 \in \{0, 1, 2, 3\}$ die Stelle, an der (nach der möglichen Vertauschung der größten Zahl mit der vierten Zahl) die zweitgrößte Zahl steht. Die zweitgrößte Zahl wird dann, falls nötig, mit der an Stelle drei stehenden Zahl getauscht, usw. Falls zwei Zahlen gleich groß sind, wird die größere Stelle gewählt. Die Formel zur Berechnung der Sortierungsnummer lautet dann $24k_4 + 6k_3 + 2k_2 + k_1$. Beispiel: $J_0 = (123, 42, 3312, 2532, 452)$, dann gilt $k_4 = 2, k_3 = 3, k_2 = 2, k_1 = 0$ und wir erhalten $f(J_i) = 70$. Die Zahl 70 repräsentiert die Sortierung (10432).

Durch die Bedingungen an die k_i gilt $24 > 6k_3 + 2k_2 + k_1 \leq 23$ und $6 > 2k_2 + k_1 \leq 5$ und $2 > k_1 \leq 1$. Das heißt $f(J_i) \in \{0, 1, ..., 119\}$ und falls J_i und J_j unterschiedlich sortiert sind, gilt $f(J_i) \neq f(J_j)$. Der Algorithmus zählt nun einfach, wie oft die 120 möglichen Zahlen, die die Sortierungen repräsentieren, auftauchen. Wir nennen diese Anzahlen $N_0, N_1, ..., N_{119}$.

Das Ziel ist, Theorem 8.2 auf unsere Sortierungsanzahlen $N_0, N_1, ..., N_{119}$ anzuwenden.

8.2 Marsaglias Weg

Zu jeder Sortierung α gibt es eine Sortierung β, die zu α „gespiegelt“ ist. So hätte $J_i = (1, 12, 24, 312, 4123)$ die gespiegelte Sortierung von $J_j = (3123, 421, 300, 123, 12)$. Diese gespiegelten Sortierungen sind eindeutig. Im „maps“-Vektor hat Marsaglia festgehalten, welche Sortierungen zueinander gespiegelt sind. Die Anzahlen der zueinander gespiegelten Sortierungen werden addiert in einem Vektor X und subtrahiert in einem Vektor Y festgehalten. Da die Anzahlen der Sortierungen normalverteilt sein sollten, ist nun, falls Pseudoinversen der Kovarianzmatrizen von X und Y gegeben sind (bei Marsaglia sind dies ‚r‘ und ‚s‘), ein Chi-Quadrat-Test möglich. Leider gibt Marsaglia keine Rechenvorschriften an, so dass nicht direkt nachverfolgt werden kann, ob r und s stimmen. Die Resultate bei Verwendung dieser Pseudoinversen sprechen gegen die verwendeten Matrizen r und s...

Wir wenden trotz aller Warnungen den Operm5-Test aus *dieharder*, Version 3.29.4beta und älter an. Als Zufallsgenerator ist in diesem Beispiel AES (Advanced Encryption Standard) gewählt. AES ist die von NIST als Standard empfohlene Verschlüsselungsfunktion und ist in den USA für die Ver-

schlüsselung von staatlichen Dokumenten höchster Geheimhaltungsstufe zugelassen [15].

```
#=============================================================================#
#            dieharder version 3.31.0 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
        AES_OFB|  1.19e+07  |1663462009|
#=============================================================================#
#          Diehard Overlapping 5-Permutations Test.
# This is the OPERM5 test.  It looks at a sequence of one mill-
# ion 32-bit random integers.  Each set of five consecutive
# integers can be in one of 120 states, for the 5! possible or-
# derings of five numbers.  Thus the 5th, 6th, 7th,...numbers
# each provide a state. As many thousands of state transitions
# are observed,  cumulative counts are made of the number of
# occurences of each state.  Then the quadratic form in the
# weak inverse of the 120x120 covariance matrix yields a test
# equivalent to the likelihood ratio test that the 120 cell
# counts came from the specified (asymptotically) normal dis-
# tribution with the specified 120x120 covariance matrix (with
# rank 99).  This version uses 1,000,000 integers, twice.
#
# Note that Dieharder runs the test 100 times, not twice, by
# default.
#
# WARNING! This test currently fails ALL RNGs including ones that
# are strongly believed to be "good" ones (that pass the other
# dieharder tests).  DO NOT USE THIS TEST TO ASSESS RNGs!  It very
# likely contains either implementation bugs or incorrect data used
# to compute the test statistic.  rgb
#=============================================================================#
#                          Histogram of test p-values                         #
#=============================================================================#
# Bin scale = 0.100000
#    220|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#    198|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#    176|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#    154|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#    132|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#    110|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#     88|****|    |    |    |    |    |    |    |    |****|
#       |****|****|****|****|****|    |    |****|****|****|
#     66|****|****|****|****|****|****|    |****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#     44|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#     22|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
      diehard_operm5|   5|   1000000|    1000|0.00000000|  FAILED
```

Das Ergebnis des Kuiper-KS-Tests bestätigt, was an der grafischen Darstellung erkennbar ist: Die p-Werte sind (sehr wahrscheinlich) nicht gleichverteilt. Da mit AES nicht irgendein Zufallsgenerator durchfällt und Operm5 für alle Generatoren solche Ausgaben erstellt muss der Fehler an der Implementierung des Tests liegen. Dazu kommt eine zu große Abhängigkeit von den Parametern und negative Werte als Teststatistik für den Chi-Quadrat-Test.

8.3 Berechnung der Kovarianzmatrix

Sei S_i die Sortierung von J_i und sei α eine Sortierung. Dann ist $P(\alpha)$ die Wahrscheinlichkeit für die Sortierung α, also $\frac{1}{120}$. Wir definieren wie Tsang und Pang mit Z_i^α eine neue Zufallsvariable. So soll gelten $Z_i^\alpha = 1$, falls $S_i = \alpha$, und $Z_i^\alpha = 0$ sonst. Der Erwartungswert von Z^α ist dann $P(\alpha)$. Es gilt $N_{f(\alpha)} = \sum_{i=0}^{n} Z_i^\alpha$. Sei β eine weitere Sortierung. Wenden wir das Tsang-Pang-Theorem 3.9. aus [19] auf unseren Fall an erhalten wir:

Theorem 8.3. *Für die Kovarianzmatrix C von N gilt*

$$C = (c_{\alpha\beta}) \quad mit \quad c_{\alpha\beta} = n \left(\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta \right] - 9 \cdot P(\alpha)P(\beta) \right).$$

Beweis.

$$\begin{aligned}
c_{\alpha\beta} =& E\left[(N(\alpha) - E[N(\alpha)])(N(\beta) - E[N(\beta)]) \right] \\
&\text{Mit dem Verschiebungssatz von Steiner folgt:} \\
=& E\left[N(\alpha)N(\beta) \right] - E[N(\alpha)]E[N(\beta)] \\
=& E\left[N(\alpha)N(\beta) \right] - n^2 P(\alpha)P(\beta) \\
=& E\left[\left(\sum_{i=1}^{n} Z_i^\alpha \right) \left(\sum_{j=1}^{n} Z_j^\beta \right) \right] - n^2 P(\alpha)P(\beta) \\
=& \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \left(E\left[Z_i^\alpha Z_j^\beta \right] \right) \right) - n^2 P(\alpha)P(\beta) \\
&\text{Da wir uns auf einem Ring bewegen und die Position egal ist,} \\
&\text{wählen wir } t \text{ als ein festes } i. \\
=& n \sum_{j=1}^{n} \left(E\left[Z_t^\alpha Z_j^\beta \right] \right) - n^2 P(\alpha)P(\beta) \\
=& nP(\alpha)P(\beta) \left(\sum_{j=1}^{n} \left(\frac{E\left[Z_t^\alpha Z_j^\beta \right]}{P(\alpha)P(\beta)} \right) - n \right)
\end{aligned}$$

$$
\begin{aligned}
c_{\alpha\beta} =& nP(\alpha)P(\beta)\sum_{j=1}^{n}\left(\frac{E\left[Z_t^\alpha Z_j^\beta\right]}{P(\alpha)P(\beta)} - 1\right) \\
&\text{Für } |t-j| \geq 5 \text{ sind } Z_t^\alpha \text{ und } Z_j^\beta \text{ voneinander unabhängig.} \\
=& nP(\alpha)P(\beta)\sum_{j=t-4}^{t+4}\left(\frac{E\left[Z_t^\alpha Z_j^\beta\right]}{P(\alpha)P(\beta)} - 1\right) \\
=& n\left(\left(\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta\right]\right) - 9\cdot P(\alpha)P(\beta)\right)
\end{aligned}
$$

□

Der Ansatz um $E\left[Z_t^\alpha Z_j^\beta\right]$ für $j \in \{-4,-3,...,4\}$ zu berechnen ist einfach: jeweils die Zahl der günstigen Fälle durch alle Fälle teilen. Die günstigen Fälle für positive j, sind die Ausgaben $I_t, I_{t+1}, ..., I_{t+j+4}$ des Zufallsgenerators mit $f(I_t, I_{t+1}, I_{t+2,I}\, t+3, I_{t+4}) = f(J_t) = \alpha$ und gleichzeitig $f(J_{t+j}) = \beta$; analog für negative j die Ausgaben $I_{t+j}, I_{t+j+1}, ..., I_{t+4}$ des Zufallsgenerators mit $f(J_{t+j}) = \beta$ und $f(J_t) = \alpha$.

Um hierbei sauber arbeiten zu können, ist es hilfreich, dass die Sortierungen eindeutig sind. Wir möchten daher ausschließen, dass bei der Berechnung von f zwei gleiche Zahlen auftauchen. Die Wahrscheinlichkeit, dass zwei gleiche Zahlen auftreten, liegt bei

$$\frac{(M+1)^7\binom{9}{2}(M+1)}{(M+1)\cdot M\cdot ...\cdot (M-7)} \quad < \quad 8,4\cdot 10^{-9}$$

bei einer einfachen Überlappung, d.h. $j \in \{-4,4\}$,

$$\frac{(M+1)^6\binom{8}{2}(M+1)}{(M+1)\cdot M\cdot ...\cdot (M-6)} \quad < \quad 6,6\cdot 10^{-9}$$

bei einer zweifachen Überlappung ($j \in \{-3,3\}$),

$$\frac{(M+1)^5\binom{7}{2}(M+1)}{(M+1)\cdot M\cdot ...\cdot (M-5)} \quad < \quad 4,9\cdot 10^{-9}$$

bei einer dreifachen Überlappung ($j \in \{-2,2\}$),

$$\frac{(M+1)^4\binom{6}{2}(M+1)}{(M+1)\cdot M\cdot ...\cdot (M-4)} \quad < \quad 3,5\cdot 10^{-9}$$

bei einer vierfachen Überlappung ($j \in \{-1,1\}$), und bei

$$\frac{(M+1)^3\binom{5}{2}(M+1)}{(M+1)\cdot M\cdot ...\cdot (M-3)} \quad < \quad 2,4\cdot 10^{-9}$$

bei kompletter Überlappung ($j = 0$).

Die Wahrscheinlichkeit ist also sehr gering. Daher wird im Folgenden angenommen, dass in neun aufeinander folgenden Zahlen des RNGs nie gleiche Zahlen auftreten.
Bevor wir zum Allgemeinen kommen, zunächst ein Beispiel. Seien $\alpha = 04123$ und $\beta = 20413$ Sortierungen und wir möchten $\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_{t+j}^\beta\right]$ berechnen.
Wir bezeichnen die ‚zentrale' Sortierung α als Hauptsortierung und die an α entlangwandernde Sortierung β als Nebensortierung. Die für die Hauptsortierung α günstigen Fälle werden durch ineinander verschachtelte Summen repräsentiert. Der Index k repräsentiert die kleinste Zahl von J_t, in diesem Fall also I_t. Der Index l steht für die zweitgrößte Zahl von J_t, also I_{t+1}, m für I_{t+2}, n für I_{t+4} und o für I_{t+3}. Lässt man die Nebensortierung außer Acht, ergeben sich

$$\sum_{k=0}^{M}\sum_{l=k+1}^{M}\sum_{m=l+1}^{M}\sum_{n=m+1}^{M}\sum_{o=n+1}^{M} 1 = \binom{M+1}{5}$$

Möglichkeiten, dass $f(J_t) = \alpha$ ist. Dies ist genau ein 120-stel von $(M+1)\cdot M\cdot(M-1)\cdot(M-2)\cdot(M-3)$, der Zahl aller möglichen Kombinationen von J_r.

Die Nebensortierung β muss sich der Hauptsortierung, falls möglich, anpassen.

j	Y_{t-4}	Y_{t-3}	Y_{t-2}	Y_{t-1}	Y_t	Y_{t+1}	Y_{t+2}	Y_{t+3}	Y_{t+4}	Y_{t+5}	Y_{t+6}	Y_{t+7}	Y_{t+8}
					0	4	1	2	3				
-4	2	0	4	1	3								
-3		2	0	4	1	3							
-2			2	0	4	1	3						
-1				2	0	4	1	3					
0					2	0	4	1	3				
1						2	0	4	1	3			
2							2	0	4	1	3		
3								2	0	4	1	3	
4									2	0	4	1	3

Wie in der Tabelle zu erkennen, ist z.B. $E\left[Z_t^\alpha Z_{t-2}^\beta\right] = 0$, da Y_t nicht gleichzeitig kleiner (für α) und größer (für β) als Y_{t+1} sein kann. Werte ungleich Null ergeben sich nur für $j \in \{-4, -3, -1, 4\}$. In diesem Fall ergibt sich

$$\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta\right] = E[Z_t^\alpha Z_{t-4}^\beta] + E[Z_t^\alpha Z_{t-3}^\beta] + E[Z_t^\alpha Z_{t-1}^\beta] + E[Z_t^\alpha Z_{t+4}^\beta].$$

Die zu β gehörenden Zahlen, die nicht auch in J_t vorkommen, werden mit Binomialkoeffizienten dargestellt. Die Indizes, die Zahlen repräsentieren, die zu α und β gehören, bestimmen wie groß die Menge der Zahlen ist, die von den außerhalb der Überlappung und zu β gehörenden I_i zur Verfügung steht. Das heißt, die oberen Einträge der Binomialkoeffizienten sind schon durch α und die Größe der Überlappung festgelegt. Von β hängen die unteren Einträge der Binomialkoeffizienten, nämlich wieviele Zahlen unter, zwischen oder über den Zahlen liegen, die zu α und β gehören, ab.

$$\begin{aligned} E[Z_t^\alpha Z_{t-4}^\beta] &= \frac{\sum_{k=0}^{M}\sum_{l=k+1}^{M}\sum_{m=l+1}^{M}\sum_{n=m+1}^{M}\sum_{o=n+1}^{M}\left(\binom{k}{3}\binom{M-k-4}{1}\right)}{\binom{M+1}{9}9!} \\ &= \frac{1}{72576} \end{aligned}$$

Der erste Binomialkoeffizient steht für die Zahl, die es in J_{t-4} geben muss, die kleiner als I_t ist, welches durch k repräsentiert wird. Der zweite Binomialkoeffizient steht für die drei Zahlen in J_{t-4}, die größer als I_t sind. Hier muss allerdings darauf geachtet werden, dass vier Zahlen aus J_t größer sind als I_t. Da neun aufeinanderfolgende Zahlen keine zwei gleichen Zahlen enthalten, müssen diese vier Zahlen abgezogen werden.

$$\begin{aligned} E[Z_t^\alpha Z_{t-3}^\beta] &= \frac{\sum\limits_{k=0}^{M}\sum\limits_{l=k+1}^{M}\sum\limits_{m=l+1}^{M}\sum\limits_{n=m+1}^{M}\sum\limits_{o=n+1}^{M}\left(\binom{k}{1}\binom{o-k-1-3}{1}\binom{M-o}{1}\right)}{\binom{M+1}{8}8!} \\ &= \frac{1}{10080} \end{aligned}$$

Für $j = -3$ erhalten wir drei Binomialkoeffizienten. Der erste steht für die Zahlen, die kleiner als beide (!) Zahlen sind, die die Schnittmenge von J_{t-3} und J_t darstellen, also I_r und I_{t+1}. Der mittlere Binomialkoeffizient $\binom{o-k-1-3}{1}$ steht für die Zahl, die zwischen I_t und I_{t+1} liegt. I_t ist bezüglich α die kleinste Zahl und I_{t+1} ist bezüglich α die größte Zahl. Zwischen k und l liegen $k - l - 1$ Zahlen, von denen wir noch drei weitere abziehen müssen, wieder um der Annahme, dass keine Zahlen mehrfach auftreten, gerecht zu werden. Der letzte Binomialkoeffizient steht für die Zahl, die größer als I_t und I_{t+1} ist.

$$\begin{aligned} &E[Z_t^\alpha Z_{t-1}^\beta] \\ &\quad= \frac{\sum\limits_{k=0}^{M}\sum\limits_{l=k+1}^{M}\sum\limits_{m=l+1}^{M}\sum\limits_{n=m+1}^{M}\sum\limits_{o=n+1}^{M}\left(\binom{k}{0}\binom{l-k-1}{0}\binom{m-l-1}{1}\binom{o-m-1-1}{0}\binom{M-o}{0}\right)}{\binom{M+1}{6}6!} \\ &\quad= \frac{1}{720} \end{aligned}$$

Für $j = -1$ ergeben die meisten Binomialkoeffizienten Eins. Da α und β sich vierfach überlappen, gibt es nur eine zu β gehörende Zahl, die nicht auch in α steckt. Trotzdem werden alle Binomialkoeffizienten der Systematik wegen notiert, da wir an einer allgemeinen Lösung interessiert sind.

$$\begin{aligned} E[Z_t^\alpha Z_{t+4}^\beta] &= \frac{\sum_{k=0}^{M}\sum_{l=k+1}^{M}\sum_{m=l+1}^{M}\sum_{n=m+1}^{M}\sum_{o=n+1}^{M}\left(\binom{n-3}{2}\binom{M-n-1}{2}\right)}{\binom{M+1}{9}9!} \\ &= \frac{1}{12096} \end{aligned}$$

Da wir auch für $j = 4$ eine einfache Überlappung haben, gleicht der Aufbau von $E[Z_t^\alpha Z_{t+4}^\beta]$ dem von $E[Z_t^\alpha Z_{t-4}^\beta]$.

Haben wir für alle α und β die Summen $\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta\right]$ berechnet, können wir mit 8.3 die Kovarianzmatrix bestimmen. Wir berechnen die Kovarianzmatrix C für $n = 1$ und bilden mit einem Matheprogramm eine Pseudoinverse P von C.
Eine Implementierung der Berechnung der Kovarianzmatrix in Mathematica ist in Anhang B.

Theorem 8.4. *Sei C die Kovarianzmatrix aus Theorem 8.3 für $n = 1$ und P_C eine ihrer Pseudoinversen. Dann ist die Pseudoinverse von $\widetilde{C} = n \cdot C$ gleich $\widetilde{P_C} = \frac{1}{n}P_C$.*

Beweis. Aus $C = CP_CC$ folgt

$$\begin{aligned} \widetilde{C}\widetilde{P_C}\widetilde{C} &= n \cdot C \cdot \frac{1}{n}P_C \cdot n \cdot C \\ &= n \cdot C \\ &= \widetilde{C}. \end{aligned}$$

Genauso folgt aus $P_C = P_CCP_C$, dass $\widetilde{P_C}\widetilde{C}\widetilde{P_C} = \widetilde{P_C}$. □

8.4 Funktionserläuterungen

Sei *rev* die Funktion, die eine Sortierung α spiegelt. Bsp.: $rev\,((01234)) = (43210)$. Aus Symmetriegründen folgt direkt $E[Z_t^\alpha Z_{t+s}^\beta] = E[Z_t^{rev(\alpha)} Z_{t-s}^{rev(\beta)}]$ und daraus

$$\begin{aligned} \sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta\right] = {} & E[Z_t^\alpha Z_{t-4}^\beta] + E[Z_t^\alpha Z_{t-3}^\beta] + E[Z_t^\alpha Z_{t-2}^\beta] + E[Z_t^\alpha Z_{t-1}^\beta] \\ & + E[Z_t^\alpha Z_t^\beta] + E[Z_t^{rev(\alpha)} Z_{t-1}^{rev(\beta)}] + E[Z_t^{rev(\alpha)} Z_{t-2}^{rev(\beta)}] \\ & + E[Z_t^{rev(\alpha)} Z_{t-3}^{rev(\beta)}] + E[Z_t^{rev(\alpha)} Z_{t-4}^{rev(\beta)}]. \end{aligned}$$

Es langt folglich, Funktionen zu schreiben, die bezüglich der Hauptsortierung α linksseitige Überlappungen berechnen, da eine rechtsseitige Überlappung leicht in eine linksseitige Überlappung überführt werden kann.

Die Funktion Overlap1(α, β) berechnet die Anzahl der Kombinationen von J_{t-4} und J_t mit $f(J_{t-4}) = \beta$ und $f(J_t) = \alpha$. Es gilt

$$E[Z_t^\alpha Z_{t-4}^\beta] = \frac{1}{\binom{M+1}{9}9!}\,\text{Overlap1}(\alpha,\beta)$$

und

$$E[Z_t^\alpha Z_{t+4}^\beta] = E[Z_t^{rev(\alpha)} Z_{t-4}^{rev(\beta)}] = \frac{1}{\binom{M+1}{9}9!}\,\text{Overlap1}(rev(\alpha), rev(\beta)).$$

Seien Overlap2(α, β), Overlap3(α, β) und Overlap4(α, β) analog dazu die Funktionen, die die Anzahlen der bezüglich α linksseitigen zweifachen, dreifachen und vierfachen Überlappungen berechnen. Overlap5(α, β) berechnet die Anzahl der Möglichkeiten, dass $f(J_t) = \alpha = \beta$ ist. Dann erhalten wir

$$\begin{aligned}
\sum_{j=t-4}^{t+4} E\left[Z_t^\alpha Z_j^\beta\right] = \\
&\frac{1}{\binom{M+1}{9}9!}\,\text{Overlap1}(\alpha,\beta) && + \frac{1}{\binom{M+1}{8}8!}\,\text{Overlap2}(\alpha,\beta)\\
+&\frac{1}{\binom{M+1}{7}7!}\,\text{Overlap3}(\alpha,\beta) && + \frac{1}{\binom{M+1}{6}6!}\,\text{Overlap4}(\alpha,\beta)\\
+&\frac{1}{\binom{M+1}{5}5!}\,\text{Overlap5}(\alpha,\beta) && + \frac{1}{\binom{M+1}{6}6!}\,\text{Overlap4}(rev(\alpha),rev(\beta))\\
+&\frac{1}{\binom{M+1}{7}7!}\,\text{Overlap3}(rev(\alpha),rev(\beta)) && + \frac{1}{\binom{M+1}{8}8!}\,\text{Overlap2}(rev(\alpha),rev(\beta))\\
+&\frac{1}{\binom{M+1}{9}9!}\,\text{Overlap1}(rev(\alpha),rev(\beta)).
\end{aligned}$$

Die Funktion PossibleOverlap$(\alpha, \beta, \text{overlap})$ testet, ob es möglich ist, dass sich α und β linksseitig von α overlap-fach überlappen. Dazu berechnet sie die ‚Teilsortierungen' in dem überlappendem Bereich. Seien $\alpha = (a_1a_2a_3a_4a_5)$ und $\beta = (b_1b_2b_3b_4b_5)$. Falls overlap = 3, so werden die Sortierungen von $(a_1a_2a_3)$ und $(b_3b_4b_5)$ benötigt. Ein Vergleichen reicht dabei nicht aus, da z.B. die Sortierungen $\alpha = (01234)$ und $\beta(01234)$ sich dreifach überlappen können, obwohl $(012) \neq (234)$ ist. Um die Teilsortierung von α zu erhalten, vergleicht die Funktion die Zahlen a_1, a_2 und a_3 mit a_4 und a_5. Zunächst werden alle a_i, die größer als a_4 sind, um Eins reduziert (auch a_5, falls $a_4 > a_5$,

was notwendig ist). Danach werden alle a_i um Eins reduziert, die größer als a_5 sind. Nun ist $(a_1a_2a_3)$ die Teilsortierung von α für den überlappenden Bereich mit β.
Beispiele verschiedener α mit resultierenden Teilsortierungen für overlap = 3:

$$(04231) \xrightarrow{\text{Vergleich mit } a_4} (03221) \xrightarrow{\text{Vergleich mit } a_5} (02120) \xrightarrow{\text{Teilsortierung}} (021)$$

$$(01234) \xrightarrow{\text{Vergleich mit } a_4} (01233) \xrightarrow{\text{Vergleich mit } a_5} (01233) \xrightarrow{\text{Teilsortierung}} (012)$$

$$(43012) \xrightarrow{\text{Vergleich mit } a_4} (32011) \xrightarrow{\text{Vergleich mit } a_5} (21011) \xrightarrow{\text{Teilsortierung}} (210)$$

Auf die gleiche Weise verfahren wir mit β. Im ersten Schritt werden b_3, b_4 und b_5 um Eins reduziert, falls sie größer als b_1 sind. Im zweiten Schritt werden sie um Eins reduziert, falls sie größer als b_2 sind. Die Teilsortierung ist dann $(b_3b_4b_5)$.
Beispiele:

$$(01234) \xrightarrow{\text{Vergleich mit } b_1} (00123) \xrightarrow{\text{Vergleich mit } b_2} (00012) \xrightarrow{\text{Teilsortierung}} (012)$$

$$(31204) \xrightarrow{\text{Vergleich mit } b_1} (31203) \xrightarrow{\text{Vergleich mit } b_2} (21102) \xrightarrow{\text{Teilsortierung}} (102)$$

Sind die zwei Teilsortierungen von α und β berechnet, müssen sie nur noch verglichen werden. Sind sie gleich, so ist eine Überlappung möglich, ansonsten nicht.

Wenden wir den Operm5-Test mit der neuen Pseudoinverse nochmals auf AES an erhalten wir eine plausible (gekürzte) Ausgabe:

```
#=================================================================#
#            dieharder version 3.31.0 Copyright 2003 Robert G. Brown  #
#=================================================================#
   rng_name    |rands/second|   Seed   |
        AES_OFB|  1.17e+07  |2753822418|
#=================================================================#
#                         Histogram of test p-values                  #
#=================================================================#
# Bin scale = 0.100000
#    200|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    180|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    160|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    140|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |****|    |    |****|    |    |    |
#    100|****|    |    |****|****|    |****|    |****|****|
#       |****|    |****|****|****|****|****|****|****|****|
#     80|****|    |****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#     60|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#     40|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#     20|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================#
#=================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=================================================================#
      diehard_operm5|   0|   1000000|    1000|0.41995260|  PASSED
```

Von den 61 GSL-Zufallsgeneratoren (GSL = GNU Scientific Library) in *dieharder* fallen bei der Verwendung der neuen Pseudoinverse sieben Generatoren durch. Darunter z.B. *randu*, ein Vorzeigezufallsgenerator, wenn es darum geht das Verhalten von schlechten Zufallsgeneratoren zu zeigen.

„...its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists!“ - Donald Knuth [6]

Ausblick

Die beweisbare Sicherheit von SWIFFT sollte auch in Zukunft Kryptologen beschäftigen. Das einfache Design und die elegante Berechnung sprechen für SWIFFT. Mit den hier vorgestellten Algorithmen ist es leicht möglich die Größe der Gitter und damit die Sicherheit flexibel zu erhöhen, ohne eine komplett neue Funktion zu entwerfen.

Dadurch, dass inzwischen viele Rechner mit mehreren Prozessoren ausgestattet sind, lässt sich die Berechnungsgeschwindigkeit nochmals erhöhen. Viele Berechnungen der einzelnen NTTs sind voneinander unabhängig und identisch und lassen sich parallel berechnen, so wie es auch schon in einer der SWIFFTX-Implementierungen in [2] geschieht.

Die Wahl einer Primzahl $p = 2^{2^n} + 1$ wie von den SWIFFT-Autoren gewählt, machte es möglich, die Berechnung der NTTs lediglich mit Additionen, Subtraktionen, Bitshifts und Modulrechnen zu berechnen. NTTs dieser Form werden auch Fermat-Transformierungen genannt. Interessant wäre die Alternative modulo einer Mersenne-Zahl $M_n = 2^n - 1$ zu rechnen. Die Multiplikationen bei der Berechnung der NTTs entsprechen dann Bitrotationen. Der NTT wird in diesem Fall Mersenne-Transformierung genannt [20].

Nach der Korrektur von Operm5 wäre das nächste Ziel, den Test zu verallgemeinern und einen Test OpermX zu programmieren, der mit Parameter X gestartet wird und Sortierungen von Wörtern der Länge X zählt und überprüft. Die dazugehörige Kovarianzmatrix (genauer: Pseudoinverse der Kovarianzmatrix) müsste dann mit jedem Programmaufruf berechnet werden. Der Code aus dieser Arbeit lässt noch viel Raum für Optimierungen. Sehr viele Teilsummen der Matrixeinträge sind identisch und müssen nicht jedesmal neu berechnet werden.

Literatur

[1] Stefan C. Agapie and Paula A. Whitlock. "Random packing of hyperspheres and Marsaglia's Parking Lot Test", 2011. `www.sci.brooklyn.cuny.edu/~whitlock/parking.pdf`. Zitiert auf der Seite 42.

[2] Yuriy Arbitman, Gil Dogon, Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifftx: A proposal for the sha-3 standard. Submission to NIST, 2008. `http://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifftx.pdf`. Zitiert auf den Seiten 11 und 60.

[3] Robert G. Brown. "DieHarder: A Gnu Public Licensed Random Number Tester", 2011. `http://www.phy.duke.edu/~rgb/General/dieharder.php`. Zitiert auf der Seite 38.

[4] Robert G. Brown. "Dieharder Testsuite", 2011. `http://www.phy.duke.edu/~rgb/General/dieharder.php`. Zitiert auf der Seite 38.

[5] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, 19:297, 1965. Zitiert auf der Seite 14.

[6] Donald E. Knuth. *The Art of Computer Programming, Vol.2: Seminumerical Algorithms*. Addison-Wesley (Reading MA), 1969. Zitiert auf der Seite 59.

[7] Carl Friedrich Gauß. *Theoria interpolationis methodo nova tractata*. Königliche Gesellschaft der Wissenschaften, Göttingen, 1866. Nachlass. Zitiert auf der Seite 14.

[8] Daniel Kehlmann. *Die Vermessung der Welt*. Rowohlt, 2006. Zitiert auf der Seite 14.

[9] Nicolaas Hendrik Kuiper. Tests concerning random points on a circle, 1962. Zitiert auf der Seite 37.

[10] George Marsaglia. A Current View of Random Number Generators. In L. (Lynne) Billard, editor, *Computer science and statistics: proceedings of the Sixteenth Symposium on the Interface, Atlanta, Georgia, March 1984*, pages 3–10, Amsterdam, The Netherlands, 1985. Elsevier Science Publishers B.V. Zitiert auf der Seite 40.

[11] George Marsaglia. "Diehard Testsuite", 2011. `http://www.stat.fsu.edu/pub/diehard/`. Zitiert auf den Seiten 38 und 44.

[12] George Marsaglia. "Randomness of Pi and Other Decimal Expansions", 2011. interstat.statjournals.net/YEAR/2005/articles/0510005.pdf. Zitiert auf der Seite 42.

[13] George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness, 2002. Zitiert auf der Seite 40.

[14] Alexander May. Public Key Kryptanalyse, 2008. www.cits.rub.de/imperia/md/content/may/pkk08/skript.pdf. Zitiert auf der Seite 2.

[15] National Institute of Standards and Technology. *FIPS-197*. NIST, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Zitiert auf der Seite 51.

[16] Spiegel Online. "Glücksspiel - In Israel fallen die gleichen Lottozahlen wie im Vormonat", 2010. http://www.spiegel.de/panorama/0,1518,723587,00.html. Zitiert auf der Seite 34.

[17] Prof. Johannes Buchmann, Patrick Schmidt. Post-Quantum Cryptography, 2010. www.cdc.informatik.tu-darmstadt.de/lehre/WS09_10/vorlesung/pqc_files/PQC.pdf. Zitiert auf der Seite 2.

[18] Spektrum. *Lexikon der Mathematik, Band 1-5*. Spektrum, 2001. Zitiert auf den Seiten 34, 35, 36 und 37.

[19] Wai Wan Tsang and Chi yin Pang. The Mathematics of the Overlapping Chi-square Test. Zitiert auf den Seiten 48, 49 und 52.

[20] Unbekannt. "Efficient Polynomial Multiplication Over Finite Fields With Using Discrete Fourier Transform, Special NTTs For Efficient Multiplication In GF(q to the m)", 2011. http://mcs.cankaya.edu.tr/ogrenciler/proje2009Guz/200422022KayhanISPIR/rapor.pdf. Zitiert auf der Seite 60.

[21] Jürgen Lehn, Helmut Wegmann. *Einführung in die Statistik*. Teubner, 2004. Zitiert auf den Seiten 35 und 36.

A NTT-Code

Sämtlicher NTT-Code in diesem Kapitel ist Java-Code!

Die Funktion `reverseBits` dreht den Bitvektor, der die Zahl `input` repräsentiert um. Die Zahl `twoToNumOfBits` gibt dabei an, wieviele Einträge der Bitvektor hat. Für `twoToNumOfBits` $= 3$ wird so aus `input` $= 1$ die Zahl 4.

```
static int reverseBits(int input, int twoToNumOfBits)
{
  int reversed = 0;

  for (input |= twoToNumOfBits; input > 1; input >>= 1)
  reversed = (reversed << 1) | (input & 1);

  return reversed;
}
```

Der folgende Algorithmus dreht nur einen Teil des Inputs. Durch ihn ist es nicht nur möglich bei Verwendung des Cooley-Tukey-NTTs mit Vorberechnungen auf das Bitdrehen in den äußeren NTTs zu verzichten, sondern auch bei der Berechnung welche vorberechneten Werte an die äußeren NTTs gehen.

```
/**
 * Beispiel:
 * Wir haben einen NTT mit Ordnung 24 und Vorberechnungsgröße a = 3.
 * Das heißt die äußeren NTTs haben Ordnung 24/3 = 8.
 *
 * Wir permutieren nur für die äußeren NTTs, also nur modulo 8.
 *
 * Zahlenbeispiel:
 * reverseParts(17,8) führt zu
 * 17 = 16 + 1 = 1 mod 8; 1 = 001; reverseBits -> 100 = 4; 16+4 = 20
 */
static int reverseParts(int input, int b)
{
  int temp = input - (input % b);
  input = input % b;
  input = reverseBits(input,b);
  input += temp;
  return input;
}
```

Dies ist die direkte Berechnung von $NTT(\texttt{nttinput})$. Es wird also für jedes i die Summe $\sum_{k=0}^{d-1} \texttt{nttinput}(k) \cdot \texttt{gamma}^{i \cdot k}$ berechnet.

```
static int[] simpleNTT(int[] nttinput,int gamma, int prime)
{
  int[]gammapowers = new int[nttinput.length];

  gammapowers[0] = 1;
  for (int i = 1; i < nttinput.length; i++)
  {
    gammapowers[i] = gammapowers[i-1]*gamma %prime;
  }
  int[] nttoutput = new int[nttinput.length];

  for (int i0=0; i0 < nttinput.length; i0++)
```

```
  {
    int sum = 0;
    for (int k0=0; k0 < nttinput.length; k0++)
    {
      sum += (nttinput[k0]*gammapowers[k0*i0 %nttinput.length]) %prime;
    }
    nttoutput[i0] = sum %prime;
  }

  return nttoutput;
}
```

Die Funktion `cooleyTukey` nutzt das in Kapitel 4.2 vorgestellte Cooley-Tukey-Netzwerk um $NTT($`nttinput`$)$ zu berechnen.

```
static int[] cooleyTukey(int[] nttinput, int gamma,int prime)
// Mit Indexpermutation, in-place-operations
// Ohne Vorberechnungen
{
  int i = 0;
  int j = 0;
  int k = 0;
  int jump = 0; // Die Sprungweite in den NTT-Ebenen
  int temp = 0;
  int iter = 0; // Wird vom Twiddlefaktor benutzt
  int twiddle = 0; // Der Vorberechnungsfaktor, "Twiddle"-Faktor
  int[] nttoutput = new int[nttinput.length];
  int[]gammapowers = new int[nttinput.length];

  gammapowers[0] = 1;
  for (i = 1; i < nttinput.length; i++)
  {
    gammapowers[i] = gammapowers[i-1]*gamma %prime;
  }
    // (Bitpermutation vom Index)
  for (i = 0; i<nttinput.length; i++)
  {
    nttoutput[i] = nttinput[reverseBits(i,nttinput.length)];
  }
    // Der Algorithmus:
  long n = Math.round(Math.log(nttinput.length)/Math.log(2));
  for (i = 0; i<n; i++)
  {
    k = 0;
    for (jump = 0; jump < (nttinput.length / Math.pow(2,i+1) ); jump++)
    {
      iter = 0;
      for (j = k; j<k+( Math.pow(2, i) ); j++)
      {
        // TWIDDLEN:
        // nttoutput[j + 2^i] = gamma^(iter * 2^[log_2(length)-i-1])
        //                      * nttoutput[j + 2^i];
        twiddle = (int) (gammapowers[iter * (int)Math.pow(2,n-i-1) %nttinput.length])
            %prime;
        nttoutput[j + (int)Math.pow(2, i)] = twiddle * nttoutput[j + (int)Math.pow(2,
            i)] %prime;
        // ADDIEREN, SUBTRAHIEREN
        temp = nttoutput[j];
        nttoutput[j] = (nttoutput[j] + nttoutput[j + (int)Math.pow(2, i)]) %prime;
```

```
        nttoutput[j + (int)Math.pow(2, i)] = (temp - nttoutput[j + (int)Math.pow(2, i)
            ]) %prime;
        iter++;
      }
      k = k + (int)Math.pow(2, i+1);
    }
  }
  return nttoutput;
}
```

Der `revCooleyTukey` ist der `cooleyTukey` ohne die Bitpermutationen. Die Funktion `revCooleyTukey` wird von den Funktionen benutzt, die Vorberechnungen durchführen. Die Permutationen werden dann in den Vorberechnungen durchgeführt.

```
static int[] revCooleyTukey(int[] nttinput, int gamma,int prime)
// Mit Indexpermutation, in-place-operations
// Ohne Vorberechnungen
{
  int i,j,k = 0;
  int jump = 0; // Die Sprungweite in den NTT-Ebenen
  int temp = 0;
  int iter = 0; // Wird vom Twiddlefaktor benutzt
  int twiddle = 0; // Der Vorberechnungsfaktor, "Twiddle"-Faktor
  int[]gammapowers = new int[nttinput.length];
  gammapowers[0] = 1;
  for (i = 1; i < nttinput.length; i++)
  {
    gammapowers[i] = gammapowers[i-1]*gamma %prime;
  }
  long n = Math.round(Math.log(nttinput.length)/Math.log(2));
  for (i = 0; i<n; i++)
  {
    k = 0;
    for (jump = 0; jump < (nttinput.length / Math.pow(2,i+1) ); jump++)
    {
      iter = 0;
      for (j = k; j<k+( Math.pow(2, i) ); j++)
      {
        twiddle = (int) (gammapowers[iter * (int)Math.pow(2,n-i-1) %nttinput.length])
            %prime;
        nttinput[j + (int)Math.pow(2, i)] = twiddle * nttinput[j + (int)Math.pow(2, i)
            ] %prime;
        temp = nttinput[j];
        nttinput[j] = (nttinput[j] + nttinput[j + (int)Math.pow(2, i)]) %prime;
        nttinput[j + (int)Math.pow(2, i)] = (temp - nttinput[j + (int)Math.pow(2, i)])
             %prime;
        iter++;
      }
      k = k + (int)Math.pow(2, i+1);
    }
  }
  return nttinput;
}
```

Der `precompNTT` ermöglicht Vorberechnungen wie sie in Kapitel 6 vorgestellt wurden. Mit `precompLevel` wird festgelegt, wieviel vorberechnet werden soll und entspricht a in 6. Die Größe der äußeren NTTs ist dann $d/a = b$. Da für die äußeren NTTs der Cooley-Tukey-Algorithmus verwendet wird, muss `precompLevel` so gewählt sein, dass

`b` eine Zweierpotenz ist. Im Gegensatz zu den Funktionen `simpleNTT`, `cooleyTukey` und `revCooleyTukey` darf der Eingabevektor `nttinput` nur aus Nullen und Einsen bestehen. Die Vorberechnungen in `precompNTT` beinhalten bereits die Multiplikationen der Elemente aus `nttinput` mit den Potenzen von `omega`.

Die Vorberechnungen wurden in den Code integriert. Für die Zeitmessungen wurden sie herausgenommen und mit an die Funktion gegeben.

```
static int[] precompNTT(int[] nttinput, int gamma,int omega,int prime, int
    precompLevel)
// nttinput nur 0 und 1!
// für a = 8, nttinput.length = 64 ist dies der SWIFFT-Fall
{ int i0,k1,k0 = 0;
  int x = 0;
  int a = precompLevel;   // precompLevel so wählen, dass b eine Zweierpotenz ist!
  int b = nttinput.length/a; // der äußere NTT hat Ordnung b
  int db = nttinput.length/b;
  int[] innerSumoutput = new int[a];
  int[] outerNTTinput = new int[b];
  int[] outerNTToutput = new int[b];
  int[] nttoutput = new int[nttinput.length];
  int[] omegapowers = new int[2*nttinput.length];
  int[] gammapowers = new int[nttinput.length];
  int[][] multipliers = new int[a][b]; // i0 k0
  int[][] precompSum = new int[a][(int)Math.pow(2,a)];

//  Über die Permutationen:
//  Die k1-Komponente wird NICHT permutiert
//  Die k0-Komponente wird in den multipliers permutiert

  // BEGINN DER VORBERECHNUNGEN
  omegapowers[0] = 1;
  for (int i = 1; i < 2*nttinput.length; i++)
  {
    omegapowers[i] = omegapowers[i-1]*omega %prime;
  }
  gammapowers[0] = 1;
  for (int i = 1; i < nttinput.length; i++)
  {
    gammapowers[i] = gammapowers[i-1]*gamma %prime;
  }
  // Die multipliers
  for (k0 = 0; k0 < b; k0++)
  {
    for (i0 = 0; i0 < db; i0++)
    {
      multipliers[i0][reverseBits(k0,b)] = omegapowers[(2*i0 + 1) * k0 %(2*nttinput.
          length)] %prime;
    }
  }
  // Das Bilden der inneren 2^a Summen
  // (Jede innere Summe hat a outputs)
  for (x = 0; x < Math.pow(2,a); x++)
  {
    for (i0 = 0; i0<db; i0++)
    {
      int temp = 0;
      for (k1 = 0; k1<db; k1++)
        {
```

```
          temp += omegapowers[(2*i0 +1)*b*k1 %(2*nttinput.length)]*((x >> k1) & 1) %
              prime;
        }
      innerSumoutput[i0] = temp;
    }
    for (i0 = 0; i0<db; i0++)
    {
      precompSum[i0][x] = innerSumoutput[i0];
    }
  } // ENDE DER VORBERECHNUNGEN

  // Jetzt die äußeren NTTs mithilfe von multiplieres und precompNTT
  for (i0 = 0; i0 < a; i0++)
  {
    for (k0 = 0; k0 < b; k0++)
    {
      x = 0;
      for (k1 = 0; k1<db; k1++)
      {
        x +=(int)Math.pow(2,k1) * nttinput[k0 + b*k1];
      }
      outerNTTinput[k0] = multipliers[i0][k0] * precompSum[i0][x] %prime;
    }
    outerNTToutput = revCooleyTukey(outerNTTinput,gammapowers[a],prime);
    for (k0 = 0; k0 < b; k0++)
    {
      nttoutput[i0 + a * k0] = outerNTToutput[k0] %prime;
    }
  }
  return nttoutput;
}
```

Dieser Code soll verständlicher machen, wie die Algorithmen funktionieren:

```
static void Vergleiche()
{
  // NTT mit Ordnung 16:
  // precomp-Level muss eine Zweierpotenz sein!
  int[] nttinput = {1,0,1,1,1,1,0,0,1,0,1,1,1,1,0,1};
  int gamma = 225;  // Ordnung 16
  int omega = 15; // Ordnung 32
  int prime = 257;
  // NTT mit Ordnung 24:
  // precomp-Level muss die Form 3 * 2^k haben!
//  int[] nttinput = {1,0,1,1,1,1,0,0,1,0,1,0,1,0,1,1,1,0,1,0,1,0,1,0}; // 24 = 8*3
//  int gamma = 4; // Ordnung 24
//  int omega = 2; // Ordnung 48
//  int prime = 97;

  int precompLevel = 4; // Muss nttinput.length teilen
  int[] nttoutput = new int[nttinput.length];
  int[] nttinputtest = new int[nttinput.length];
  int[] revnttinput = new int[nttinput.length];
  int[] omegapowers = new int[2*nttinput.length];

  System.out.format("\nNTT-Größe:␣%d",nttinput.length);


  omegapowers[0] = 1;
```

```
  for (int i = 1; i < nttinput.length; i++)
  {
    omegapowers[i] = omegapowers[i-1]*omega %prime;
  }
  // das Multiplizieren mit den Omegas
  for (int i = 0; i < nttinput.length; i++)
  {
    nttinputtest[i] = omegapowers[i] * nttinput[i];
  }
  // Der simple NTT
  nttoutput = simpleNTT(nttinputtest,gamma, prime);
  System.out.format("\nSimpleNTT-output:\n");
  for (int i = 0; i < nttoutput.length; i++)
  {
    System.out.format("%d ", nttoutput[i]);
  }

  // Der Cooley-Tukey-NTT
  nttoutput = cooleyTukey(nttinputtest,gamma, prime);
  System.out.format("\nCooley-Tukey-NTT-output:\n");
  for (int i = 0; i < nttoutput.length; i++)
  {
    System.out.format("%d ", nttoutput[i]);
  }

  // Der precomp-NTT
  // Es wird reverseBlocks auf nttinput angewandt, nicht auf nttinputtest !!!
  for (int i = 0; i < nttinput.length; i++)
  {
    revnttinput[i] = nttinput[reverseParts(i,nttinput.length/precompLevel)];
  }
  nttoutput = precompNTT(revnttinput, gamma, omega, prime, precompLevel);
  System.out.format("\nprecompNTT-output:\n");
  for (int i = 0; i < nttoutput.length; i++)
  {
    System.out.format("%d ", nttoutput[i]);
  }
}
```

Führen wir `Vergleiche` aus, erhalten wir die Ausgabe

```
NTT-Größe: 16
SimpleNTT-output:
128 120 197 31 232 26 84 20 224 243 41 58 240 50 18 103
Cooley-Tukey-NTT-output:
128 -137 -60 31 -25 26 -173 -237 -33 243 41 -199 -17 -207 -239 103
Vorberechnungs-NTT-output:
128 120 197 31 -25 26 -173 -237 -33 -14 41 58 -17 50 18 103
```

Die Ergebnisse stimmen modulo 257 überein. Falls das andere NTT-Beispiel gewählt wird, muss der Cooley-Tukey-Algorithmus ausgeklammert werden, da 24 keine Zweierpotenz ist!

B Operm5-Code

Sämtlicher Code zur Berechnung der Kovarianzmatrix ist Mathematica-Code!

```
Rev[a_] := (*dreht den Vektor um, speichert den umgedrehten Vektor in \
temp*)
  (
  temp = a;
  For[i = 1, i <= 5, i++,
   temp[[i]] = a[[6 - i]];
   ];
  Return[temp];
  )
```

`PossibleOverlap` ist die in 8.4 beschriebene Funktion, die testet, ob zwei Sortierungen überlappen können.

```
(*Falls Overlap gleich 2,3,4, teste hier ob Überlappung möglich ist*)
PossibleOverlap[a_, b_, overlap_] :=
(
overlapispossible = True;
Clear [partSort1];
partSort1 = a;
For[i = overlap + 1, i <= 5, i++,
  For[j = 1, j <= 5, j++,
    If[partSort1[[j]] > partSort1[[i]],
      partSort1[[j]] = partSort1[[j]] - 1;
    ]
  ]
];
Clear [partSort2];
partSort2 = b;
For[i = 1, i <= 5 - overlap, i++,
  For[j = 1, j <= 5, j++,
    If[partSort2[[j]] > partSort2[[i]],
      partSort2[[j]] = partSort2[[j]] - 1;
    ]
    ]
];
(*Ist eine Überlappung also möglich?*)
For[i = 1, i <= overlap, i++,
  If[partSort1[[i]] !=  partSort2[[i + 5 - overlap]],
    overlapispossible = False;
  ]
];
Return[overlapispossible];
```

Dies sind `Overlap1`,`Overlap2`,`Overlap3`,`Overlap4` und `Overlap5` wie sie in 8.4 beschrieben sind.

```
Overlap1[a_, b_] := (*falls a und b einfach überlappen*)
(
NrBigger = 4-b[[5]]; (*so viele Zahlen von b sind größer als die fünfte*)
NrSmaller = b[[5]]; (*so viele Zahlen von b sind kleiner als die fünfte*)
Sum[
 Sum[
  Sum[
   Sum[
    Switch[a[[1]],
     0, (Binomial[M - k - 4, NrBigger]*Binomial[k, NrSmaller]),
     1, (Binomial[M - l - 3, NrBigger]*
       Binomial[l - 1, NrSmaller]),
     2, (Binomial[M - m - 2, NrBigger]*
       Binomial[m - 2, NrSmaller]),
     3, (Binomial[M - n - 1, NrBigger]*
       Binomial[n - 3, NrSmaller]),
     4, (Binomial[M - o, NrBigger]*Binomial[o - 4, NrSmaller])
    ],
   {o, n + 1, M}],
  {n, m + 1, M}],
 {m, l + 1, M}],
{l, k + 1, M}],
{k, 0, M}]
)
```

```
Overlap2[a_, b_] := (*falls a und b zweifach überlappen*)
(
If[b[[4]] > b[[5]], shift = 0, shift = 1];
(*b[[4+shift]] ist größer als b[[5-shift]]*)
(*also ist auch a[[1+shift]] größer als a[[2-shift]]*)
NrSmaller = b[[5 - shift]];
NrBigger = 4 - b[[4 + shift]];
NrBetween = b[[4 + shift]] - b[[5 - shift]] - 1;
Sum[
 Sum[
  Sum[
   Sum[
    Switch[a[[2 - shift]], (*die kleinere Zahl, kann nicht 4 sein*)
     0, Switch[a[[1 + shift]],
      1, (Binomial[M - l - 3, NrBigger]*Binomial[k, NrSmaller]*
        Binomial[l - k - 1, NrBetween]),
      2, (Binomial[M - m - 2, NrBigger]*Binomial[k, NrSmaller]*
        Binomial[m - k - 1 - 1, NrBetween]),
      3, (Binomial[M - n - 1, NrBigger]*Binomial[k, NrSmaller]*
        Binomial[n - k - 1 - 2, NrBetween]),
      4, (Binomial[M - o, NrBigger]*Binomial[k, NrSmaller]*
        Binomial[o - k - 1 - 3, NrBetween])
      ],
     1,
     Switch[a[[1 + shift]], (*die größere Zahl,
      kann nicht 0 und muss größer als die andere sein*)
      2, (Binomial[M - m - 2, NrBigger]*Binomial[l - 1, NrSmaller]*
        Binomial[m - l - 1, NrBetween]),
      3, (Binomial[M - n - 1, NrBigger]*Binomial[l - 1, NrSmaller]*
        Binomial[n - l - 1 - 1, NrBetween]),
      4, (Binomial[M - o, NrBigger]*Binomial[l - 1, NrSmaller]*
        Binomial[o - l - 1 - 2, NrBetween])
      ],
     2, Switch[a[[1 + shift]],
      3, (Binomial[M - n - 1, NrBigger]*Binomial[m - 2, NrSmaller]*
        Binomial[n - m - 1, NrBetween]),
      4, (Binomial[M - o, NrBigger]*Binomial[m - 2, NrSmaller]*
        Binomial[o - m - 1 - 1, NrBetween])
      ],
     3, (Binomial[M - o, NrBigger]*Binomial[n - 3, NrSmaller]*
       Binomial[o - n - 1, NrBetween])
     ],
    {o, n + 1, M}],
   {n, m + 1, M}],
  {m, l + 1, M}],
 {l, k + 1, M}],
{k, 0, M}]
)
```

```
Overlap3[a_, b_] :=
(
Switch[partSort2[[3]],
 0,
 If[partSort2[[4]] < partSort2[[5]], (* also {x,x,0,1,2}*)
  shiftsmall = 0; shiftmiddle = 1; shiftbig = 2, (*{x,x,0,2,1}*)
  shiftsmall = 0; shiftmiddle = 2; shiftbig = 1;
```

```
  ],
 1,
 If[partSort2[[4]] < partSort2[[5]], (* also {x,x,1,0,2}*)
  shiftsmall = 1; shiftmiddle = 0; shiftbig = 2, (*{x,x,1,2,0}*)
  shiftsmall = 2; shiftmiddle = 0; shiftbig = 1;
  ],
 2,
 If[partSort2[[4]] < partSort2[[5]], (* also {x,x,2,0,1}*)
  shiftsmall = 1; shiftmiddle = 2; shiftbig = 0, (* also {x,x,2,1,0}*)
  shiftsmall = 2; shiftmiddle = 1; shiftbig = 0;
  ]
 ];
NrSmaller = b[[3 + shiftsmall]];
NrBigger = 4 - b[[3 + shiftbig]];
NrSmallBetween = b[[3 + shiftmiddle]] - b[[3 + shiftsmall]] - 1;
NrBigBetween = b[[3 + shiftbig]] - b[[3 + shiftmiddle]] - 1;
Sum[
 Sum[
  Sum[
   Sum[
    Sum[
     Switch[a[[1 + shiftsmall]], (*die kleinste Zahl, kann nicht 3,
      4 sein*)
      0,
      Switch[a[[1 + shiftmiddle]], (*die mittlere Zahl,
       kann nicht 1,4 sein, muss größer als die kleinste sein*)
       1,
       Switch[a[[1 + shiftbig]], (*die größte Zahl, kann nicht 0,1
        sein und muss größer als die anderen sein*)
        2, (Binomial[M - m - 2, NrBigger]*Binomial[k, NrSmaller]*
          Binomial[m - l - 1, NrBigBetween]*
          Binomial[l - k - 1, NrSmallBetween]) ,
        3, (Binomial[M - n - 1, NrBigger]*Binomial[k, NrSmaller]*
          Binomial[n - l - 1 - 1, NrBigBetween]*
          Binomial[l - k - 1, NrSmallBetween]) ,
        4, (Binomial[M - o, NrBigger]*Binomial[k, NrSmaller]*
          Binomial[o - l - 1 - 2, NrBigBetween]*
          Binomial[l - k - 1, NrSmallBetween])
        ],
       2, Switch[a[[1 + shiftbig]],
        3, (Binomial[M - n - 1, NrBigger]*Binomial[k, NrSmaller]*
          Binomial[n - m - 1, NrBigBetween]*
          Binomial[m - k - 1 - 1, NrSmallBetween]) ,
        4, (Binomial[M - o, NrBigger]*Binomial[k, NrSmaller]*
          Binomial[o - m - 1 - 1, NrBigBetween]*
          Binomial[m - k - 1 - 1, NrSmallBetween])
        ],
       3,  (Binomial[M - o, NrBigger]*Binomial[k, NrSmaller]*
         Binomial[o - n - 1, NrBigBetween]*
         Binomial[n - k - 1 - 2, NrSmallBetween])
       ],
      1,
      Switch[a[[1 + shiftmiddle]], (*die mittlere Zahl,
       kann nicht 1,4 sein, muss größer als die kleinste sein*)
       2,
       Switch[a[[1 + shiftbig]], (*die größte Zahl, kann nicht 0,
        1 sein und muss größer als die anderen sein*)
        3, (Binomial[M - n - 1, NrBigger]*
          Binomial[l - 1, NrSmaller]*
          Binomial[n - m - 1, NrBigBetween]*
          Binomial[m - l - 1, NrSmallBetween]) ,
        4, (Binomial[M - o, NrBigger]*Binomial[l - 1, NrSmaller]*
          Binomial[o - m - 1 - 1, NrBigBetween]*
          Binomial[m - l - 1, NrSmallBetween])
        ],
       3,  (Binomial[M - o, NrBigger]*Binomial[l - 1, NrSmaller]*
         Binomial[o - n - 1, NrBigBetween]*
         Binomial[n - l - 1 - 1, NrSmallBetween])
       ],
      2, (Binomial[M - o, NrBigger]*Binomial[m - 2, NrSmaller]*
        Binomial[o - n - 1, NrBigBetween]*
        Binomial[n - m - 1, NrSmallBetween])
      ],
     {o, n + 1, M}],
    {n, m + 1, M}],
   {m, l + 1, M}],
  {l, k + 1, M}],
 {k, 0, M}]
)
```

```
Overlap4[a_, b_] :=
(
Switch[partSort2[[2]],
 0,
 Switch[partSort2[[3]],
  1, If[partSort2[[4]] < partSort2[[5]], (*{x,0,1,2,3}*)
```

```
  shiftsmallest = 0; shiftsmall = 1; shiftbig = 2;
  shiftbiggest = 3;
  ,(*{x,0,1,3,2}*)
  shiftsmallest = 0; shiftsmall = 1; shiftbig = 3;
  shiftbiggest = 2;
  ],
 2, If[partSort2[[4]] < partSort2[[5]], (*{x,0,2,1,3}*)
  shiftsmallest = 0; shiftsmall = 2; shiftbig = 1;
  shiftbiggest = 3;
  ,(*{x,0,2,3,1}*)
  shiftsmallest = 0; shiftsmall = 3; shiftbig = 1;
  shiftbiggest = 2;
  ],
 3, If[partSort2[[4]] < partSort2[[5]], (*{x,0,3,1,2}*)
  shiftsmallest = 0; shiftsmall = 2; shiftbig = 3;
  shiftbiggest = 1;
  ,(*{x,0,3,2,1}*)
  shiftsmallest = 0; shiftsmall = 3; shiftbig = 2;
  shiftbiggest = 1;
  ]
 ],
1,
Switch[partSort2[[3]],
 0, If[partSort2[[4]] < partSort2[[5]], (*{x,1,0,2,3}*)
  shiftsmallest = 1; shiftsmall = 0; shiftbig = 2;
  shiftbiggest = 3;
  ,(*{x,1,0,3,2}*)
  shiftsmallest = 1; shiftsmall = 0; shiftbig = 3;
  shiftbiggest = 2;
  ],
 2, If[partSort2[[4]] < partSort2[[5]], (*{x,1,2,0,3}*)
  shiftsmallest = 2; shiftsmall = 0; shiftbig = 1;
  shiftbiggest = 3;
  ,(*{x,1,2,3,0}*)
  shiftsmallest = 3; shiftsmall = 0; shiftbig = 1;
  shiftbiggest = 2;
  ],
 3, If[partSort2[[4]] < partSort2[[5]], (*{x,1,3,0,2}*)
  shiftsmallest = 2; shiftsmall = 0; shiftbig = 3;
  shiftbiggest = 1;
  ,(*{x,1,3,2,0}*)
  shiftsmallest = 3; shiftsmall = 0; shiftbig = 2;
  shiftbiggest = 1;
  ]
 ],
2,
Switch[partSort2[[3]],
 0, If[partSort2[[4]] < partSort2[[5]], (*{x,2,0,1,3}*)
  shiftsmallest = 1; shiftsmall = 2; shiftbig = 0;
  shiftbiggest = 3;
  ,(*{x,2,0,3,1}*)
  shiftsmallest = 1; shiftsmall = 3; shiftbig = 0;
  shiftbiggest = 2;
  ],
 1, If[partSort2[[4]] < partSort2[[5]], (*{x,2,1,0,3}*)
  shiftsmallest = 2; shiftsmall = 1; shiftbig = 0;
  shiftbiggest = 3;
  ,(*{x,2,1,3,0}*)
  shiftsmallest = 3; shiftsmall = 1; shiftbig = 0;
  shiftbiggest = 2;
  ],
 3, If[partSort2[[4]] < partSort2[[5]], (*{x,2,3,0,1}*)
  shiftsmallest = 2; shiftsmall = 3; shiftbig = 0;
  shiftbiggest = 1;
  ,(*{x,2,3,1,0}*)
  shiftsmallest = 3; shiftsmall = 2; shiftbig = 0;
  shiftbiggest = 1;
  ]
 ],
3,
Switch[partSort2[[3]],
 0, If[partSort2[[4]] < partSort2[[5]], (*{x,3,0,1,2}*)
  shiftsmallest = 1; shiftsmall = 2; shiftbig = 3;
  shiftbiggest = 0;
  ,(*{x,3,0,2,1}*)
  shiftsmallest = 1; shiftsmall = 3; shiftbig = 2;
  shiftbiggest = 0;
  ],
 1, If[partSort2[[4]] < partSort2[[5]], (*{x,3,1,0,2}*)
  shiftsmallest = 2; shiftsmall = 1; shiftbig = 3;
  shiftbiggest = 0;
  ,(*{x,3,1,2,0}*)
  shiftsmallest = 3; shiftsmall = 1; shiftbig = 2;
  shiftbiggest = 0;
  ],
 2, If[partSort2[[4]] < partSort2[[5]], (*{x,3,2,0,1}*)
```

```
   shiftsmallest = 2; shiftsmall = 3; shiftbig = 1;
   shiftbiggest = 0;
   ,(*{x,3,2,1,0}*)
   shiftsmallest = 3; shiftsmall = 2; shiftbig = 1;
   shiftbiggest = 0;
   ]
  ]
 ];
NrSmaller = b[[2 + shiftsmallest]];
NrSmallBetween = b[[2 + shiftsmall]] - b[[2 + shiftsmallest]] - 1;
NrMiddleBetween = b[[2 + shiftbig]] - b[[2 + shiftsmall]] - 1;
NrBigBetween = b[[2 + shiftbiggest]] - b[[2 + shiftbig]] - 1;
NrBigger = 4 - b[[2 + shiftbiggest]];
Sum[
 Sum[
  Sum[
   Sum[
    Sum[
     Switch[a[[1 + shiftsmallest]], (*die kleinste Zahl,
      kann nicht 2,3,4 sein*)
      0,
      Switch[a[[1 + shiftsmall]], (*die zweitkleinste Zahl,
       kann nicht 1,3,4 sein, muss größer als die kleinste sein*)
       1,
       Switch[a[[1 + shiftbig]], (*die zweitgrößte Zahl,
        kann nicht 0,1,4 sein und muss größer als die anderen sein*)
        2,
        Switch[a[[1 + shiftbiggest]], (*die drittgrößte Zahl,
         kann nicht 0,1,2 sein und muss größer als die anderen sein*)
         3, (Binomial[M - n - 1, NrBigger]*
           Binomial[n - m - 1, NrBigBetween]*
           Binomial[m - l - 1, NrMiddleBetween]*
           Binomial[l - k - 1, NrSmallBetween]*
           Binomial[k, NrSmaller]) ,
         4, (Binomial[M - o, NrBigger]*
           Binomial[o - m - 1 - 1, NrBigBetween]*
           Binomial[m - l - 1, NrMiddleBetween]*
           Binomial[l - k - 1, NrSmallBetween]*
           Binomial[k, NrSmaller])
         ],
        3, (Binomial[M - o, NrBigger]*
          Binomial[o - n - 1, NrBigBetween]*
          Binomial[n - l - 1 - 1, NrMiddleBetween]*
          Binomial[l - k - 1, NrSmallBetween]*
          Binomial[k, NrSmaller])
        ],
       2, (Binomial[M - o, NrBigger]*
         Binomial[o - n - 1, NrBigBetween]*
         Binomial[n - m - 1, NrMiddleBetween]*
         Binomial[m - k - 1 - 1, NrSmallBetween]*
         Binomial[k, NrSmaller])
       ],
      1, (Binomial[M - o, NrBigger]*
        Binomial[o - n - 1, NrBigBetween]*
        Binomial[n - m - 1, NrMiddleBetween]*
        Binomial[m - l - 1, NrSmallBetween]*
        Binomial[l - 1, NrSmaller])
      ],
     {o, n + 1, M}],
    {n, m + 1, M}],
   {m, l + 1, M}],
  {l, k + 1, M}],
 {k, 0, M}]
)
```

```
Overlap5[a_, b_] :=
(
 Sum[
  Sum[
   Sum[
    Sum[
     Sum[
      1,
     {o, n + 1, M}],
    {n, m + 1, M}],
   {m, l + 1, M}],
  {l, k + 1, M}],
 {k, 0, M}]
)
```

Die Funktion `AddSums` addiert die Wahrscheinlichkeiten aller möglichen Überlappungen. Dazu testet sie zunächst mit `PossibleOverlap` ob die Sortierungen überlappen können. Anschließend wir die Zahl der möglichen Überlappungen durch die

Gesamtzahl geteilt.

```
AddSums[a_, b_] :=
(
Summe = Overlap1[a,b]/
  ((M + 1) M (M - 1) (M - 2) (M - 3) (M - 4) (M - 5) (M - 6) (M - 7));
If[PossibleOverlap[a, b, 2],
 Summe = Summe + Overlap2[a,b]/
  ((M + 1) M (M - 1) (M - 2) (M - 3) (M - 4) (M - 5) (M - 6));
 ];
If[PossibleOverlap[a, b, 3],
 Summe = Summe + Overlap3[a,b] /
 ((M + 1) M (M - 1) (M - 2) (M - 3) (M - 4) (M - 5));
 ];
If[PossibleOverlap[a, b, 4],
 Summe = Summe + Overlap4[a,b] /
 ((M + 1) M (M - 1) (M - 2) (M - 3) (M - 4));
 ];
If[a == b,
 Summe = Summe + (Overlap5[a, b]/ ((M + 1) M (M - 1) (M - 2) (M - 3)));
 ];
reva = Rev[a]; (*ab hier ist a gedreht*)
revb = Rev[b]; (*ab hier ist b gedreht*)
If[PossibleOverlap[reva, revb, 4],
 Summe = Summe + Overlap4[reva, revb]/ ((M + 1) M (M - 1) (M - 2) (M - 3)
       (M - 4));
 ];
If[PossibleOverlap[reva, revb, 3],
 Summe = Summe + Overlap3[reva, revb]/ ((M + 1) M (M - 1) (M - 2) (M - 3)
       (M - 4) (M - 5));
 ];
If[PossibleOverlap[reva, revb, 2],
 Summe = Summe + Overlap2[reva, revb]/ ((M + 1) M (M - 1) (M - 2) (M - 3)
       (M - 4) (M - 5) (M - 6));
 ];
Summe = Summe + Overlap1[reva, revb]/ ((M + 1) M (M - 1) (M - 2) (M - 3)
     (M - 4) (M - 5) (M - 6) (M - 7));
Summe (*Die Ausgabe*)
)
```

Die Sortierungen, ihrer Sortierungsnummer nach geordnet:

```
Sortierungen = {
{4, 0, 1, 2, 3}, {4, 1, 0, 2, 3}, {4, 2, 0, 1, 3}, {4, 2, 1, 0, 3},
{4, 0, 2, 1, 3}, {4, 1, 2, 0, 3}, {4, 3, 1, 0, 2}, {4, 3, 0, 1, 2},
{4, 3, 0, 2, 1}, {4, 3, 1, 2, 0}, {4, 3, 2, 0, 1}, {4, 3, 2, 1, 0},
{4, 0, 3, 1, 2}, {4, 1, 3, 0, 2}, {4, 2, 3, 0, 1}, {4, 2, 3, 1, 0},
{4, 0, 3, 2, 1}, {4, 1, 3, 2, 0}, {4, 0, 1, 3, 2}, {4, 1, 0, 3, 2},
{4, 2, 0, 3, 1}, {4, 2, 1, 3, 0}, {4, 0, 2, 3, 1}, {4, 1, 2, 3, 0},
{3, 4, 1, 2, 0}, {3, 4, 0, 2, 1}, {3, 4, 0, 1, 2}, {3, 4, 1, 0, 2},
{3, 4, 2, 1, 0}, {3, 4, 2, 0, 1}, {2, 4, 1, 0, 3}, {2, 4, 0, 1, 3},
{1, 4, 0, 2, 3}, {0, 4, 1, 2, 3}, {1, 4, 2, 0, 3}, {0, 4, 2, 1, 3},
{2, 4, 3, 1, 0}, {2, 4, 3, 0, 1}, {1, 4, 3, 0, 2}, {0, 4, 3, 1, 2},
{1, 4, 3, 2, 0}, {0, 4, 3, 2, 1}, {2, 4, 1, 3, 0}, {2, 4, 0, 3, 1},
{1, 4, 0, 3, 2}, {0, 4, 1, 3, 2}, {1, 4, 2, 3, 0}, {0, 4, 2, 3, 1},
{3, 0, 4, 2, 1}, {3, 1, 4, 2, 0}, {3, 2, 4, 1, 0}, {3, 2, 4, 0, 1},
{3, 0, 4, 1, 2}, {3, 1, 4, 0, 2}, {2, 3, 4, 0, 1}, {2, 3, 4, 1, 0},
{1, 3, 4, 2, 0}, {0, 3, 4, 2, 1}, {1, 3, 4, 0, 2}, {0, 3, 4, 1, 2},
{2, 0, 4, 1, 3}, {2, 1, 4, 0, 3}, {1, 2, 4, 0, 3}, {0, 2, 4, 1, 3},
{1, 0, 4, 2, 3}, {0, 1, 4, 2, 3}, {2, 0, 4, 3, 1}, {2, 1, 4, 3, 0},
{1, 2, 4, 3, 0}, {0, 2, 4, 3, 1}, {1, 0, 4, 3, 2}, {0, 1, 4, 3, 2},
{3, 0, 1, 4, 2}, {3, 1, 0, 4, 2}, {3, 2, 0, 4, 1}, {3, 2, 1, 4, 0},
{3, 0, 2, 4, 1}, {3, 1, 2, 4, 0}, {2, 3, 1, 4, 0}, {2, 3, 0, 4, 1},
{1, 3, 0, 4, 2}, {0, 3, 1, 4, 2}, {1, 3, 2, 4, 0}, {0, 3, 2, 4, 1},
{2, 0, 3, 4, 1}, {2, 1, 3, 4, 0}, {1, 2, 3, 4, 0}, {0, 2, 3, 4, 1},
{1, 0, 3, 4, 2}, {0, 1, 3, 4, 2}, {2, 0, 1, 4, 3}, {2, 1, 0, 4, 3},
{1, 2, 0, 4, 3}, {0, 2, 1, 4, 3}, {1, 0, 2, 4, 3}, {0, 1, 2, 4, 3},
{3, 0, 1, 2, 4}, {3, 1, 0, 2, 4}, {3, 2, 0, 1, 4}, {3, 2, 1, 0, 4},
{3, 0, 2, 1, 4}, {3, 1, 2, 0, 4}, {2, 3, 1, 0, 4}, {2, 3, 0, 1, 4},
{1, 3, 0, 2, 4}, {0, 3, 1, 2, 4}, {1, 3, 2, 0, 4}, {0, 3, 2, 1, 4},
{2, 0, 3, 1, 4}, {2, 1, 3, 0, 4}, {1, 2, 3, 0, 4}, {0, 2, 3, 1, 4},
{1, 0, 3, 2, 4}, {0, 1, 3, 2, 4}, {2, 0, 1, 3, 4}, {2, 1, 0, 3, 4},
{1, 2, 0, 3, 4}, {0, 2, 1, 3, 4}, {1, 0, 2, 3, 4}, {0, 1, 2, 3, 4}
};
```

Anschließend lässt sich die Kovarianzmatrix mit Theorem 8.3 berechnen:

```
Kovarianzmatrix =
 For[g = 1, g <= 120, g++,
  For[h = 1, h <= 120, h++,
   Kovarianzmatrix[[g, h]] = AddSums[Sortierungen[[g]], Sortierungen[[h
        ]]] - (9/(120*120));
  ];
  Print[Kovarianzmatrix];
 ]
```

Dies ist der im Zuge dieser Arbeit veränderte Operm5-Code (in C++) aus *dieharder*, Version 3.31.0. Die Variable `pseudoInv` bezeichnet die Pseudoinverse der neu berechneten Kovarianzmatrix.

```
/*
 * ================================================================
 * See copyright in copyright.h and the accompanying file COPYING
 * ================================================================
 */

/*
 *========================================================================
 * This is the Diehard OPERM5 test, rewritten from the description
 * in tests.txt on George Marsaglia's diehard site.
 *
 *          THE OVERLAPPING 5-PERMUTATION TEST                  ::
 * This is the OPERM5 test.  It looks at a sequence of one mill- ::
 * ion 32-bit random integers.  Each set of five consecutive     ::
 * integers can be in one of 120 states, for the 5! possible or- ::
 * derings of five numbers.  Thus the 5th, 6th, 7th,...numbers   ::
 * each provide a state. As many thousands of state transitions  ::
 * are observed,  cumulative counts are made of the number of    ::
 * occurences of each state.  Then the quadratic form in the     ::
 * weak inverse of the 120x120 covariance matrix yields a test   ::
 * equivalent to the likelihood ratio test that the 120 cell     ::
 * counts came from the specified (asymptotically) normal dis-   ::
 * tribution with the specified 120x120 covariance matrix (with  ::
 * rank 99).  This version uses 1,000,000 integers, twice.       ::
 *
 * Note -- the original diehard test almost certainly had errors,
 * as did the documentation.  For example, the actual rank is
 * 5!-4!=96, not 99.  The original dieharder version validated
 * against the c port of dieharder to give the same answers from
 * the same data, but failed gold-standard generators such as AES
 * or the XOR supergenerator with AES and several other top rank
 * generators.  Frustration with trying to fix the test with very
 * little useful documentation caused me to eventually write
 * the rgb permutations test, which uses non-overlapping samples
 * (and hence avoids the covariance problem altogether) and can
 * be used for permutations of other than 5 integers.  I was able
 * to compute the covariance matrix for the problem, but was unable
 * break it down into the combination of R, S and map that Marsaglia
 * used, and I wanted to (if possible) use the GSL permutations
 * routines to count/index the permutations, which yield a different
 * permutation index from Marsaglia's (adding to the problem).
 *
 * Fortunately, Stephen Moenkehues (moenkehues@googlemail.com) was
 * bored and listless and annoyed all at the same time while using
 * dieharder to test his SWIFFTX rng, a SHA3-candidate and fixed
 * diehard_operm5.  His fix avoids the R, S and map -- he too went
 * the route of directly computing the correlation matrix but he
 * figured out how to transform the correlation matrix plus the
 * counts from a run directly into the desired statistic (a thing
 * that frustrated me in my own previous attempts) and now it works!
 * He even made it work (correctly) in overlapping and non-overlapping
 * versions, so one can invoke dieharder with the -L 1 option and run
 * what should be the moral equivalent of the rgb permutation test at
 * -n 5!
 *
 * So >>thank you<< Stephen!  Thank you Open Source development
 * process!  Thank you Ifni, Goddess of Luck and Numbers!  And anybody
 * who wants to tackle the remaining diehard "problem" tests, (sums in
 * particular) should feel free to play through...
 *========================================================================
 */

#include <dieharder/libdieharder.h>

static int tflag=0;
static double tcount[120];

/*
* kperm computes the permutation number of a vector of five integers
* passed to it.
*/
int kperm(uint v[],uint voffset)
{
 int i,j,k,max;
 int w[5];
 int pindex,uret,tmp;

 /*
  * work on a copy of v, not v itself in case we are using
  * overlapping 5-patterns.
  */
 for(i=0;i<5;i++){
```

```
  j = (i+voffset)%5;
  w[i] = v[j];
 }

 if(verbose == -1){
  printf("=========================================================\n");
  printf("%10u %10u %10u %10u %10u\n",w[0],w[1],w[2],w[3],w[4]);
  printf(" Permutations = \n");
 }

 pindex = 0;
 for(i=4;i>0;i--){
  max = w[0];
  k = 0;
  for(j=1;j<=i;j++){
   if(max <= w[j]){
    max = w[j];
    k = j;
   }
  }
  pindex = (i+1)*pindex + k;
  tmp = w[i];
  w[i] = w[k];
  w[k] = tmp;
  if(verbose == -1){
   printf("%10u %10u %10u %10u %10u\n",w[0],w[1],w[2],w[3],w[4]);
  }
 }

 uret = pindex;

 if(verbose == -1){
  printf(" => %u\n",pindex);
 }

 return uret;
}

int diehard_operm5(Test **test, int irun)
{

 int i,j,k,kp,t,vind;
 uint v[5];
 double count[120];
 double av,norm,x[120],chisq,ndof,pvalue;

 /*
  * Zero count vector, was t(120) in diehard.f90.
  */
 for(i=0;i<120;i++) {
  count[i] = 0.0;
  if(tflag == 0){
   tcount[i] = 0.0;
   tflag = 1;
  }
 }

 if(overlap){
  for(i=0;i<5;i++){
   v[i] = gsl_rng_get(rng);
  }
  vind = 0;
 } else {
  for(i=0;i<5;i++){
   v[i] = gsl_rng_get(rng);
  }
 }

 for(t=0;t<test[0]->tsamples;t++){

  /*
   * OK, now we are ready to generate a list of permutation indices.
   * Basically, we take a vector of 5 integers and transform it into a
   * number with the kperm function.  We will use the overlap flag to
   * determine whether or not to refill the entire v vector or just
   * rotate bytes.
   */
  if(overlap){
   kp = kperm(v,vind);
   count[kp] += 1;
   v[vind] = gsl_rng_get(rng);
   vind = (vind+1)%5;
  } else {
   for(i=0;i<5;i++){
    v[i] = gsl_rng_get(rng);
   }
```

```
    kp = kperm(v,0);
    count[kp] += 1;
  }
}

for(i=0;i<120;i++){
  tcount[i] += count[i];
}

chisq = 0.0;
av = test[0]->tsamples/120.0;
norm = test[0]->tsamples; // this belongs to the pseudoinverse
/*
 * The pseudoinverse P of the covariancematrix C is computed for n = 1.
 * If n = 100 the new covariancematrix is C_100 = 100*C. Therefore the
 * new pseudoinverse is P_100 = (1/100)*P.  You can see this from the
 * equation C*P*C = C
 */

if(overlap==0){
  norm = av;
}
for(i=0;i<120;i++){
  x[i] = count[i] - av;
}

if(overlap){
  for(i=0;i<120;i++){
    for(j=0;j<120;j++){
      chisq = chisq + x[i] * pseudoInv[i][j]*x[j];
    }
  }
}

if(overlap==0){
  for(i=0;i<120;i++){
    chisq = chisq + x[i]*x[i];
  }
}

if(verbose == -2){
  printf("norm = %10.2f, av = %10.2f",norm,av);
  for(i=0;i<120;i++){
    printf("count[%u] = %4.0f; x[%u] = %3.2f ",i,count[i],i,x[i]);
    if((i%2)==0){printf("\n");}
  }
  if((chisq/norm) >= 0){
    printf("\n\nchisq/norm: %10.5f :-) and chisq: %10.5f\n",(chisq/norm)
        , chisq);
  }
}

if((chisq/norm) < 0){
  printf("\n\nCHISQ NEG.! chisq/norm: %10.5f and chisq: %10.5f",(chisq/
      norm), chisq);
}

chisq = fabs(chisq / norm);

ndof = 96; /* the rank of the covariancematrix and the pseudoinverse */
if(overlap == 0){
  ndof = 120-1;
}

MYDEBUG(D_DIEHARD_OPERM5){
  printf("# diehard_operm5(): chisq[%u] = %10.5f\n",kspi,chisq);
}

test[0]->pvalues[irun] = gsl_sf_gamma_inc_Q((double)(ndof)/2.0,chisq
    /2.0);

MYDEBUG(D_DIEHARD_OPERM5){
  printf("# diehard_operm5(): test[0]->pvalues[%u] = %10.5f\n",irun,test
      [0]->pvalues[irun]);
}

kspi++;

return(0);
}
```