Dokumentation zur IHK-Abschlussprüfung Sommer 2020

Entwicklung eines Softwaresystems

Geschrieben von: Felix Warschewski

Prüflingsnummer 101 20580

Tag der Einreichung: 26.06.2020

Programmiersprache: C#

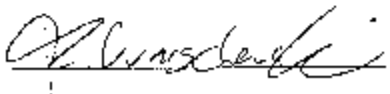
Inhaltsverzeichnis

1 Eigenhändigkeitserklärung

Eigenhändigkeitserklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Name: **Felix Warschewski**



Eschweiler, Freitag den 26.06.2020

2 Änderungen zu Tag 1

Sämtliche Variablen und Eigenschaften mit dem Datentyp long wurden in double umgeändert. Dies betreffen vor allem die Einstellparameter, sowie Rechnungen und hat den Vorteil, dass sämtliche mathematische Funktionen nicht umgewandelt werden müssen.

Zur Speicherung der Beispiel-LCGs wird eine Konstanten-Klasse verwendet. In dieser werden die Beispielparameter für die LCGs als statische Eigenschaften festgelegt.

Darüber hinaus wurde anstatt einer einfachen Verteilungsklasse eine Verteilungsschnittstelle implementiert. An diese Schnittstelle wurden zwei Unterklassen angebunden (Gleichverteilung, Standardnormalverteilung). Diese Klassen implementieren die Funktion ‚transformiere‘, welche eine gleichverteilte Zufallsvariable in eine nach der entsprechenden Verteilung transformierte Zufallsvariable umwandelt. Im Falle der Standardnormalverteilung geschieht dies bspw. über die Polarmethode.

Die Polar-Klasse wird gestrichen und die Funktionalität der Polar-Methode wird in der Funktion „transformiere(x)“ der Standardnormalverteilungsklasse implementiert. Dies hat den Vorteil, dass die Funktion direkt an der richtigen Stelle ist und nicht erst noch über eine Hilfsklasse aufgerufen werden muss.

3 Einleitung

Die erstellte Arbeit handelt von der Implementierung von Zufallszahlengeneratoren und das Testen der Güte mit den passenden Testverfahren. Es wurde eine Klassenbibliothek erstellt mit der man Zufallszahlen generieren kann und einen Zufallszahlengenerator, je nach eingestellter Verteilung, auf seine Güte mit einem entsprechenden Güte-Testverfahren prüfen kann. Des Weiteren wurde eine Konsolenanwendung implementiert, welche eine Eingabe-Datei mit Testparameter einliest und dynamisch nach den Eingaben die Berechnungen durchführt. Genauere Informationen befinden sich in der Benutzeranleitung.

4 Benutzeranleitung zur Ausführung

4.1 Laufzeitumgebung

Das verwendete Betriebssystem ist Windows 10 64-bit Version. Die Programmiersprache ist C# (Version C# 7.3) und es wurde in der Entwicklungsumgebung Visual Studio 2017 Enterprise Edition geschrieben. Das Zielframework der Klassenbibliothek ist .NET Standard 2.0 und das Zielframework der Konsolenapplikation ist das .NET Framework 4.7. Ursprünglich sollte die Konsolenanwendung in .NET Core programmiert werden, da dieses Framework auf einer breiteren Anzahl an Geräten läuft. Dies wurde aufgrund der Tatsache verworfen, dass der Compiler für die Core-Anwendung keine direkt ausführbare Datei generiert, sondern nur eine dll, die separat installiert werden muss.

4.2 Installation und Aufrufen

Die Installation erfolgt über das Extrahieren der abgegebenen zip-Datei. In dieser zip-Datei wird es eine Ordnerstruktur geben. In dem bin-Ordner befindet sich die .exe einer Konsolenanwendung, eine batch-Datei, sowie zwei Ordner „Tests“ und „Ergebnisse“. Es wird empfohlen das Programm über die Batch-Datei zu starten. Hierbei werden die Testfälle, welche in den Interpretationen aufgegriffen werden, neu berechnet und in den Ordner „Ergebnisse“ abgelegt.

Auf der ersten Ebene gibt es einen zweiten Ordner mit dem Namen „Programmcode“. In diesem Ordner befindet sich die Projektmappe zu der Klassenbibliothek und der Konsolenanwendung.

Darüber hinaus sind in dem zip-Ordner eine Dokumentation und eine Entwicklerdokumentation vorhanden.

4.3 Dateiformat

Mithilfe des args-Parameter in der Main-Methode kann ein Dateipfad übergeben werden. Dieser Pfad sollte auf eine Textdatei zeigen, in der Daten zur Bedienung der Konsolenanwendung stehen. Die batch-Datei kann ohne jegliche Vorkonfiguration gestartet werden.

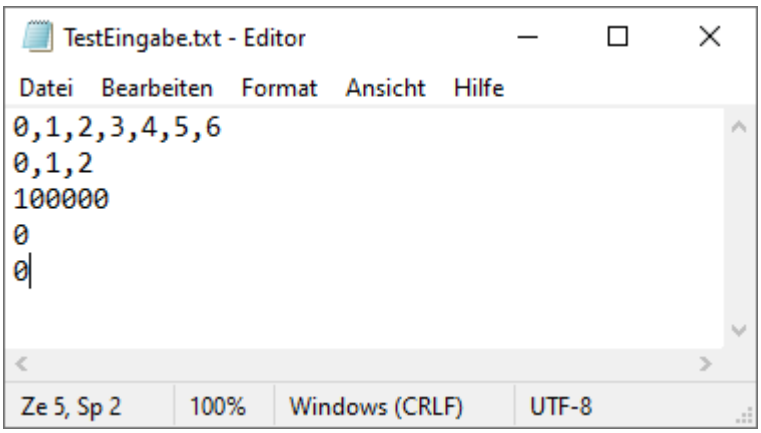


Abbildung 1, Eingabe-Datei

Die erste Zeile beschreibt die Zufallszahlengeneratoren, welche überprüft werden sollen. Falls mehrere Generatoren miteinander verglichen werden sollen, können diese hintereinander, mit einem Komma getrennt, aufgelistet werden. Das gleiche gilt für die Angabe der Testverfahren. In Kapitel 3.5 gibt es eine Auflistung an Möglichkeiten für die Eingabe. Die zweite Zeile beschreibt die Güte-Testverfahren, die angewendet werden sollen. Die dritte Zeile legt fest, wie hoch die Sequenzlänge ist, also wie viele Zufallszahlen für die Prüfungen erstellt werden. Die vierte Zeile legt ein k fest. Bei der seriellen Autokorrelation ist das k die Ordnung der Autokorrelation und steht für den Abstand der verglichenen Paare. Bei dem Sequenz-Up-Down-Test bestimmt das k die Länge einer bestimmten Bitfolge. Gibt man 0 für k an, erstellt die serielle Autokorrelation einen zufälligen Abstand k und bei dem Sequenztest wird über alle k iteriert und letztlich die Differenz ausgegeben. Bei dem Sequenz-Up-Down-Test ist k die Kettenlänge, welche verglichen werden soll. Die letzte Zeile gibt die Art der Verteilung der Zufallszahlen an. Man kann unterscheiden zwischen standardnormalverteilten Zufallszahlen und gleichverteilten Zufallszahlen. Hierbei sind gleichverteilte Zufallszahlen der Standard.

4.4 Aufrufbefehle

Innerhalb der aufzurufenden Textdatei:

Zeile 1 – Generatoren:

1. Ansi-C
2. Minimal Standard
3. RANDU
4. SIMSCRIPT
5. NAG’s LCG
6. Maple’s LCG
7. Datumsbasiert(eigener)

Zeile 2 – Güte-Testverfahren

0. Serielle Autokorrelation
1. Sequenz-Up-Down-Test
2. Eigener Güte-Test

Zeile 3 – Sequenzlänge

Zeile 4 – k

> Wenn k = 0 dann werden die allgemeinen Bedingungen übernommen. Siehe 6.4 Dateiformat.
> Wenn k <> 0 dann wird die k-te Ordnung des Verfahrens berechnet und zurückgegeben.

Zeile 5 – Verteilung

0. Gleichverteilung
1. Standardnormalverteilung

5 Beschreibung der Klassenbibliothek

Der Aufbau der Klassenbibliothek und der zugehörigen Konsolenanwendung kann der Abbildung 2, Klassendiagram entnommen werden. Die drei Hauptbestandteile der Klassenbibliothek sind die Schnittstellen. Es gibt eine Schnittstelle für die Zufallszahlengeneratoren (Zufallsbibliothek), eine Schnittstelle für die Güte-Testverfahren (GüteTests) und eine für die Verteilungen (Verteilung).

Die beiden Zufallszahlengeneratoren LCG und Datumsbasiert implementieren die Zufallsbibliothek-Schnittstelle. Diese beiden Generatoren haben nun die Funktionalität Zufallszahlen zu generieren. Darüber hinaus hängt die Generierung von Zufallszahlen auch von der verwendeten Verteilung ab. Je nach Verteilung werden anders verteilte Zufallszahlen ausgegeben.

Mögliche Verteilungen sind die Gleichverteilung und die Standardnormalverteilung.

Mögliche Güte-Testverfahren sind die serielle Autokorrelation, der Sequenz-Up-Down-Test und das eigene Güte-Testverfahren.

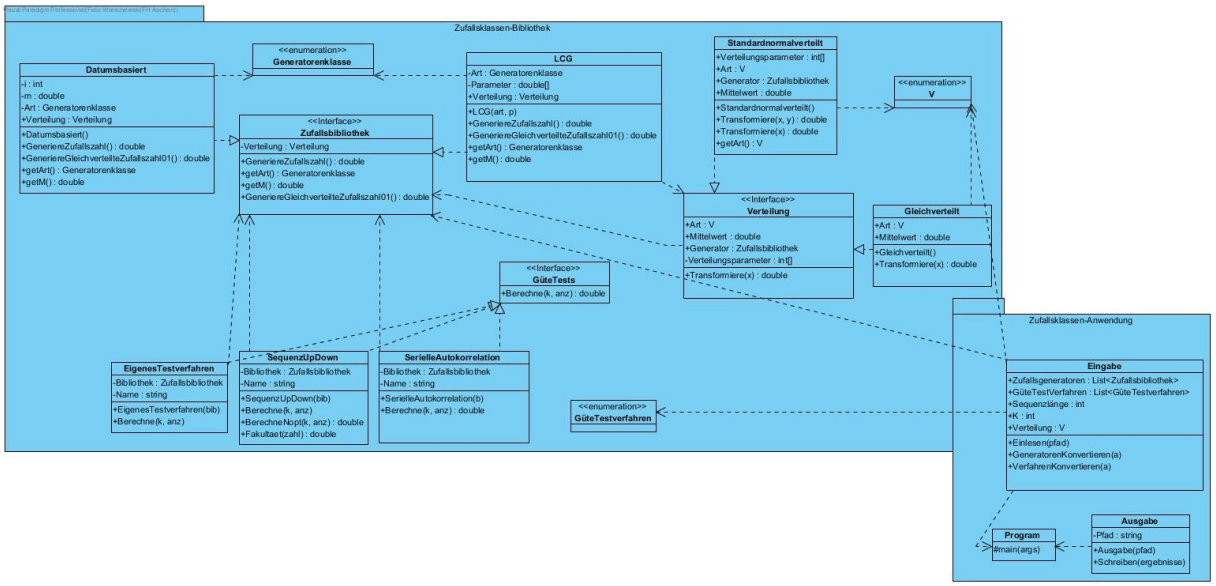


Abbildung 2, Klassendiagramm

Der Programmablauf kann in dem Sequenzdiagramm in

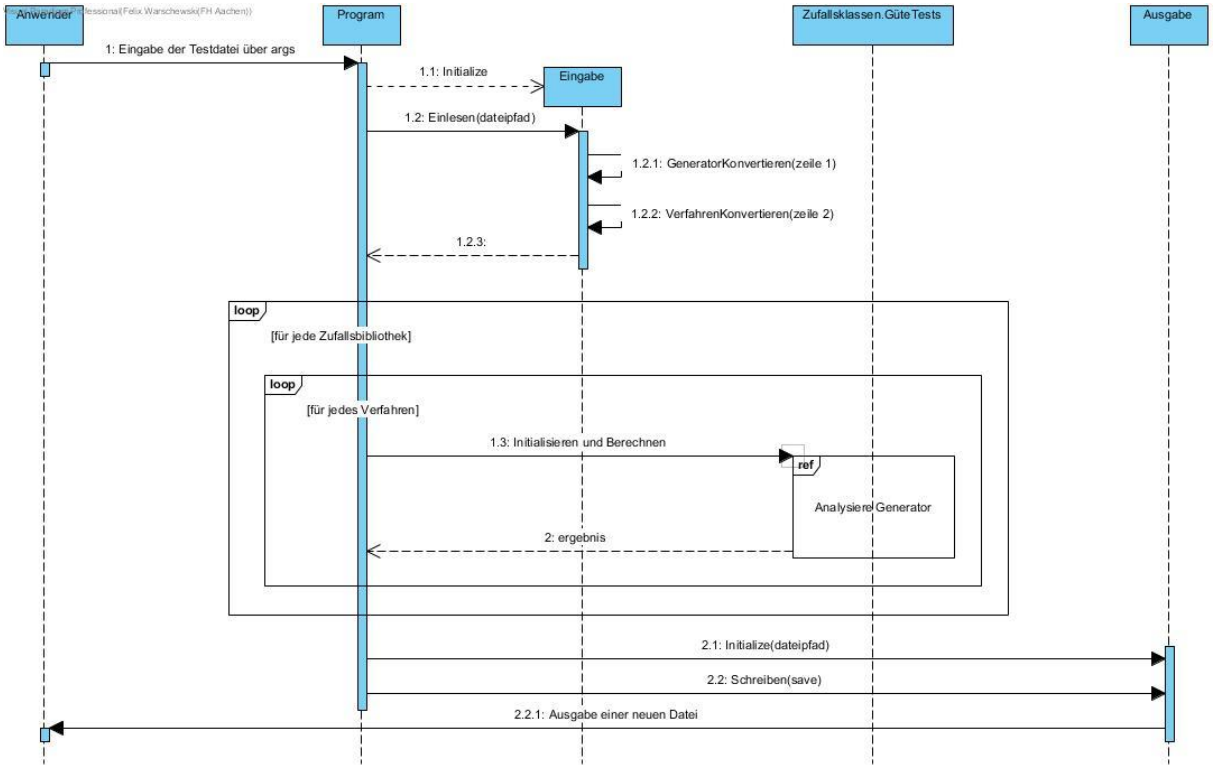


Abbildung 3, Testdurchlauf nachempfunden werden. Es wird die Konsolenanwendung mit einem Dateipfad aufgerufen. Dann wird die Textdatei unter dem mitgegebenen Dateipfad eingelesen. Je nachdem, wie viele Generatoren und Verfahren erzeugt werden sollen, iteriert das Programm durch die Verfahren. Am Ende werden die Ergebnisse in eine Ausgabedatei geschrieben. Der genauere Ablauf der Programmlogik wird in Abbildung 4, Analyse Generator dargestellt.

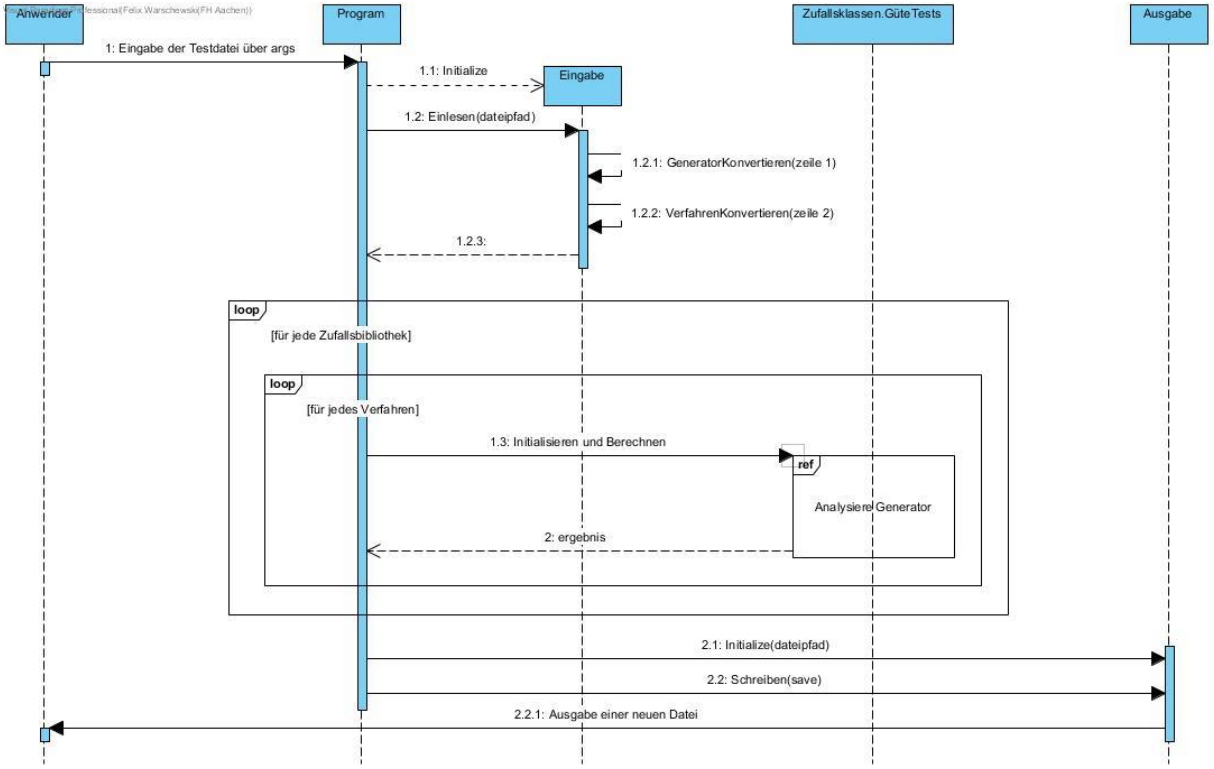


Abbildung 3, Testdurchlauf

In Abbildung 4, Analyse Generator ist die Initialisierung eines Zufallsgenerators dargestellt, (Im Detail Abbildung 5, Initialisiere Generator). Im Anschluss wird der Generator dem jeweiligen Testverfahren übergeben. Dann werden je nach Güte Test verschiedene Operationen ausgeführt. Diese kann man in der **Fehler! Verweisquelle konnte nicht gefunden werden.**, **Fehler! Verweisquelle konnte nicht gefunden werden.**, **Fehler! Verweisquelle konnte nicht gefunden werden.**, **Fehler! Verweisquelle konnte nicht gefunden werden.** sehen.

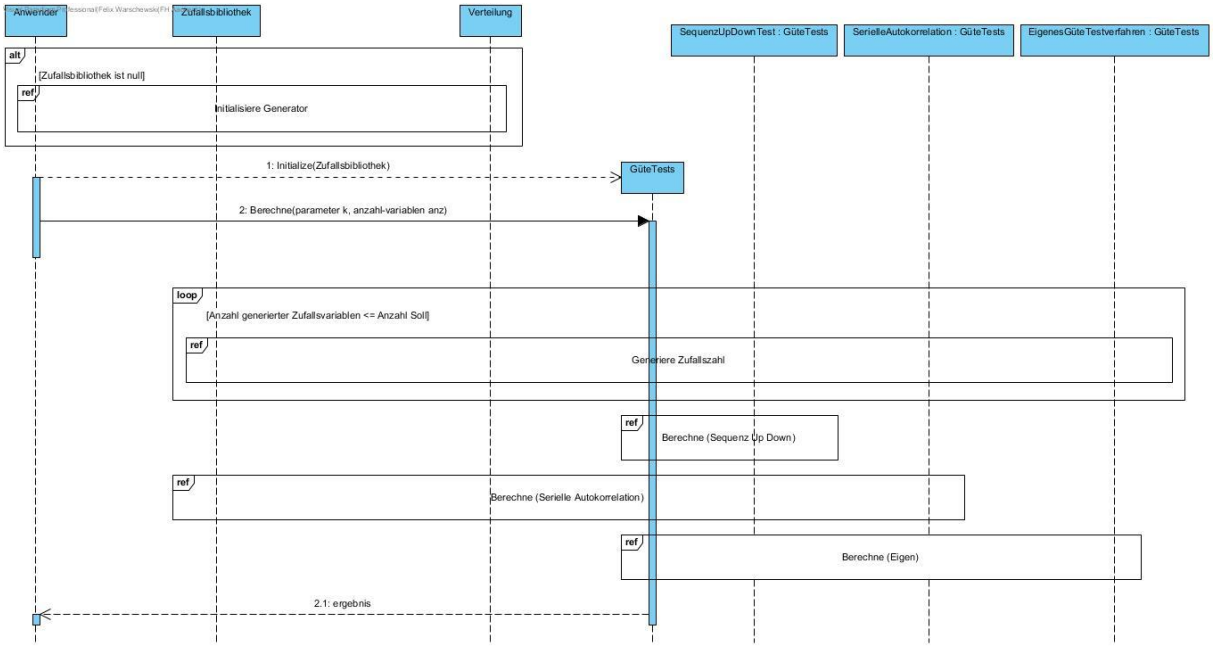


Abbildung 4, Analyse Generator

In der Abbildung 5, Initialisiere Generator steht beschrieben, dass die erstellte Instanz eines Zufallszahlengenerators seine Einstellparameter und ein Enum zugewiesen bekommt. Des Weiteren wird ihm eine Verteilung zugewiesen. Aufgrund dieser werden dann die Zufallszahlen generiert.

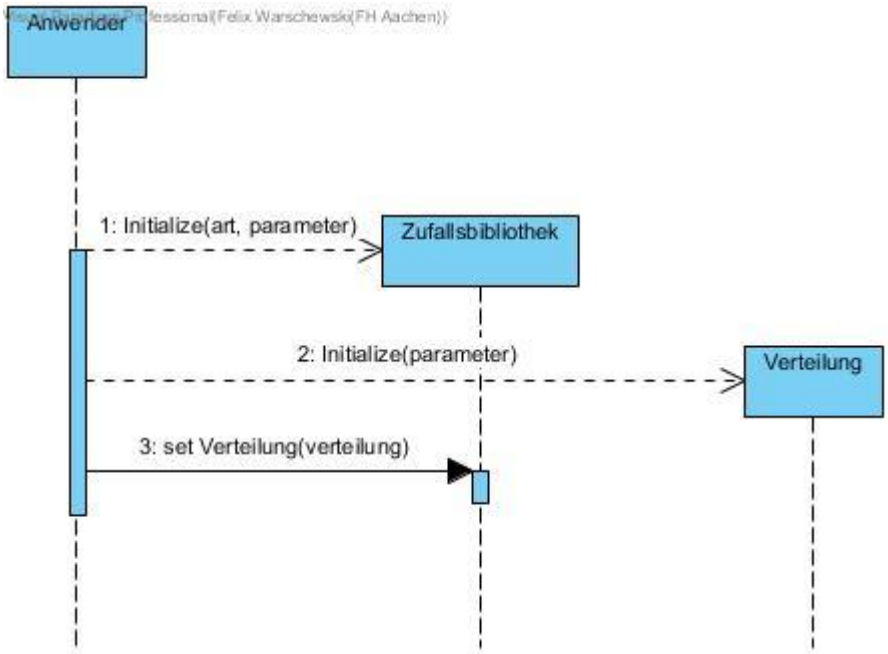


Abbildung 5, Initialisiere Generator

Es folgen die Hauptalgorithmen des Programmes in Form von Aktivitätsdiagrammen.

In der Abbildung 6, Generiere Zufallszahl (LCG) ist die Generierung einer Zufallszahl beschrieben. Es werden die Parameter gesetzt und mithilfe der LCG-Formel Zahlen in einem Zahlenraum von 0 bis m erzeugt. Dies ändert sich indem durch das Modul geteilt wird. Nun ergibt sich ein Zahlenraum von 0 bis 1. Je nach Verteilung werden die Zahlen erneut auf einen anderen Zahlenraum transformiert. Dies ist aber verteilungsabhängig. Letztlich wird eine Zufallszahl ausgegeben in Abhängigkeit der angegebenen Verteilung des Zufallszahlengenerators.

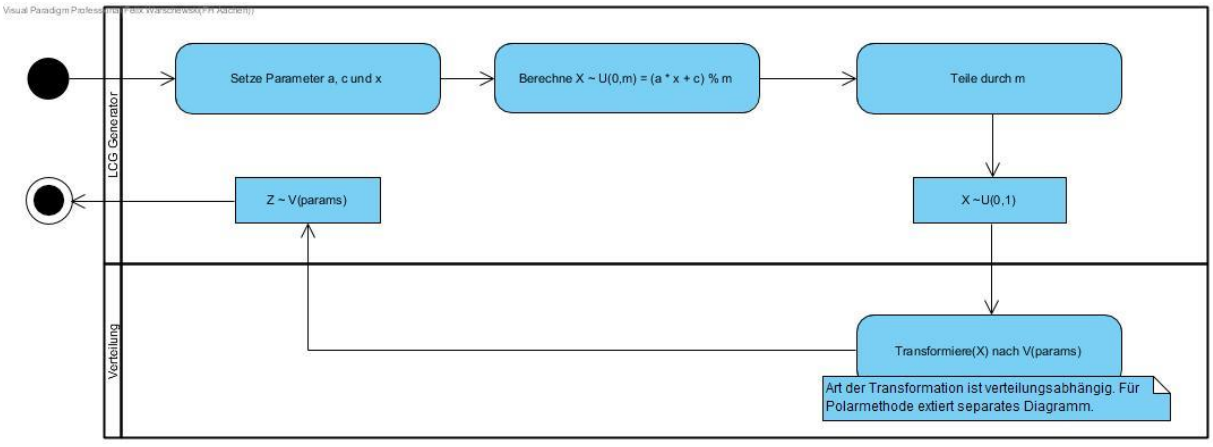


Abbildung 6, Generiere Zufallszahl (LCG)

In Abbildung 7, Generiere Zufallszahl Datumsbasiert ist die Logik des eigenen datumsbasierten Zufallszahlengenerators beschrieben. Es wird die aktuelle Zeit und Systemlaufzeit des Programmes aufsummiert und durch ein gegebenes Modul Modulo gerechnet und geteilt. Dies erzeugt eine Zufallszahl in einem Zahlenraum von 0 bis 1. Anschließend wird ggfls. transformiert, wenn nicht die Gleichverteilung als Verteilung angegeben ist, da alle generierten Zufallszahlen zu Beginn gleichverteilt sind.

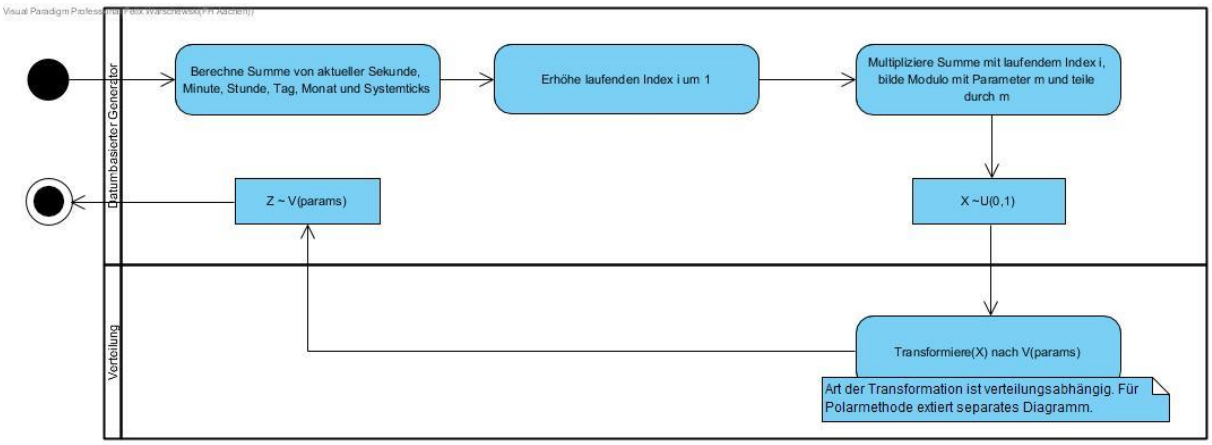


Abbildung 7, Generiere Zufallszahl Datumsbasiert

Die Abbildung 8, Polar-Methode (Normaltransformation) beschreibt die Polar-Methode, welche immer dann angewandt wird, wenn standardnormalverteilte Zufallszahlen generiert werden sollen. Hierfür wird ein gegebenes x verwendet und das auf den Zahlenraum -1 bis 1 transformiert. Anschließend wird ein y zufällig generiert und für die Polarmethode verwendet, falls x und y gemeinsam im Punkt im Einheitskreis liegen. Ansonsten wird solange ein y generiert bis diese Bedingung erfüllt ist. Durch die letzte Berechnung gibt die Methode eine Standardnormalverteilte Zufallszahl in dem Zahlenraum 0 bis 1 wieder.

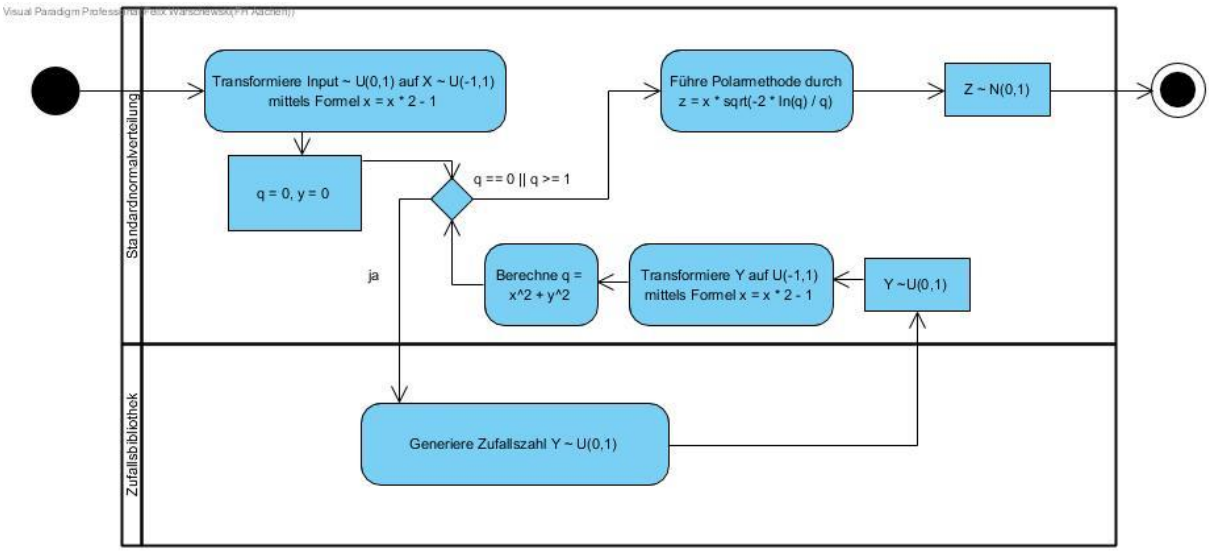


Abbildung 8, Polar-Methode (Normaltransformation)

In der Abbildung 9, Serielle Autokorrelation steht das Güte-Testverfahren der seriellen Autokorrelation beschrieben. Zuerst wird eine Testmenge mit einer bestimmten Größe erstellt. Anschließend wird die Abhängigkeit aller Zufallszahlenpaare mit einem gegebenen Abstand geprüft. Dies ergibt dann den Korrelationsfaktor, welcher zurückgegeben wird.

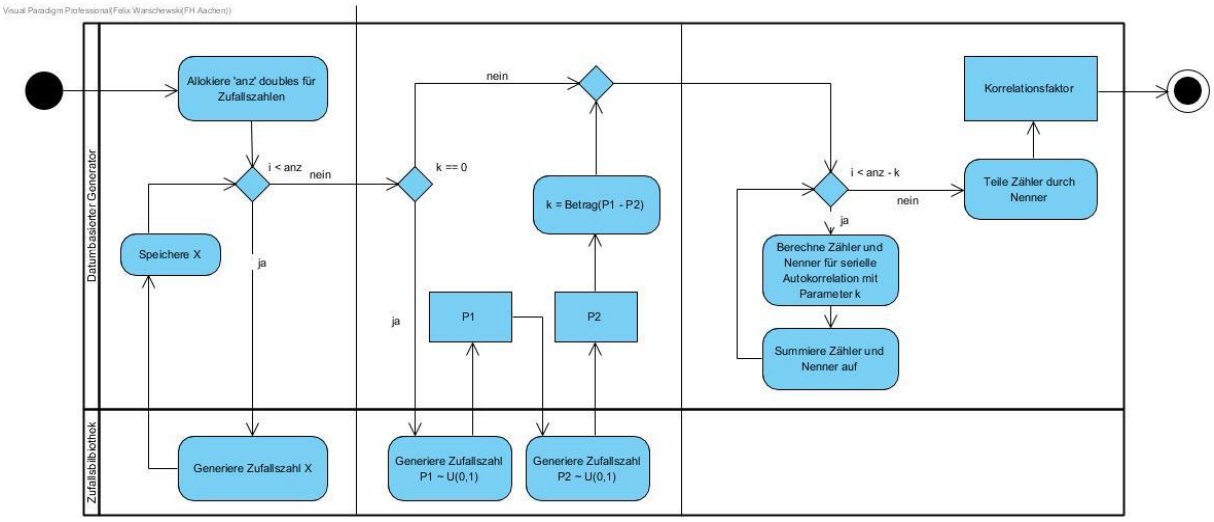


Abbildung 9, Serielle Autokorrelation

Die Abbildung 10, Sequenz-Up-Down-Test zeigt die Funktionsweise des Sequenz-Up-Down-Testverfahrens. Es wird eine Testmenge mit einer bestimmten Länge erzeugt und daraufhin eine Bit Maske erstellt. In der Bit Maske steht eine 1 dafür, dass das vorherige Element kleiner war als das jetzige und eine 0 für den anderen Fall. Anschließend wird zu jeder Kettenlänge die Häufigkeit ihrer Vorkommen in der Testmenge gespeichert. Falls ein k<>0 angegeben war, wird die Abweichung der Häufigkeit für die Kettenlänge mit Länge k mit der optimalen Häufigkeit zurückgegeben. Ansonsten wird die Gesamtdifferenz über alle Häufigkeiten aller Kettenlängen mit der optimalen Verteilung zurückgegeben.

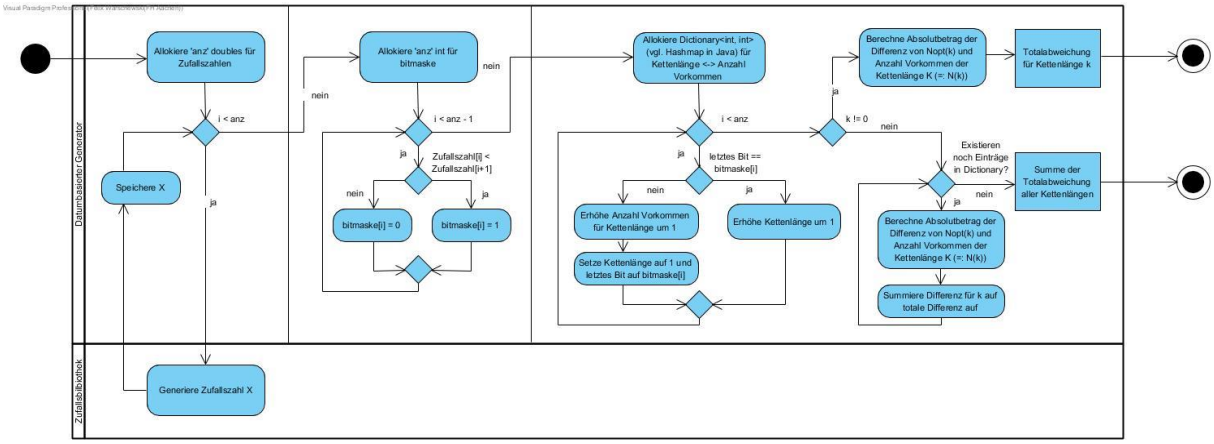


Abbildung 10, Sequenz-Up-Down-Test

In der Abbildung 11, Eigenes Güte-Testverfahren wird die Funktionalität des eigen entworfenen Güte-Testverfahren beschrieben. Dieses erstellt sich zuerst eine Testmenge und daraufhin eine Bit Maske exakt wie im vorherigen Güte-Testverfahren in Abbildung 10, Sequenz-Up-Down-Test. Daraufhin wird ein Mittelwert für alle Zustandswechsel in der Bit Maske generiert und zurückgegeben.

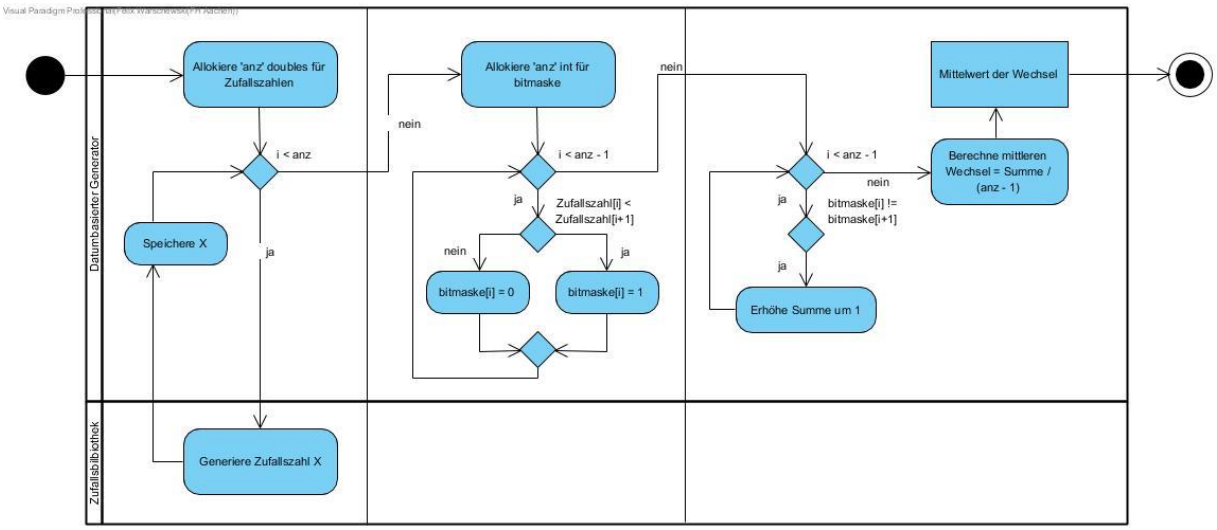


Abbildung 11, Eigenes Güte-Testverfahren

6 Beschreibung der Zufallszahlengeneratoren und Verfahren

Zu implementieren waren folgende Generatoren und Testverfahren:

- Linearer Kongruenz-Generator mit 6 verschiedenen Einstellungsmöglichkeiten
- Umwandlung gleichverteilter in normalverteilte Zufallszahlen mithilfe der Polar-Methode
- Eigener Zufallszahlengenerator (Datumsbasiert)
- Güte-Testverfahren Serielle Autokorrelation
- Güte-Testverfahren Sequenz-Up-Down-Test
- Eigenes Güte-Testverfahren

Der Lineare Kongruenz-Generator (LCG) errechnet Zufallszahlen auf Basis der Kongruenzrechnung. Hierfür sind 4 Eingabeparameter wichtig. Das Modul m, der Multiplikator a, das Inkrement c und der Startwert x0. Des Weiteren sind 6 zu implementierende Verfahren für den LCG in Tabellenform gegeben. Diese Verfahren unterscheiden sich in der Wahl ihrer Einstellungsparameter. Der LCG funktioniert auf Grundlage einer Folge, wobei die Elemente der Folge schnell sehr groß werden und willkürlich erscheinen.

Der LCG erzeugt hauptsächlich gleichverteilte Zufallszahlen. Daher sollte eine Methode zur Transformierung von gleichverteilte in normalverteilte Zufallszahlen implementiert werden. Die

Methode, welche im Programm angewendet wird, ist die Polar-Methode. Hierbei werden euklidische Koordinaten, die von den verschiedenen Zufallsverfahren generiert werden, in Polarkoordinaten umgewandelt. Wenn beide Koordinaten als Punkt im Einheitskreis liegen, werden sie zu zwei unabhängigen standardnormalverteilten Zufallszahlen transformiert. Dieses Prinzip der Verteilungstransformation kann auf jedes Verfahren des Linearen Kongruenz-Generators angewendet werden.

Der eigene Zufallszahlengenerator funktioniert auf Basis der Zeit und Laufzeit des Programms. Es werden die Sekunden, Minuten, Stunden, Tage, Monate als Integer aufsummiert, sowie die aktuelle Laufzeit des Programmes in Millisekunden. Darüber hinaus wird gespeichert, wie viele Zufallszahlen bereits mit einer Instanz des Generators erzeugt wurden. Die Summe wird mit der Anzahl der bereits erzeugten Zufallszahlen multipliziert. Das Ergebnis wird anschließend mit Modulo 1000 gerechnet und durch das Modul geteilt. So ergibt sich eine Zufallszahl zwischen 0 und 1 mit 3 Nachkommastellen. Die erhaltene Zufallszahl kann bis 1000 hochskaliert werden ohne ungenau zu werden. Will man höher skalieren benötigt man ein höheres Modul.

Das Güte-Testverfahren mithilfe der seriellen Autokorrelation misst die Abhängigkeit einer Folge von Zahlen. Die Zufallszahlen sollen paarweise möglichst unabhängig sein und im Optimalfall eine Korrelation von p=0 aufweisen. Das Verfahren verwendet eine Sequenz von Zufallszahlen, einen von der Verteilung abhängigen Mittelwert von 0,5 in der Gleichverteilung und 0 in der Normalverteilung, sowie die Ordnung k. Je nachdem wie hoch die Ordnung ist, prüft man auf weiter entfernte oder näher aneinander liegende Paare in der Sequenz. Die Wahl von k kann das Ergebnis des Güte-Tests stark beeinflussen. Daher sollte man zum Testen von bestimmten Generatoren die Verfahren mehrmals mit unterschiedlichen Ordnungen durchlaufen lassen. Wenn man in einer Textdatei kein k angibt (k=0), wird dieses zufällig generiert. Dieser Vorgang ist nur für das gegebene Programm wichtig.

Ein weiteres Güte-Testverfahren ist der Sequenz-Up-Down-Test. Dieser prüft wie viele Folgen bestimmter Länge von stetig größer oder kleiner werdenden Zahlen in einer Zufallszahlensequenz vorhanden sind. Es wird eine Bit Maske erstellt mit den Werten 0 und 1. Der Wert 0 wird in die Maske eingefügt sobald ein folgender Wert kleiner ist als der jetzige. Wenn der folgende Wert hingegen größer als der derzeitige ist, wird eine 1 in die Bit Maske eingefügt. Das Ergebnis des Verfahrens gibt einen Aufschluss darüber, wie viele Folgen es mit einer bestimmten Länge gab. Je höher die Zahl bei niedrigen Werten für k (Kettenlänge) ist, desto besser funktioniert der Zufallszahlengenerator, da die Werte öfter hin und her springen und sich nicht stetig aufbauen.

Das eigene Güte-Testverfahren erstellt auch eine Bit Maske. Hier werden alle Werte gleich dem Sequenz-Up-Down-Verfahren in die Bit Maske geschrieben. Anschließend wird die Bit Maske aufsummiert und durch die Anzahl der vorhandenen Bits der Bit Maske dividiert. Dies gibt Aufschluss über den Mittelwert der Wechsel der Zahlenfolgen. Wenn der Mittelwert deutlich unter 0,5 liegt, ist das ein Zeichen für sehr wenige Wechsel und die Entstehung längeren Folgen. Das bedeutet die Güte des Generators wäre schlechter, als wenn der Mittelwert über 0.5 liegt. Je höher desto öfter wechselt er innerhalb der Zufallszahlensequenz eine Reihenfolge und desto zufälliger werden die Zahlen generiert.

7 Diskussion der Güte-Testverfahren und der Parameter für den LCG

Es werden sechs verschiedene Verfahren für den Linearen Kongruenz-Generator vorgestellt. Diese unterscheiden sich teils stark und haben deswegen auch eine andere Abfolge zur Erzeugung von Zufallszahlen.

Der Ansi-C LCG besitzt vergleichsweise große Parameter zu dem Rest der Verfahren. Schon im zweiten Schritt wird das Modul häufig übersprungen. Da das Verfahren so große Parameter besitzt, bekommt es für längere Zahlenfolgen auch immer gleichere Muster heraus und wird bei einer hohen Sequenzlänge (z.B. 10.000) schlechter abschneiden als Verfahren mit kleineren Multiplikatoren. Das Ansi-C Verfahren weist auch ein Muster von geraden und ungeraden Zufallszahlen im dauerhaften Wechsel auf. Dies liegt daran, dass das Modul eine gerade Zahl ist und der Multiplikator ungerade. Durch das ungerade Inkrement springen die Zufallszahlen immer von geraden Zahlen zu ungeraden Zahlen und umgekehrt.

Für das Minimal Standard-Verfahren wird ein ungeraderes Modul verwendet und ein deutlich kleinerer Multiplikator als bei den meisten anderen Verfahren. Der kleinere Multiplikator sorgt dafür, dass die Zahlen in den ersten Schritten nicht so häufig über das Modul herauskommen. Nichtsdestotrotz sollte durch die exponentiell ansteigende Zahl, vor Abzug des Modulo, das Modul schnell erreicht werden.

Das RANDU -Verfahren fängt mit dem Startwert 1 an und wird mit einem ungeraden Multiplikator multipliziert. Da das Modul gerade ist, bekommen wir nur ungerade Zahlen heraus. Das bedeutet die mögliche Zahlenmenge des RANDU -Verfahren ist halbiert. Des Weiteren weisen die ersten zehn generierten Zufallszahlen eine steigende Abfolge auf, obwohl das Modulo mehrmals überschritten wird. Dies sollte sehr schlechte Güte-Ergebnisse für kleine Sequenzlängen bis 10 zu Folge haben.

SIMSCRIPT besitzt als einziges Verfahren einen geraden Multiplikator und ein ungerades Modul. Das Verfahren ist von der Höhe der Parameter im Vergleich zu den anderen am meisten ausgeglichen. Daher kann die Vermutung aufgestellt werden, dass die generierten Zufallszahlen über alle Sequenzlängen am besten verteilt ist.

NAG‘s LCG generiert ausschließlich ungerade Zahlen, wobei der zur Verfügung gestellte Zahlenraum mit 2 hoch 59 deutlich über dem der anderen Verfahren liegt. Das gibt eine breitere Streuung der generierten Zufallszahlen, wenn man von der Differenz zu einem Mittelwert ausgeht. Dies könnte ein Nachteil für das Güte-Testverfahren der seriellen Autokorrelation darstellen. Des Weiteren ist der Startwert relativ hoch und das Verfahren hat früh eine gewisse Länge der Zufallszahlen erreicht. Das bedeutet das Verfahren funktioniert für eine geringe Sequenzlänge schon willkürlich, im Gegensatz zum RANDU-Verfahren.

Bei dem Maple’s LCG wird als Modul ein in Relation niedriges Modul gewählt in Kombination mit einem sehr hohen Multiplikator. Dies führt dazu, dass das Modul oft übertroffen wird. Da der Multiplikator beinahe so groß wie das Modul ist, werden vor allem Zahlen generiert, welche das Modul beinahe annähern. Daher haben diese auch im Verhältnis zueinander keine allzu großen Abweichungen und sind relativ konstant. Das kann zur Folge haben, dass das Verfahren eine relativ konstant gute Güte über diverse Sequenzlängen aufzeigt.

Um zwei Beispielverfahren aus der Menge der sechs gegebenen Verfahren zu wählen, verwende ich den Ansi-C LCG und den RANDU LCG. Beide haben ein Modul von 2^{31}. Des Weiteren haben beide Verfahren für das Modul einen geraden Wert und für den Multiplikator einen ungeraden Wert. Ohne das ungerade Inkrement des Ansi-C Verfahrens würde dieser auch ausschließlich ungerade Zahlen generieren. Durch das Inkrement werden aber gerade und ungerade Zufallszahlen im Wechsel generiert. Der Ansi-C Generator hat einen deutlich höheren Multiplikator von 1.103.515.245 im Vergleich zu dem RANDU-Multiplikator von 65.539. Das Inkrement und der Startwert bei dem Ansi-C

ist auch höher. Allein dies gibt eine Aussage darüber, wie schnell der Algorithmus den Wert des Moduls erreicht und übertrifft. Wenn er den Wert des Moduls übertrifft, fängt dieser wieder bei 0 an und zählt nach oben. Ansi-C erreicht das Modul deutlich schneller und besitzt einen Wechsel von größeren und kleineren Zahlen. Der RANDU LCG hingegen braucht länger und besitzt mehr Folgen von aufeinander größer werdenden Zufallszahlen. In den folgenden Testfällen muss geprüft werden, ob dies eine Auswirkung auf die Güte des Generators hat. Sobald aber der Parameter x eine gewisse Größe erreicht hat, werden die Sprünge immer größer und das Ergebnis überschreitet öfter das Modul. Dies bedeutet, dass die Werte öfter variieren und die Folgen von aufeinanderfolgenden größer werdenden Zufallszahlen deutlich kleiner werden. Daher sollte man die beiden Zufallszahlengeneratoren mit mehreren beliebig großen Sequenzlängen testen.

8 Diskussion der Testbeispiele

Die Testbeispiele geben Aufschluss über die Güte der Zufallszahlengeneratoren. Deswegen muss hier differenziert werden und es kann nicht nur ein Testverfahren mit einer bestimmten Sequenzlänge und einer bestimmten Ordnung verwendet werden. Es müssen mehrere unterschiedliche Testvorgänge durchgeführt werden.

Aus dem Kapitel 7 kann angenommen werden, dass sich die Güte der Testverfahren je nach Sequenzlänge ändern kann. Genauso können Rückschlüsse über einen Zufallszahlengenerator gewonnen werden, indem man die Abstände der verglichenen Zufallszahlen beziehungsweise die Kettenlängen der überprüften Folgen ändert. Die Änderungen kann man anhand der Variablen „k“ bestimmen, welche in der Eingabe-Datei gesetzt werden kann.

Zum Vergleich werden folgende Testfälle betrachtet. Alle Testfälle können mit ihren Werten in dem Auslieferungsordner unter „TestsZurDoku“ gefunden werden.

Fall 1:

Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 10

Fall 2:

Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 1000

Fall 3:

Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 100000

Fall 4:

Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 10

Fall 5:

Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 1000

Fall 6:

Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 100000

Fall 7:

Alle LCG-Verfahren mit Sequenzlänge 100000 und einem k von 1

Fall 8:

Alle LCG-Verfahren mit Sequenzlänge 100000 und einem k von 10000

Fall 9:

Alle LCG-Verfahren mit beiden vorgegebenen Testverfahren und einer Länge von 100000 und Standardnormalverteilung

Fall 10:

Eigener Datumsbasierter Zufallszahlengenerator für eine Sequenzlänge von 1000

Fall 11:

Alle LCG-Verfahren auf den eigenen Güte-Test für eine Sequenzlänge von 100000

9 Vergleich und Interpretation der Ergebnisse

Um die Ergebnisse der Testfälle besser einordnen zu können, folgt eine Beschreibung ihrer Aussagekraft. Das Güte-Testverfahren der seriellen Autokorrelation fällt besser aus je näher das Ergebnis an null liegt. Wenn das Ergebnis sich signifikant von null unterscheidet, wird der Zufallsgenerator abgelehnt. In dem Fall ist dieser nicht brauchbar, da er Muster aufzeigt und voneinander abhängige Zufallszahlen bildet. Eine signifikante Unterscheidung von 0 wäre bei der seriellen Autokorrelation schon ein Wert größer 0,2 oder kleiner -0,2.

Der Sequenz-Up-Down-Test gibt einen Wert für die Gesamtdifferenz zu dem optimalen Wert aller Kettenlängen zurück. So kann gesagt werden, wie stark ein Verfahren von der optimalen Verteilung abweicht. Wenn der Wert deutlich zu den anderen gewonnen Werten ausschlägt, ist die Verteilung der Folgen von verschiedenen Kettenlängen besonders und nicht optimal. Es gibt dann Ausreißer für verschiedene Kettenlängen und dies ist ein Zeichen dafür, dass Muster gebildet werden.

Der Vergleich der Testergebnisse, sowie der Vergleich folgt zuerst in Reihenfolge der Testfälle. Es wird auf jeden Testfall eingegangen und direkte Zusammenhänge interpretiert. Die Zusammenfassung über alle Generatoren und Verfahren folgt im Kapitel 8 „Zusammenfassung und Ausblick“.

Fall 1: Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 10

Die Folge, die am schlechtesten abschneidet, ist das RANDU-Verfahren. Es wird ein Wert von 1 zurückgegeben. Dies ist der schlechtmöglichste Wert, der erreicht werden kann. Dies liegt daran, dass die Werte für RANDU in dem Intervall 1-10 stetig größer werden. Die generierten Zahlen sehen nicht aus wie Zufallszahlen, sondern wie eine immer größer werdende Folge. Das beste Ergebnis für eine derartig kleine Sequenzlänge liefert das Minimal Standard-Verfahren mit -0,2. Ein negatives Ergebnis kommt zustande, wenn mehr Zufallszahlen unter dem Stichprobenmittelwert von 0,5 liegen als darüber. Das Ergebnis ist das beste von allen sechs Generatoren. Dennoch ist der Wert keineswegs optimal. Dies liegt vor allem daran, dass alle Verfahren auf dem Linearen Kongruenz-Generator basieren und dieser wie eine Folge mit spezifischen Eingabeparametern funktioniert. Der LCG braucht eine bestimmte Menge an Durchläufen, bis er zuverlässige Zufallszahlen ausgibt. In den nächsten Testschritten wird geprüft, wie lange die verschiedenen Verfahren brauchen, um eine gute Güte zu erreichen.

Fall 2: Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 1000

Die Werte für die Güte der Verfahren hat sich deutlich gebessert. Der einzige Zufallszahlengenerator, welcher eine unbrauchbare Güte aufweist, ist der NAG’s LCG mit einer Güte von -0,42. Dieser Generator besitzt das größte Modul. Durch die extreme Breite des Zahlenraums, der mithilfe des gewählten Moduls zur Verfügung gestellt wird, treten auch stärkere Abweichungen von einem Mittelwert auf. In den folgenden Testfällen ist zu prüfen, ob sich mit einer größeren Sequenzmenge die Güte des Verfahrens verbessert.

Sehr gute Werte liefern die Generatoren Minimal Standard (0,0006) und Maple’s LCG (-0,0003). Das Maple’s Verfahren nähert, wie in Kapitel 6 beschrieben, das Modul an und liefert Zufallszahlen in einem recht konstanten Bereich. Dies reflektiert das starke Güte-Testergebnis mit der seriellen Autokorrelation bei einer Sequenzlänge von 1.000. Bei dem Minimal Standard-Verfahren wird genau das gegenteilige Prinzip angewandt. Hier ist der Multiplikator in Relation niedrig und das Modul größer als beim Maple’s LCG. Dies sorgt dafür, dass sich die Zahlen relativ schnell ausgleichen und nicht allzu weit voneinander entfernt liegen.

Fall 3: Alle LCG-Verfahren mit serieller Autokorrelation für eine Sequenzlänge von 100000

Für den dritten Testfall bleibt das NAG’s Verfahren das mit der schlechtesten Güte. Diese hat sich sogar mit zunehmender Sequenzlänge nochmal verschlechtert (-0,5). Dies wird vor allem durch den großen Zahlenraum, welcher durch das große Modul bereitgestellt wird, hervorgerufen. Die Werte können sich hier deutlicher unterscheiden als bei kleineren Zahlenräumen.

Die Werte für die Güte der anderen Generatoren haben sich nicht sonderlich verändert. Die besten Gütewerte haben die Verfahren RANDU (0,0018), SIMSCRIPT (-0,0013) und Maple’s (-0,0024).

Das Minimal Standard-Verfahren, welches im vorherigen Testfall die beste Güte aufzeigte, hat sich auf eine Güte von 0,01 verschlechtert. Dies kann bedeuten, dass der Minimal Standard bis 1.000 Zufallszahlen eine relativ geringe Abweichung aufweist und danach wieder mehr streut. Also wird es bei diesem Verfahren Schübe geben, in denen die Folge besser zufällig generiert werden bzw. schlechter.

Für das Güte-Testverfahren der seriellen Autokorrelation ist besonders der RANDU-Generator hervorzuheben. Bei einer sehr geringen Sequenzlänge hat dieser noch die schlechtmöglichste Güte und je größer die Zahlen werden, desto besser wird seine Güte. Darüber hinaus kann man über den NAG’s LCG sagen, dass der große Zahlenraum die Werte deutlich weiter streuen lässt und die Differenzen der einzelnen Zahlen zum Mittelwert sehr verschieden ausfallen können, was den Generator nach dem Prinzip der seriellen Autokorrelation schlecht macht.

Fall 4: Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 10

Bei dem Sequenz-Up-Down-Test schneidet der Maple’s Generator am schlechtesten ab (3,35). Es wird die Differenz für die Häufigkeit sämtlicher Kettenlängen mit der Häufigkeit im angenäherten Optimalfall gebildet. Wenn eine hohe Zahl als Güte ausgegeben wird, bedeutet dies, dass der Generator über den ganzen Verlauf der optimalen Häufigkeit verschiedener Kettenlängen relativ schlecht war. Das beste Ergebnis erzielte RANDU mit einer Güte von 0,999. Da man pro Kettenlänge die absolute Differenz auf die Gesamtgüte aufsummiert, besagt das Ergebnis wie sehr die Kurve der Optimal Verteilung approximiert wird. Für eine derartig niedrige Sequenzlänge sind die Ergebnisse nicht wirklich aussagekräftig. Dennoch kann beobachtet werden, dass der RANDU LCG am besten abschneidet, wobei das RANDU-Verfahren bei der seriellen Autokorrelation am schlechtesten bei niedrigen Sequenzlängen abgeschnitten hat. Dies zeigt, dass sich die Verfahren in der Erfassung der Güte deutlich unterscheiden.

Fall 5: Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 1000

Für diesen Testfall schneidet der NAG’s LCG am schlechtesten ab (150,15). Am besten verläuft das Testverfahren für den SIMSCRIPT-Generator mit 23,47.

Das NAG’s-Verfahren hat stärkere Ausreißer für die Anzahl mancher Kettenlängen. Also gibt es Stellen, an denen das Verfahren besonders häufig oder selten auf oder absteigende Zahlenfolgen erstellt, die an den Stellen der optimalen Verteilung liegen.

Der SIMSCRIPT hingegen hat eine relativ ähnliche Verteilung, wie die optimale Verteilung.

Fall 6: Alle LCG-Verfahren mit Sequenz-Up-Down-Test für eine Sequenzlänge von 100000

Das Maple’s LCG-Verfahren approximiert am besten die optimale Verteilung mit einer Gesamtdifferenz von 242,9. Der NAG’s LCG sticht deutlich mit einer schlechten Approximation hervor. Die Gesamtdifferenz liegt bei 15.000. Durch den immensen Zahlenraum und den hohen Startwert, sowie den großen Multiplikator werden die Zufallszahlen vor allem in der zweiten Hälfte des Zahlenraums generiert. Nun muss geprüft werden, ob die extremen Differenzen für kleine Zeichenketten(k) oder für große Zeichenketten(k) auftreten. Dies geschieht in den folgenden Testfällen.

Fall 7 & 8: Alle LCG-Verfahren mit Sequenzlänge 100.000 und einem k von 1 und 10.000

Der NAG’s LCG weist eine absolute Differenz von über 8.000 für k = 1 aufweist. Diese Differenz entsteht dadurch, dass es in dem NAG-Verfahren mehr Zeichenkettenfolgen mit der Länge 1 gibt. Der richtige Differenzwert zwischen optimaler Verteilung und NAG liegt bei -8.000. Dies bedeutet, dass es kaum immer größer werdende oder kleiner werdende Folgen gibt, sondern die Muster durch das Springen der Zufallszahlen entstehen. Es muss ein Muster vorhanden sein, sonst wäre der Wert bei der seriellen Autokorrelation nicht derartig schlecht.

Ansonsten sind keine großen Auffälligkeiten bei der Betrachtung verschiedener k mit gleicher Sequenzlänge zu sehen.

Fall 9 & 10 & 11:

Diese Testfälle zeigen die Funktionsweise der Polar-Methode, des eigenen Zufallszahlengenerators und der eigenen Güte-Funktion.

Bei der Anwendung der Standardnormalverteilung auf alle Zufallszahlen ist zu erkennen, dass die Werte relativ ähnlich bleiben. Es werden nur die Zahlenräume verschoben und die Gesamtzusammenhänge der Güte-Ergebnisse ändern sich nicht. Die Zufallszahlen wurden alle mit dem gleichen Verfahren von einem gleichverteilten Intervall [0,1] auf eine Standardnormalverteilung mit dem Intervall [-1,1] verschoben.

Der eigene Zufallszahlengenerator, welcher Datumsbasiert Zahlen generiert, besitzt eine nicht optimale Güte (-0,43 und 598 bei einer Sequenzlänge von 1.000). Dies könnte aber durch die Erweiterung des Moduls verbessert werden oder durch die Erweiterung der Datumsparameter.

Das eigene Güte-Testverfahren liefert den Mittelwert der Wechsel der Zahlenfolgen, d.h. wie häufig eine Folge mit stetigen Zahlen unterbrochen wird. Je höher dieser Wert ist, desto besser. In dem Testfall 11 lieferte der NAG’s LCG die besten Werte, da diese am höchsten waren mit ungefähr 0,75. Dies bedeutet, dass es kaum große Zahlenfolgen gibt, die sich stetig aufbauen.

10 Zusammenfassung und Ausblick

Alle Generatoren, außer dem NAG’s LCG, weisen ab einer bestimmten Länge eine starke Güte auf.

Der NAG’s LCG liefert stetige Wechsel. Dies wurde erfasst aus dem eigenen Güte-Testverfahren und Sequenz-Up-Down. Da der Zahlenraum extrem groß ist, unterscheiden sich die Differenzen zum Mittelwert stark und es resultiert eine schlechte Güte. Der NAG’s LCG ist also kein schlechter Generator, welcher nur Muster liefert, sondern die Testverfahren waren eher auf die anderen Generatoren angepasst und nicht auf den LCG.

Ein besonderer Generator ist der RANDU LCG, welcher nur ungerade Zahlen liefert. Er ist anfangs für kleine Sequenzlängen sehr schwach, aber je größer die Sequenzlänge wird, desto besser wird seine Güte.

Alle Generatoren, ausgenommen dem NAG LCG, können aufgrund der Testergebnisse angenommen werden. Denn ab einer bestimmten Sequenzlänge funktionieren alle Verfahren sehr gut und können verwendet werden. Für alle Generatoren sollte es eine Mindestsequenzlänge größer 10 geben, da die Ergebnisse für das Güte-Testverfahren kleiner 10 zu schlecht ausgefallen sind. Die Generatoren werden somit für Sequenzlängen kleiner 10 abgelehnt.

Aufgrund der zufälligen Generierung der Testmenge kann es vorkommen, dass für manche Güte-Testverfahren ein untypisches Ergebnis herauskommt. Um dies zu verhindern, könnte eine Erweiterungsmethode geschrieben werden, die mehrmals über ein Testverfahren geht und den Mittelwert aller Ergebnisse zurückgibt. Dadurch würden nur stabile Ergebnisse zurückgeliefert werden und die Ergebnisse würden sich kaum noch unterscheiden.

Um die Funktionalität des NAG’s LCG nachzuprüfen, müsste ein weiteres Verfahren benutzt werden, was einen differenzierten Blick auf die Güte wirft. Das bedeutet, dass nicht nur die paarweisen Abhängigkeiten oder die generierten Kettenlängen verglichen werden, sondern auch andere Eigenschaften, welche zu Mustern führen könnten. Diese Güte-Testverfahren könnten dann bessere Ergebnisse für den NAG LCG erzielen.

In weiteren Schritten der Programmierung könnte man die Klassenbibliothek um weitere LCG-Schemas erweitern. Damit stellt man eine breitere Funktionalität des LCG-Verfahrens bereit. Darüber hinaus kann man mehr Verteilungen an die Verteilungs-Schnittstelle anbinden. Dies ermöglicht die Transformation von Koordinaten oder Zahlen in andere Verteilungen. Dafür müsste man lediglich eine neue Klasse anlegen mit der dazugehörigen „Transformiere()“-Methode der Verteilungsschnittstelle. Zusätzlich kann man bestehende Verteilungen um Parameter erweitern zum Beispiel Varianz und Mittelwert der Normalverteilung oder Intervallgrenzen bei der Gleichverteilung.

Des Weiteren kann man Klassen an die Güte-Tests-Schnittstelle an klemmen. Man kann mehr Testverfahren benutzen um die Güte der verschiedenen Generatoren, welche erzeugt wurden, zu bestimmen. Dies kann man alles mit beliebigen Parametern für die Anzahl der zufallsgenerierten Elemente (Sequenzlänge) und die Ordnung k der Funktion ausführen, indem die Berechne()-Methode aus dem Interface „GüteTests“ verwendet wird. Die drei Schnittstellen sind Hauptbestandteile um diverse Funktionalitäten zu erweitern, neue Tests einzuführen, in andere Verteilungen umzuwandeln oder komplett andere Zufallszahlen zu erstellen.

11 Programmcode im Anhang

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Eine Schnittstelle für verschiedene Zufallszahlengeneratoren
    /// </summary>
    public interface Zufallsbibliothek
    {
        /// <summary>
        /// Generiert eine Zufallszahl in Abhängigkeit der Verteilung
        /// </summary>
        /// <returns>Zufallszahl zwischen 0-1</returns>
        double GeneriereZufallszahl();

        /// <summary>
        /// Generiert eine Gleichverteilte Zufallszahl
        /// </summary>
        /// <returns>Zufallszahl zwischen 0-1</returns>
        double GeneriereGleichverteilteZufallszahl01();

        /// <summary>
        /// Beschreibt die Klassenart des Generators
        /// </summary>
        /// <returns>Generatorenklasse</returns>
        Generatorenklasse GetArt();

        /// <summary>
        /// Beschreibt das Modul der angewendeten Generatorenklasse
        /// </summary>
        /// <returns>Modulo</returns>
        double GetM();

        /// <summary>
        /// Beschreibt die Verteilung der Generatorenklasse
        /// </summary>
        Verteilung Verteilung { get;set; }
    }

    /// <summary>
    /// Bestimmt die Klasse des Zufallsgenerators
    /// </summary>
    public enum Generatorenklasse {
        LCG,
        AnsiC,
        MinimalStandard,
        RANDU,
        SIMSCRIPT,
        NAGsLCG,
        MaplesLCG,
        Datumsbasiert
    }

}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Eine Schnittstelle für die angewendeten Verteilungen der Zufallszahlengeneratoren
    /// </summary>
    public interface Verteilung
    {
        /// <summary>
        /// Transformiert die x Variable in die entsprechende Verteilung
        /// </summary>
        /// <param name="x">Wert</param>
        /// <returns>Transformiertes x</returns>
        double Transformiere(double x);

        /// <summary>
        /// Beschreibt die Art der Verteilung
        /// </summary>
        V Art { get; }

        /// <summary>
        /// Beschreibt den Mittelwert, den die jeweilige Verteilung aufweist
        /// </summary>
        double Mittelwert { get; }
    }

    /// <summary>
    /// Beschreibt die Art der Verteilung
    /// </summary>
    public enum V
    {
        Gleichverteilung,
        Standardnormalverteilung
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Beschreibt eine Schnittstelle für verschiedene Güte-Testverfahren
    /// </summary>
    public interface GüteTests
    {
        /// <summary>
        /// Berechnet einen Richtwert, um eine Aussage über die Güte eines Verfahrens
machen zu können
        /// </summary>
        /// <param name="k">0 oder beliebig</param>
        /// <param name="anz">Sequenzlänge</param>
        /// <returns></returns>
        double Berechne(int k, int anz);
    }

    /// <summary>
    /// Beschreibt das Güte-Testverfahren
    /// </summary>
    public enum GüteTestverfahren
    {
        SerielleAutokorrelation,
        SequenzUpDown,
        Eigen
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt einen Linearen Kongruenz-Generator
    /// </summary>
    public class LCG : Zufallsbibliothek
    {
        private Generatorenklasse Art;
        private double[] Parameter;

        public Verteilung Verteilung { get; set; } = new Gleichverteilung();

        /// <summary>
        /// Erstellt eine Instanz eines LCGs
        /// </summary>
        /// <param name="art">Zufallszahlengenerator</param>
        /// <param name="p">Parameter</param>
        public LCG(Generatorenklasse art ,double[] p)
        {
            this.Art = art;
            Parameter = p;
        }

        /// <summary>
        /// Generiert eine Zufallszahl in Abhängigkeit der gegebenen Parameter und
speichert den xi-Parameter nach Schleifendurchlauf.
        ///
        /// Es wird auch abgefragt, welche Verteilung hier angewendet werden soll und dann
ggflls. transformiert.
        /// </summary>
        /// <returns>Zufallszahl mit gegebener Verteilung</returns>
        public double GeneriereZufallszahl()
        {
            double x = GeneriereGleichverteilteZufallszahl01();

            return Verteilung.Transformiere(x);
        }

        /// <summary>
        /// Berechnet eine Gleichverteilte Zufallszahl nach dem Prinzip der erhaltenen
Parameter
        /// </summary>
        /// <returns>Zufallszahl</returns>
        public double GeneriereGleichverteilteZufallszahl01()
        {
            double m = Parameter[0]; double a = Parameter[1]; double c = Parameter[2];
double x = Parameter[3];
            double wertBisMudolo = ((a * x) + c) % m;
            double wert = (double)wertBisMudolo / m;
            Parameter[3] = wertBisMudolo;
            return wert;
        }
```

```
/// <summary>
        /// Gibt das Modulo zurück
        /// </summary>
        /// <returns>Modulo</returns>
        public double GetM()
        {
            return this.Parameter[0];
        }

        /// <summary>
        /// Gibt die Generatorenklasse zurück
        /// </summary>
        /// <returns>Generatorenklasse</returns>
        public Generatorenklasse GetArt()
        {
            return this.Art;
        }
    }

    /// <summary>
    /// Konstanten-Klasse für die verschiedenen LCG-Verfahrensparameter
    /// </summary>
    public class LCGVerfahrenParameter
    {
        public static readonly double[] AnsiC = new double[] { Math.Pow(2, 31),
1103515245, 12345, 12345 };
        public static readonly double[] MinimalStandard = new double[] { Math.Pow(2,
31) - 1, 16807, 0, 1 };
        public static readonly double[] RANDU = new double[] { Math.Pow(2, 31), 65539,
0, 1 };
        public static readonly double[] SIMSCRIPT = new double[] { Math.Pow(2, 31) -
1, 630360016, 0, 1 };
        public static readonly double[] NAGsLCG = new double[] { Math.Pow(2, 59),
Math.Pow(13, 13), 0, 123456789 };
        public static readonly double[] MaplesLCG = new double[] { Math.Pow(10, 12) -
11, 427419669081, 0, 1 };
    }
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt einen eigenen datumsbasierten Zufallszahlengenerator
    /// </summary>
    public class Datumsbasiert : Zufallsbibliothek
    {
        Generatorenklasse Art;
        int i;
        double m;

        /// <summary>
        /// Erstellt eine Instanz des datumsbasierten Zufallszahlengenerator
        /// </summary>
        public Datumsbasiert()
        {
            i = 0;
            m = 1000;
            Art = Generatorenklasse.Datumsbasiert;
        }

        public Verteilung Verteilung { get; set; }

        /// <summary>
        /// Generiert eine Gleichverteilte Zufallszahl im Bereich 0-1
        /// </summary>
        /// <returns>Zufallszahl im Intervall 0-1</returns>
        public double GeneriereGleichverteilteZufallszahl01()
        {
            DateTime now = DateTime.Now;
            double additionen = now.Second + now.Minute + now.Hour + now.Day + now.Month +
System.Environment.TickCount;
            this.i++;

            return (double)((additionen * i) % m) / m;  //Berechnet eine Zufallszahl,
indem die aufsummierten Zeitangaben mit dem Wert für die Anzahl der bisher generierten
Zufallzahlen multipliziert
                                                        //werden und dann durch das Modul
auf ein Intervall zwischen 0 und 1 geregelt werden
        }

        /// <summary>
        /// Generiert auf Basis der jetzigen Zeit und der Systemzeit eine Zufallszahl[0,1]
in Abhängigkeit der i-ten Zufallszahl, welche generiert wird.
        /// </summary>
        /// <returns>Zufallszahl im Intervall 0-1</returns>
        public double GeneriereZufallszahl()
        {
            double x = GeneriereGleichverteilteZufallszahl01();

            return Verteilung.Transformiere(x);
        }
```

```
        public Generatorenklasse GetArt()
        {
            return this.Art;
        }

        public double GetM()
        {
            return this.m;
        }
    }
}

using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt eine Standardnormalverteilung
    /// </summary>
    public class Standardnormalverteilung : Verteilung
    {
        private Zufallsbibliothek Generator;
        private int[] Verteilungsparameter;

        public V Art { get; private set; }
        public double Mittelwert { get { return 0; } }

        /// <summary>
        /// Erzeugt eine Instanz einer Standardnormalverteilung
        /// </summary>
        /// <param name="generator">Zufallszahlengenerator</param>
        public Standardnormalverteilung(Zufallsbibliothek generator)
        {
            this.Generator = generator;
            Verteilungsparameter = new int[] { 0, 1 };
            Art = V.Standardnormalverteilung;
        }

        /// <summary>
        /// Polarmethode wie im Arbeitsbogen beschrieben
        /// </summary>
        /// <param name="x">Zufallszahl</param>
        /// <param name="b">Bibliothek um Zufallszahlen zu generieren</param>
        /// <returns>Returnt eine von den beiden generierten unabhängigen
Standardnormalverteilten Zufallszahlen (x*p; y*p) Skaliert mit p</returns>
        private double Transformiere(double x, double y)
        {
            var q = Math.Pow(x, 2) + Math.Pow(y, 2);
            var p = Math.Sqrt((-2 * Math.Log(q)) / q);
            return x * p;
        }
```

```
/// <summary>
        /// Polarmethode wie im Arbeitsbogen beschrieben
        /// </summary>
        /// <param name="x">Zufallszahl</param>
        /// <param name="b">Bibliothek um Zufallszahlen zu generieren</param>
        /// <returns>Returnt eine von den beiden generierten unabhängigen
Standardnormalverteilten Zufallszahlen (x*p; y*p) Skaliert mit p</returns>
        public double Transformiere(double x)
        {
            x = x * 2 - 1; // In [-1,1] Verteilung ändern
            double rand = 0;
            double q = 0;
            do
            {
                rand = Generator.GeneriereGleichverteilteZufallszahl01();
                rand = rand * 2 - 1;
                q = Math.Pow(x, 2) + Math.Pow(rand, 2);
            } while ( q == 0 || q >= 1);
            return Transformiere(x, rand);
        }
    }
}


using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt eine Gleichverteilung
    /// </summary>
    public class Gleichverteilung : Verteilung
    {
        public V Art { get; private set; }

        public double Mittelwert { get { return 0.5; } }

        /// <summary>
        /// Erstellt eine Instanz der Gleichverteilung
        /// </summary>
        public Gleichverteilung()
        {
            Art = V.Gleichverteilung;
        }

        /// <summary>
        /// Transformiert in die Gleichverteilung
        /// Da aber nur gleichverteilte Zahlen bisher übergeben werden, muss nichts
transformiert werden.
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
        public double Transformiere(double x)
        {
            return x;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt eine Serielle Autokorrelation
    /// </summary>
    public class SerielleAutokorrelation : GüteTests
    {
        Zufallsbibliothek Bibliothek;
        string Name;

        /// <summary>
        /// Erstellt die Instanz einer seriellen Autokorrelation
        /// </summary>
        /// <param name="b"></param>
        public SerielleAutokorrelation(Zufallsbibliothek b)
        {
            this.Name = "SerielleAutokorrelation";
            this.Bibliothek = b;
        }

        /// <summary>
        /// Errechnet den Korrelationswert für das Zufallsgenerierte Array
        /// </summary>
        /// <param name="k">Abstand der verglichenen Zufallszahlen</param>
        /// <returns>Korrelationswert des Zufallszahlengenerators</returns>
        public double Berechne(int k, int anz)
        {
            double[] zahlen = new double[anz];
            for(int i=0; i<anz; i++)
            {
                zahlen[i] = Bibliothek.GeneriereZufallszahl();
            }

            if(k == 0)//wenn kein spezifisches k gewählt wurde, wird ein zufälliger
Abstand zweier Zahlen ausgerechnet
            {
                int punkt1 = (int)(Bibliothek.GeneriereGleichverteilteZufallszahl01() *
anz);
                int punkt2 = (int)(Bibliothek.GeneriereGleichverteilteZufallszahl01() *
anz);
                k = Math.Abs(punkt1 - punkt2);
            }
            double mittelwert = Bibliothek.Verteilung.Mittelwert; // Berechnet Mittelwert
aus allen Zufallszahlen
            double zaehler = 0;
            double nenner = 0;
            int j = 0;
            while(j < zahlen.Length - k) // Führt die Serielle Autokorrelation durch
            {
                zaehler += (zahlen[j] - mittelwert) * (zahlen[j + k] - mittelwert);
                nenner += Math.Pow(zahlen[j] - mittelwert, 2);
                j++;
            }
            return (double) zaehler / nenner;
        }}}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt ein Sequenz-Up-Down-Test
    /// </summary>
    public class SequenzUpDownTest : GüteTests
    {
        Zufallsbibliothek Bibliothek;
        string Name;

        /// <summary>
        /// Erstellt die Instanz eines Sequenz-Up-Down-Test
        /// </summary>
        /// <param name="bib">Zufallszahlengenerator</param>
        public SequenzUpDownTest(Zufallsbibliothek bib)
        {
            this.Name = "SequenzUpDownTest";
            this.Bibliothek = bib;
        }

        /// <summary>
        /// Berechnet die Anzahl der Bitfolgen mit der Länge K.
        /// </summary>
        /// <param name="k">Wenn 0 dann werden alle möglichen k überprüft, ansonsten nur
das angegebene k.</param>
        /// <param name="anz">Bestimmt die Sequenzlänge des Tests. Also die Anzahl der
untersuchten Zufallszahlen</param>
        /// <returns></returns>
        public double Berechne(int k, int anz)
        {
            double[] zahlen = new double[anz];
            for (int i = 0; i < anz; i++) //Erstelle Zahlenarray
            {
                zahlen[i] = Bibliothek.GeneriereZufallszahl();
            }
            int[] bitmaske = new int[zahlen.Length-1];
            for (int i = 0; i < zahlen.Length - 1; i++) //Erstelle Bitmaske
            {
                if (zahlen[i] < zahlen[i + 1])
                {
                    bitmaske[i] = 1;
                }
                else
                {
                    bitmaske[i] = 0;
                }
            }

            Dictionary<int, int> kettenlaengen = new Dictionary<int, int>();
            int kettenlaenge = 1; //Speichert die Anzahl der jetzigen Folge
            int vorherigesBit = bitmaske[0];
            for (int i = 1; i < bitmaske.Length; i++)
            {
```

```
if (vorherigesBit == bitmaske[i])
                {
                    kettenlaenge++;
                }
                else //Prüft, ob ein Wechsel stattfindet um den Counter zurückzusetzen und
ggfls. eine gefundene Bitfolge mit k Elementen zu speichern
                {
                    if (kettenlaengen.ContainsKey(kettenlaenge))
                    {
                        kettenlaengen[kettenlaenge]++;
                    }
                    else
                    {
                        kettenlaengen[kettenlaenge] = 1;
                    }
                    kettenlaenge = 1;
                    vorherigesBit = bitmaske[i];
                }
            }
            if (k == 0)
            {
                double differenz = 0;
                foreach (var eintrag in kettenlaengen)
                {
                    differenz += Math.Abs(BerechneNopt(eintrag.Key, anz) - eintrag.Value);
//Errechnet die Differenz der gefundenen Zahlenketten mit dem des optimalen Wertes
                }
                return differenz;
            }
            else
            {
                return Math.Abs(BerechneNopt(k, anz) - kettenlaengen[k]);
            }
        }

        /// <summary>
        /// Berechnet das optimale Ergebnis für bestimmte Kettenlängen
        /// </summary>
        /// <param name="k">Länge der Kette</param>
        /// <param name="anz">Sequenzlänge</param>
        /// <returns>Richtwert</returns>
        private double BerechneNopt(int k, int anz)
        {
            double zaehler = (Math.Pow(k, 2) + 3 * k + 1) * anz - (Math.Pow(k, 3) + 3 *
Math.Pow(k, 2) - k - 4);
            double nenner = Fakultaet(k + 3) / 2;
            return zaehler / nenner;
        }

        private double Fakultaet(double zahl)
        {
            if (zahl < 2)
                return zahl;
            return zahl * Fakultaet(zahl - 1);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Zufallsklassen
{
    /// <summary>
    /// Erzeugt ein eigenes Güte-Testverfahren
    /// </summary>
    public class EigenesGüteTestverfahren : GüteTests
    {
        Zufallsbibliothek Bibliothek;
        string Name;
        /// <summary>
        /// Erstellt die Instanz der eigenen
        /// </summary>
        /// <param name="bib"></param>
        public EigenesGüteTestverfahren(Zufallsbibliothek bib)
        {
            this.Name = "Eigen";
            this.Bibliothek = bib;
        }
        /// <summary>
        /// Angelehnt an die SequenzUpDown werden hier die Wechsel der Größen geprüft und
dann im Anschluss durch die Anzahl der möglichen Wechsel dividiert.
        /// Soll Aufschluss über die mittleren Wechsel geben. Wenn die Werte um die 0,5
oder höher liegen, liegt eine gute Güte vor! Ansonsten eine schlechte, da die Folgen sehr
lang sind.
        /// </summary>
        /// <param name="k">Sequenzlänge</param>
        /// <returns>Den Mittelwert aller Kettenwechsel.</returns>
        public double Berechne(int k, int anz)
        {
            double[] zahlen = new double[anz]; //Zufallszahlen erstellen
            for (int i = 0; i < anz; i++)
            {
                zahlen[i] = Bibliothek.GeneriereZufallszahl();
            }
            int[] bitmaske = new int[zahlen.Length-1]; //Bitmaske erstellen
            for (int i = 0; i < zahlen.Length - 1; i++)
            {
                if (zahlen[i] < zahlen[i + 1])
                {
                    bitmaske[i] = 1;
                }
                else
                {
                    bitmaske[i] = 0;
                }
            }
            long summe = 0;
            for(int i=0; i<bitmaske.Length-1; i++) //Bildet Summe der Wechsel
            {
                if (bitmaske[i] != bitmaske[i + 1])
                {
                    summe++;
                }
            }
            return (double)summe / bitmaske.Length; //Mittelwert der Wechsel
        }}}
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace GroProVorb
{
    public class Ausgabe
    {
        string Pfad;
        public Ausgabe(string pfad)
        {
            this.Pfad = pfad;
        }

        public void Schreiben(string[] ergebnisse)
        {
            try
            {
                using (var writer = new StreamWriter(Pfad))
                {
                    foreach (var ergebniss in ergebnisse)
                    {
                        writer.WriteLine(ergebniss);
                    }
                }
            }
            catch (IOException e)
            {
                Console.WriteLine("Die Datei konnte nicht gefunden werden. E: " + e);
            }
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using Zufallsklassen;

namespace GroProVorb
{
    public class Eingabe
    {
        public List<Zufallsbibliothek> Zufallsgeneratoren;
        public List<GüteTestverfahren> GüteTestVerfahren;
        public int Sequenzlänge;
        public int K;
        public V Verteilung;

        public void Einlesen(string pfad)
        {
            try
            {
                using (var reader = new StreamReader(pfad))
                {
                    string[] line1 = reader.ReadLine().Split(','); //Zufallsgeneratoren
                    string[] line2 = reader.ReadLine().Split(','); //Testverfahren
                    string line3 = reader.ReadLine(); //SampleSize-Groeße der generierten
Arrays
                    string line4 = reader.ReadLine(); //Sequenzlänge der testverfahren
                    string line5 = reader.ReadLine(); //Angabe ob Gleichverteilt oder
Standardnormalverteilt
                    Verteilung = line5 == "0" ? V.Gleichverteilung :
V.Standardnormalverteilung;
                    GeneratorenKonvertieren(line1);
                    VerfahrenKonvertieren(line2);
                    Sequenzlänge = int.Parse(line3);
                    K = int.Parse(line4);
                }
            }
            catch (IOException e)
            {
                Console.WriteLine("Die Datei konnte nicht eingelesen werden. E: " + e);
            }
        }

        private void GeneratorenKonvertieren(string[] a)
        {
            Zufallsgeneratoren = new List<Zufallsbibliothek>();
            foreach (var generator in a)
            {
                if (generator == ((int)Generatorenklasse.AnsiC).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.AnsiC,
LCGVerfahrenParameter.AnsiC));
                }
                if (generator == ((int)Generatorenklasse.MinimalStandard).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.MinimalStandard,
LCGVerfahrenParameter.MinimalStandard));
                }
```

```
if (generator == ((int)Generatorenklasse.RANDU).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.RANDU,
LCGVerfahrenParameter.RANDU));
                }
                if (generator == ((int)Generatorenklasse.SIMSCRIPT).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.SIMSCRIPT,
LCGVerfahrenParameter.SIMSCRIPT));
                }
                if (generator == ((int)Generatorenklasse.NAGsLCG).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.NAGsLCG,
LCGVerfahrenParameter.NAGsLCG));
                }
                if (generator == ((int)Generatorenklasse.MaplesLCG).ToString())
                {
                    Zufallsgeneratoren.Add(new LCG(Generatorenklasse.MaplesLCG,
LCGVerfahrenParameter.MaplesLCG));
                }
                if (generator == ((int)Generatorenklasse.Datumsbasiert).ToString())
                {
                    Zufallsgeneratoren.Add(new Datumsbasiert());
                }
            }
            foreach (var generator in Zufallsgeneratoren)
            {
                generator.Verteilung = Verteilung == V.Gleichverteilung ?
(Verteilung)new Gleichverteilung() : new Standardnormalverteilung(generator);
            }
        }

        private void VerfahrenKonvertieren(string[] a)
        {
            GüteTestVerfahren = new List<GüteTestverfahren>();
            foreach (var verfahren in a)
            {
                if (verfahren ==
((int)GüteTestverfahren.SerielleAutokorrelation).ToString())
                {
                    GüteTestVerfahren.Add(GüteTestverfahren.SerielleAutokorrelation);
                }
                if (verfahren == ((int)GüteTestverfahren.SequenzUpDown).ToString())
                {
                    GüteTestVerfahren.Add(GüteTestverfahren.SequenzUpDown);
                }
                if (verfahren == ((int)GüteTestverfahren.Eigen).ToString())
                {
                    GüteTestVerfahren.Add(GüteTestverfahren.Eigen);
                }
            }
        }
    }
}
```

```
using System;
using System.IO;
using System.Reflection;
using Zufallsklassen;

namespace GroProVorb
{
    class Program
    {
        static void Main(string[] args)
        {
            string dateipfad;
            if (args.Length > 0)
            {
                // Parameter zum einlesen der Datei
                dateipfad = args[0];
            }
            else
            {
                Assembly asm = Assembly.GetExecutingAssembly();
                string path = System.IO.Path.GetDirectoryName(asm.Location);
                // Standard Input verwenden
                dateipfad = path + "\\Tests\\AlleTestsMitSequenzlänge100000.txt";
            }
            //Einlesen der Daten
            Eingabe eingabe = new Eingabe();
            eingabe.Einlesen(dateipfad);

            // Ausgabe-Dateinamen festlegen (NameEingabedatei_Ergebnisse.txt)
            dateipfad = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location)+
"\\Ergebnisse"
               + @"\" + Path.GetFileNameWithoutExtension(dateipfad) + "_Ergebnisse.txt";


            string[] save = new string[999];
            save[0] = "Die Sequenzlänge der folgenden Ergebnisse beträgt: " +
eingabe.Sequenzlänge + " und die Ordnung(falls ungleich 0) beträgt: " + eingabe.K;
            save[1] = "Die angewandte Verteilung der Zufallszahlen ist die " +
eingabe.Verteilung.ToString();
            int i = 2;
            foreach (var generator in eingabe.Zufallsgeneratoren)
            {
                foreach (var verfahren in eingabe.GüteTestVerfahren)
                {
                    double ausgabe = double.MaxValue;
                    if (verfahren == GüteTestverfahren.SerielleAutokorrelation)
                    {
                        var v = new SerielleAutokorrelation(generator);
                        ausgabe = v.Berechne(eingabe.K, eingabe.Sequenzlänge);
                    }
                    else if (verfahren == GüteTestverfahren.SequenzUpDown)
                    {
                        var v = new SequenzUpDownTest(generator);
                        ausgabe = v.Berechne(eingabe.K, eingabe.Sequenzlänge);
                    }
                    else if (verfahren == GüteTestverfahren.Eigen)
                    {
                        var v = new EigenesGüteTestverfahren(generator);
                        ausgabe = v.Berechne(eingabe.K, eingabe.Sequenzlänge);
```

```
}
                    string a = "Generator: " + generator.GetArt().ToString() + ",
Verfahren: " + verfahren.ToString() + ", Wert: " + ausgabe;
                    Console.WriteLine(a);
                    save[i] = a;
                    i++;
                }
            }
            Ausgabe output = new Ausgabe(dateipfad);
            output.Schreiben(save);
        }
    }
}
```