Beispiellösung zu Aufgabe 10.3c)

In Aufgabe 10.3c) waren mittels zweier verschiedener Verfahren standardnormalverteilte Zufallsvariablen jeweils 100.000 mal zu simulieren.

Methode aus 5.3b) (Inversionsmethode):
Es wird eine gleichverteilte Zufallsvariable $U \in [0, 1]$ generiert. Die gesuchte Variable ergibt sich als $\Phi^{-1}(U)$, wobei Φ die kummulierte Dichte der Standardnormalverteilung darstellt. Da die Standardnormalverteilung nicht geschlossen darstellbar ist, muss man für ihre Invertierung auf numerische Verfahren zurückgreifen. In der implementierten Lösung wird dafür das Integral $\int_{-\infty}^{x} f(x)dx \approx \int_{-10}^{x} f(x)dx$ mittels der Trapezregel iterativ für regelmäßig im Intervall $[-10, 10]$ verteilte Werte x berechnet. Die Funktion $f(x) = e^{\frac{-x^2}{2}}$ gibt dabei die Dichte der Standardnormalverteilung an. Die Invertierung der Zufallsvariable U erfolgt nun mittels binärer Suche.
Diese Methode benötigt eine Vorberechnungszeit, die von der Genauigkeit der Rasterung der kummulierten Dichte abhängt, die auch die Zahl der signifikanten Stellen der simulierten Zufallsvariable bestimmt. Die Laufzeit für die Vorberechnung liegt in $\mathcal{O}(k)$, mit k als Anzahl der Rasterpunkte. Zur Simulation von n Zufallsvariablen liegt die Laufzeit dann in $\mathcal{O}(n * log(k))$. Ist s die Anzahl der signifikanten Stellen, so bedeutet das, dass die Vorberechnungszeit in $\mathcal{O}(e^s)$, die Simulationszeit in $\mathcal{O}(n * s)$ liegt.
Methode aus 10.3b) (Box-Muller-Methode):
Es werden zwei gleichverteilte Zufallsvariablen $U, V \in [0, 1]$ generiert. Wie in Aufgabe 10.3b) gezeigt, ergibt sich dann eine normalverteilte Zufallsvariable als $\sqrt{-2log(U)} * sin(2\pi V)$. Die Laufzeit liegt in $\mathcal{O}(n)$.

Die Simulation der Variablen wurde in Python implementiert, die Plots durch matplotlib erzeugt. Für die Generierung der gleichverteilten Variablen wurde der eingebaute Zufallszahlgenerator mit einem festen Seed verwendet. Die Exponentialfunktion, der Sinus, der Logarithmus und die Wurzel wurden mittels der eingebauten Funktionen berechnet. Im Folgenden ist eine Beispielausführung des Skripts samt der beiden Histogramme angegeben.

```
Vorberechnungszeit für Methode aus 5.3b) in Mikrosekunden: 1428580
Zeit für Methode aus 5.3b) in Mikrosekunden: 429663
Zeit für Methode aus 10.3b) in Mikrosekunden: 76321
```

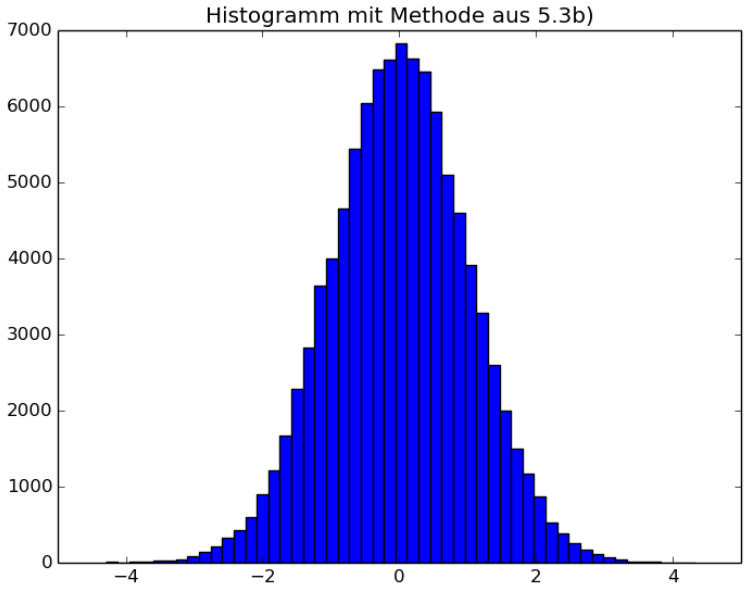

Abbildung 1: Histogramm für Methode aus 5.3b)

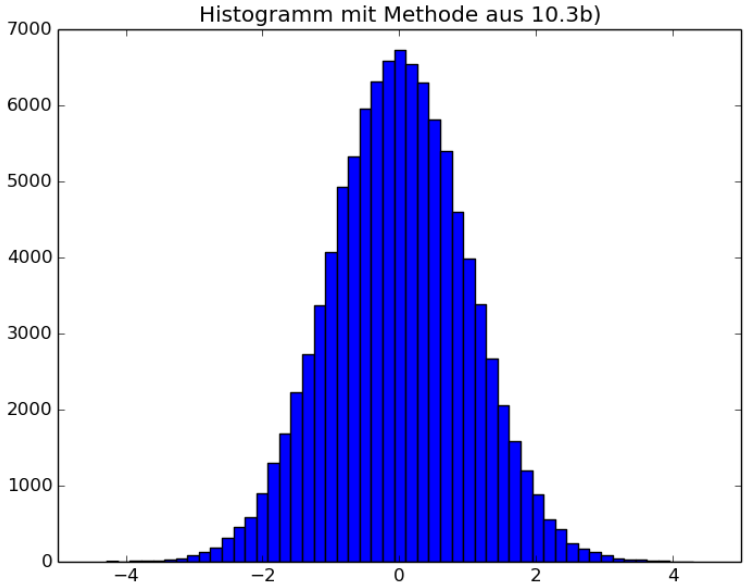

Abbildung 2: Histogramm für Methode aus 10.3b)

Zusammenfassend ist festzustellen, dass die Methode aus 10.3b) der Methode aus 5.3b) deutlich überlegen ist. Sie weist eine deutlich bessere Laufzeit auf, benötigt keine Vorberechung und ist in der Anzahl der signifikanten Stellen nur durch die zugrundeliegende Arithmetik beschränkt und nicht, wie in 5.3b), durch die Laufzeit der Vorberechnung. Die Zahl der Rasterpunkt wächst hier exponentiell mit der Anzahl der signifikanten Stellen. Da jeder Wert gespeichert werden muss, wächst damit auch der Speicherbedarf exponentiell, während er bei der Box-Muller-Methode konstant ist. Zugute kommt dieser Methode außerdem, dass für die Wurzel-, Sinus- und Logarithmusfunktion bereits ausgesprochen effektive Implementationen vorhanden sind, die es für die inverse kummulierte Dichte nicht gibt.
Abschließend ist diesem Dokument noch der verwendete Quellcode angefügt.

```
#!/usr/bin/python
# -*- coding: iso-8859-15 -*-
# Blatt 10 - Aufgabe 3c)
# Jannes Münchmeyer
# Undine Falkenhagen
# Quelltext online unter http://www.math.hu-berlin.de/~mnchme/blatt10.py
#

import random,math,datetime,matplotlib.pyplot as plt

random.seed(42) #answer to live, the universe and everything
default_step=0.00001 #Ergibt die Zahl der signifikanten Stellen für Methode 5.3b)

#Dichtefunktion
def density(x):
        return 1/math.sqrt(2*math.pi)*math.exp(-x*x/2)

#Generator für Liste der kummulierten Dichte
def generate_normcdf(start,stop,step=default_step):
    df_sum=0
    last=0
    while start<stop:
        yield df_sum
        start += step
```

```
        val=density(start)
        df_sum += (last+val)/2*step
        last=val

#Methode aus 10.3b)
def bin10():
        u=random.random()
        v=random.random()
        r=math.sqrt(-2*math.log(u))
        return r*math.sin(v*2*math.pi)

#Methode aus 5.3b)
#Binäre Suche zum Invertieren
def bin5(cdf):
        u=random.random()
        low=0
        high=len(cdf)
        while (high-low>1):
                mid=(high+low)/2
                val=cdf[mid]
                if (val>u):
                        high=mid
                if (val<u):
                        low=mid
        return low*default_step-10.0

t1=datetime.datetime.now()
precomp_cdf=[x for x in generate_normcdf(-10,10)]
t2=datetime.datetime.now()

a = datetime.datetime.now()
val10=[(bin10()) for x in range(0,100000)]
b = datetime.datetime.now()
val5=[(bin5(precomp_cdf)) for x in range(0,100000)]
c = datetime.datetime.now()

d2=b-a
d1=c-b
d3=t2-t1

print('Vorberechnungszeit für Methode aus 5.3b) in
```

```
    Mikrosekunden: '+str(d3.seconds*1000000+d3.
    microseconds))
print('Zeit für Methode aus 5.3b) in Mikrosekunden:
    '+str(d1.seconds*1000000+d1.microseconds))
print('Zeit für Methode aus 10.3b) in Mikrosekunden:
     '+str(d2.seconds*1000000+d2.microseconds))

#Erstellen der Histogramme der beiden Methoden
#plt.xkcd()
plt.figure(1)
plt.hist(val10,bins=51)
plt.title('Histogramm mit Methode aus 10.3b)')
plt.show(block=False)

plt.figure(2)
plt.hist(val5,bins=51)
plt.title('Histogramm mit Methode aus 5.3b)')
plt.show()
```

Listing 1: Quellcode zu 10.3c)