Punkt-Warping: Inversionsmethode mit Hash-Tabelle

Lukas Knirsch

***Zusammenfassung*—Bei wissenschaftlichen Simulationen oder bei hochqualitativen Schätzberechnungen werden sehr große Datensätze benötigt, die nach bestimmten Merkmalen verteilt sind [3]. Diese Daten können häufig durch Funktionen wie eine Normalverteilung oder durch multivariante Gaussche Verteilungen beschrieben werden. Umso gleichmäßiger die Daten aber verteilt sind, umso genauer können die verwendeten Algorithmen die Simulationen oder Schätzungen berechnen. Allerdings besitzen die dafür bereits verwendeten Methoden alle aber verschiedene Nachteile. Einer der häufigsten Nachteile ist der hohe Rechenaufwand für die ungefähr gleichmäßige Verteilung von Proben im Raum.**

Aus diesem Grund soll in dieser Ausarbeitung die Funktionsweise und eine Implementierung der hash-basierten Inversionsmethode erarbeitet werden und ihr möglicher Einsatz bei der gleichmäßigen Verteilung von Daten und Punkt-Warping im Allgemeinen.

I. Einleitung

In dieser Ausarbeitung soll eine Methode beschrieben werden, mit deren Hilfe man durch endlich viele Zufallszahlen eine ungefähr gleichmäßige Verteilung von Punkten im Raum erzeugen kann. Damit kann man sehr gut Daten simulieren, welche eine bestimmte Verteilung besitzen und, wie in der Natur üblich, keine harten Kanten besitzen oder komplett zufällig verteilt sind, sondern langsam abfallend strukturiert sind. Zu diesen Dichteverteilungen gehört zum Beispiel auch die Gaußschen Normalverteilung.

Falls nicht anders beschrieben, werden in dieser Ausarbeitung für die randomisierten Eingaben sogenannte ”Golden Ratio Sequences”verwendet. Diese wurden von Schretter [5] vorgestellt und verteilen zufällig uniform verteilte Punkte im $[0, 1]$-Quadrat, wie in 1 zu sehen ist. Durch die Inversionsmethode [2] können diese Punkte dann auf alle möglichen Größen und Verteilungen gekrümmt werden. Deshalb werden diese gleichmäßig verteilten Punkte auf die von Chen und Asau [1] und Devroye [2] beschriebene, verbesserte Inversionsmethode mit Hashing angewandt, wodurch das gewünschte Ergebnis erzielt wird. Diese Verfahrenskette wird unter Implementation näher erläutert und durchgeführt.

A. Problemstellung

Mithilfe dieser Arbeit über die Hash-basierte Inversionsmethode, soll gezeigt werden, wie diese funktioniert und ob sie bei der Generierung von multivariaten Verteilungen in höheren Dimensionen mithilfe von uniformen Dichten immer noch korrekt arbeitet und einsetzbar ist. Außerdem soll verglichen werden, ob sich der Einsatz der hier beschriebenen Methode als Ersatz von bereits existierenden anderen Methoden lohnt.

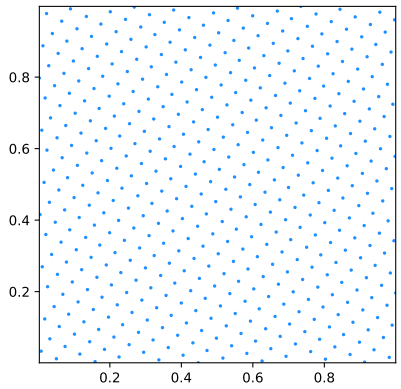


Abbildung 1: 500 gleichmäßige, zufällige Punkte einer Goldenen-Schnitt-Sequenz [5].

Name	Funktion	Zufällige Variable
Exponentiell	$1 - e^{-x}$	$\log(1/U)$
Logistisch	$1/(1 + e^{-x})$	$-\log(\frac{1-U}{U})$
Cauchy	$1/2 + (1/\pi)\arctan(x)$	$\tan(\pi U)$

Tabelle I: Explizit invertierbare Verteilungsfunktionen [2]

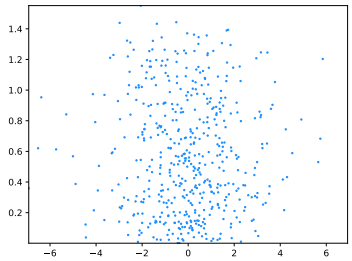
B. Stand der Technik

Um das Inverse einer Funktion anzunähern, kann nicht nur die hash-basierte Inversionsmethode verwendet werden, sondern beispielsweise auch ein Feld der Partialsummen der einzelnen Wahrscheinlichkeiten, einen Punkt an einer bestimmten Stelle anzutreffen. Auf diesem Feld können dann wiederum unterschiedliche Methoden zum Finden der passenden Werte verwendet werden. Dazu gehören lineare Suche von vorne nach hinten, binäre Suche oder auch ein Huffman-Baum.

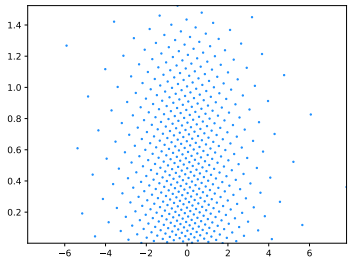
Zur Generierung uniformer Zufallszahlen gibt es anstelle der Sequenzen des goldenen Schnittes auch Methoden wie die von Halton [6], sogenannten Blue Noise [7] oder Rang-1-Gitter [4]. Doch da Schretter in seinem Paper zeigt, dass die von diesen Methoden generierten Zahlen nicht so gleichmäßig verteilt werden, wie von seiner neuen Methode [5], wird hier diese verwendet.

II. Funktionsweise

Dieser Abschnitt erklärt die Arbeitsweise der Inversionsmethode im Allgemeinen. Danach wird kurz das Prinzip der Hashtabelle wiederholt, um danach die Funktionsweise der Hash-basierten Inversionsmethode zu erläutern.



(a) Eingabewerte zufällig generiert mit `random`



(b) Eingabewerte aus einer Goldenen Schnitt Sequenz

Abbildung 2: 500 Punkte mit einer logistischen Dichte auf der X- und einer sinusoiden auf der Y-Achse

A. *Inversionsmethode*

Devroye [2] beschreibt in seinem Buch die Inversionsmethode als eine Methode, bei der mithilfe des Inversen F^{-1} einer bekannten Verteilungsfunktion F und mit einer uniformen, zufälligen Zahl $U \in [0,1]$ eine zufällige Zahl $X = F^{-1}(U)$ generiert wird. Für schnell berechenbare und bekannte Inverse, zu denen die Funktionen in Tabelle I gehören, ist diese Methode sehr einfach umzusetzen. Dadurch, dass die erzeugten Zahlen aus fortlaufenden Zufallszahlen entstehen, gibt es eine monotone Beziehung zwischen U und X. Das führt dazu, dass ein Paar (x_1, x_2), bei dem X_1 und X_2 maximal anti-korreliert zueinander sind, mit

$$(F^{-1}(U),\, F^{-1}(1-U))$$

berechnet werden kann. Diese Eigenschaft wird vor allem in Simulationen benötigt.

Da im Folgenden nur noch von gleichmäßig verteilten Ausgangswerten gesprochen wird, wird kurz erklärt, warum sich diese Ausarbeitung nur darauf beschränkt. In Abbildung 2 sind zwei Abbildungen zu sehen. Beide Abbildungen bilden 500 Punkte $y \in [0,1]$ auf 500 andere Punkte ab. Dabei wird der X-Achsenanteil der Punkte mithilfe einer logarithmischen Funktion, siehe I, und der Y-Achsenanteil mit einer Sinusfunktion verformt. In 2a sind die gewählten Ausgangswerten mit der `uniform`- Funktion des Pythonpaketes *random* zufällig generiert worden. Zum einen lässt sich dadurch die Funktion nicht genauso gut erkennen, wie mit nicht-zufälligen Punkten. Zum anderen entstehen dadurch die für computergenerierte, naive Zufallsfunktionen typischen Ketten an Punkten und Stellen mit höherer Dichte, sowie leere Flecken. Wie in der Einleitung beschrieben, benötigen Simulationsalgorithmen aber eine homogene Verteilung, um genauer simulieren zu können. Um dies zu erreichen, kann, wie schon erwähnt, eine Goldene-Schnitt-Sequenz verwendet werden. Die selbe Funktion wie aus 2a ist auch in 2b verwendet worden, allerdings wurde eine Goldene-Schnitt-Sequenz verwendet. Diese Sequenz beschreibt auch zufällig verteilte Zahlen, allerdings werden die Punkte durch die Berechnungsvorschrift gleichmäßig verteilt. Dies hat auch sichtbare Auswirkkungen auf die Verformung, da auch nach der Verformung die Punkte noch gleichmäßig verteilt sind und keine Cluster entstehen. Offensichtlich kann ein Algorithmus diese Daten besser für Schätzberechnungen verwenden, als zufällig verteilte Daten.

Durch die Erzeugung der Zufallsvariable X durch das Inverse einer Verteilungsfunktion kann die Inversionsmethode allerdings nur dann optimal angewandt werden, wenn F^{-1} einfach zu berechnen ist oder sogar schon bekannt ist. Durch heutige numerische Algorithmen können aber auch Funktionen, welche nicht „per Hand“ invertierbar sind, in der vorgestellten Weise benutzt werden.

Nach Devroye können, nur unter Verwendung sogenannter „einfachen Transformationen“, Paare von unabhängigen Variablen generiert werden. Zum Beispiel kann mit der Standardnormalverteilung $\frac{e^{-x^2/2}}{\sqrt{2\pi}}$ ein Paar von unabhängigen, zufälligen standardnormalverteilten Variablen mit

$$(X,Y) = \left(\sqrt{\log(\frac{1}{U_1})}\cos(2\pi U_2),\, \sqrt{\log(\frac{1}{U_1})}\sin(2\pi U_2)\right)$$

generiert werden. Dabei sind $U_1, U_2 \in [0,1]$ unabhängige gleichverteile zufällige Variablen.

Bei einem Zufallsvektor $(X_1, ..., X_n) \in \mathbb{R}^n$ wird die Inversionsmethode auf je eine Dimension angewandt, wobei die Zufallszahl dieser Dimension generiert wird und dann rekursiv die nachfolgenden Dimensionen berechnet werden. Genauer bedeutet das, dass zuerst die Variable X_n der höchsten Dimension berechnet wird. Danach wird X_{n-1} mit der zugewiesenen Funktion berechnet, wobei X_n als bedingte Verteilung mit eingerechnet wird.

B. *Hashtabelle*

Eine Hashtabelle ist eine Datenstruktur, die mithilfe einer Hashfunktion $H : V \to K$ einen Wert $v \in V$ auf den zugehörigen Schlüssel $k \in K$ abbildet. Die Werte werden dann in einem Array gespeichert, wobei der Schlüssel die Position angibt. Da es in der Praxis meistens mehr Werte als Stellen im Array gibt, treten allerdings Kollisionen auf. Diese können entweder dadurch gelöst werden, dass das Array pro Schlüssel eine verkettete Liste enthält. In diese werden dann die kollidierende Werte hintereinander geschrieben. Die zweite verbreitete Möglichkeit ist lineares Sondieren. Dabei wird der Wert immer an die nächste freie Stelle in der Hashtabelle, von seinem eigentlichem Platz aus gesehen, geschrieben.

C. Inversion mit Hashing

Bei komplizierteren Inversen ist die normale Inversionsmethode nicht mehr effizient anwendbar. Um sie beschleunigen zu können wurden deshalb mehrere Möglichkeiten entwickelt. Eine davon ist es, eine Hashtabelle in der von Chen und Asau [1] erarbeiteten Vorgehensweise verwendet wird. Um diese Methode zu verwenden, wird für $j \in [1, n]$ die Wahrscheinlichkeit p_j als $p_j = \mathbb{P}(X = v_j)$ definiert. Mithilfe einer uniformen Variablen $U \in [0, 1]$ wird die Zufallsvariable X durch

$$F(v_{j-1}) < U \leq F(v_j) \tag{1}$$

berechnet, wobei $X = v_j$ ist. Diese Ungleichheit kann zwar ohne großen Aufwand berechnet werden, doch bei sehr großem n läuft jede einzelne Generierung trivial in $O(n)$. Mit dem Einsatz einer Hashtabelle, die wie eine Indextabelle agiert, wird für jede Generierung nur approximiert $O(1)$ Zeit benötigt. Dafür wird aus den Werten $F(v_j)$ der Wert

$$I_j = \lfloor F(v_j) * d \rfloor + 1 \tag{2}$$

berechnet. Dabei sollte $d = 10^m$ so gewählt werden, dass $\min\limits_{0 \leq j \leq n} (I_j) = 1$ ist. Danach wird die Indextabelle $T(k)$, $k \in [1, \max\limits_{0 \leq j \leq n} (I_j)]$ initialisiert. Dafür werden die I_j als Indexliste verwendet. Für jedes I_j wird $T(I_j)$ als das k gesetzt, für das $I(k) = I(j)$ ist. Falls ein Element I_k nicht in der Indexliste vorhanden ist, gilt $T(I_k) = T(I_{k+1})$.

Nach diesen Vorberechnungen und Initialisierungen kann die Hashtabelle verwendet werden, um eine Zufallszahl X zu berechnen. Dafür wird aus einer uniformen zufälligen Zahl U das zugehörige I_U (2) und $i = T(I_U)$ berechnet. i beschränkt damit die Teilmengen $F(v_j)$, die (1) erfüllen können. Die zu überprüfenden Teilmengen sind gegeben durch die Menge $\{F(v_i), \ldots, F(v_{r-1})\}$, wobei r die nächstgrößere Zahl ist, für die $I(i) \neq I(r)$ gilt. Für den Fall, dass $U > F(v_{r-1})$ ist, gilt bei Einsatz der Hashtabelle T immer $U \leq F(v_r)$, da ansonsten nicht $i = T(I_U)$ gelten würde. Nachdem ein v_j gefunden wurde, für das (1) erfüllt ist, wird X auf v_j gesetzt.

12 Jahre nach Chen und Asau's Ansatz wird in Devroye [2] eine abgewandelte Methode vorgestellt. Dabei wird eine Hashtabelle der Größe N erstellt, wobei N die Anzahl der möglichen Werte angibt. An der i-ten Stelle in der Hashtabelle wird der Wert von X mit $\mathbb{P}(X) = p_X$ gespeichert, falls $U = \frac{i}{N}$, $i \in [0, N)$ wäre. Nach dieser Initialiserung wird der Startindex Z, ab welchem eine Lösung zu finden ist, mit $Z = \lfloor N * U \rfloor$ berechnet. Falls man nun eine kummulative Verteilung vorliegen hat, oder eine Tabelle mit Partialsummen erstellt hat, kann jetzt mit minimalem Aufwand ab der Z-ten Partialsumme eine Lösung der bekannten Formel (1) von Chen und Asau gesucht werden.

III. Einschränkungen

Nachdem die Funktionsweise der Hash-basierten Inversionsmethode erklärt wurde, werden nun mögliche Einschränkungen der Methode näher betrachtet und, wenn möglich, Lösungsansätze aufgezeigt.

A. Dimension

Da in dem Abschnitt über die Funktionsweise der Inversionsmethode schon erklärt wurde, wie die Inversionsmethode in mehreren Dimensionen eingesetzt werden kann, ist leicht zu erkennen, dass mit etwas Mehraufwand auch die hash-basierte Methode in n Dimensionen verwendet werden kann. Da die Werte der einzelnen Dimensionen rekursiv und abhängig voneinander berechnet werden, reicht nicht mehr nur eine Hashtabelle aus. Für jede Dimension muss eine Hashtabelle erstellt werden. Aufgrund der Abhängigkeit muss die Hashtabelle der höchsten Dimension zuerst berechnet werden. Dies funktioniert nach dem bekannten Muster, aber für alle weiteren Dimensionen müssen nun andere Funktionen zum Populieren der Tabellen verwendet werden.

Anstelle der gegebenen Dichtefunktion F mit einer Verteilung $\mathbb{P}(X) = p_X$ ist die jeweilige Verteilung gegeben durch $\mathbb{P}(X|Y) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(Y)}$, wobei X der gesuchte Wert in der aktuellen Dimension ist und Y der Wert der höheren Dimension. An diesem Aufbau sieht man, dass die Hash-basierte Inversionsmethode zwar theoretisch für Punkt-Warping in beliebig vielen Dimensionen verwendet werden kann, doch bei zu vielen Dimensionen wird der Verwaltungsaufwand zu groß und die Initialisierung zu rechenaufwändig. Ab wievielen Dimensionen das der Fall ist, kann alllerdings nicht pauschal gesagt werden, da man mit kleineren Hashtabellen zwar ungenauer invertiert, aber weniger Rechenaufwand hat.

B. Abbildung

Da die Inversionsmethode nur sinnvoll einsetzbar ist, wenn es möglich ist, das Inverse einer Funktion zu berechnen, oder die Dichtefunktion direkt bekannt ist, ist auch die hashbasierte Inversion nur dann einsetzbar. Ansonsten erhält die Methode alle Eigenschaften ihrer Eingabefunktion. Dazu gehören Winkelerhalt, aber auch Korrelation und statistische Unabhängigkeit, da durch die Hashtabelle keine Eigenschaften verloren gehen, solange die Punkte nicht zu dicht beieinander liegen (siehe Dichte).

C. Dichte

Da durch den Einsatz der Hashtabelle nur endlich viele Werte gespeichert werden können, ist eine Berechnung von Zufallsvariablen X nur dann sinnvoll, wenn diese nicht zu dicht beieinander liegen und durch Annäherung auf einen Punkt gemappt werden würden, da die Tabelle eine zu geringe Größe hat, um ausreichend viele unterschiedliche Werte zu speichern. Deshalb sollte vor Einsatz der Tabelle die Dichte bekannt sein, damit die Größe N der Hashtabelle optimal gewählt werden kann.

IV. Vergleich

Chen und Asau haben in ihrer Arbeit über die hash-basierte Inversionsmethode gezeigt, dass ihre Methode für große Zahlen um eine Magnitude schneller ist, als triviale Methoden. Durch den Einsatz der Hashtabelle werden, anstelle von $O(n)$ Berechnungen wie bei der trivialsten Methode lineare Suche

Size	Method	Number of random variates generated					
		200	300	500	1000	3000	7000
5	Standard	0.0623	0.0914	0.1511	0.3062	0.9498	2.1846
	bisection	0.0849	0.1245	0.2152	0.4230	1.2352	3.0187
	proposed	0.0826	0.1233	0.2117	0.4296	1.1698	2.4856
10	Standard	0.0725	0.1151	0.1922	0.3626	1.0965	2.4887
	bisection	0.1043	0.1549	0.2616	0.5246	1.6378	3.5596
	proposed	0.0703	0.1108	0.1860	0.3499	1.0479	2.4198
20	Standard	0.1118	0.1672	0.2672	0.5415	1.6140	3.6730
	bisection	0.1248	0.1862	0.3117	0.6111	1.9478	4.3232
	proposed	0.0765	0.1279	0.2112	0.4288	1.3543	2.7897
50	Standard	0.1834	0.2959	0.5357	0.9973	3.1698	7.1161
	bisection	0.1549	0.2245	0.3745	0.7491	2.3863	5.3035
	proposed	0.0865	0.1288	0.2161	0.4401	1.4096	2.9702
100	Standard	0.3465	0.5196	0.8968	1.8139	5.4206	11.2695
	bisection	0.1737	0.2612	0.4399	0.8641	2.4420	6.2331
	proposed	0.0959	0.1469	0.2488	0.5282	1.5539	3.5197

Abbildung 3: Laufzeiten unterschiedlicher Algorithmen: Standard ((1) wird trivial berechnet), Bisection ((1) wird mit einfacher binarär Suche berechnet) und Proposed (die vorgestellte Methode mit Indextabelle). Auszug aus Chen und Asau [1].

oder $O(\log(n))$ Berechnungen bei binärer Suche, bei gut gewählter Größe der Hashtabelle nur noch $O(1)$ Berechnungen je Zufallsvariable durchgeführt. Diese Verbesserung hängt mit der Eigenschaft von Arrays und Hashtabellen zusammen, auf jedes Element in $O(1)$ zugreifen zu können. Deshalb kann die hashbasierte Inversionsmethode ihre Stärken erst bei vielen zu überprüfenden Intervallen, und damit Einträgen in der Tabelle, ausspielen. In Chen und Asau's Tests, deren Ergebnisse in der Tabelle 3 aufgezeigt sind, sieht man, das für wenige Intervalle (5) die Hashtabelle langsamer ist als triviales Testen der Ungleichungen. Mit 100 Einträgen ist die vorgestellte Methode aber schon mehr als 3-mal so schnell wie herkömmliche Verfahren. Das liegt daran, dass bei wenigen Intervallen die Initialisierung der Hashtabelle einen sog. Overhead erschafft, da der Aufwand, sie mit Werten zu befüllen und alle Möglichkeiten vorher zu berechnen, nicht durch viele Zugriffe wieder gutgemacht wird.

Wie in der Einleitung benannt, gibt es noch die Möglichkeit, Huffman-Bäume einzusetzen. Diese kodieren Werte nach ihrer Häufigkeit, wodurch häufig vorkommende Werte kürzer dargestellt werden und dadurch weniger Platz, verbrauchen. Diese Bäume können auch traversiert werden, wobei auch hier der kürzere Weg für häufigere Werte von Vorteil ist. Es ist möglich, einen solchen Baum abzuwandeln, damit er sinnvoll in der Partialsummenberechnung eingesetzt werden kann. Dadurch ergeben sich optimale Laufzeiten, die durch konstante Werte je nach Implementierung sogar besser als jene einer Hashtabelle sein können. In dieser Arbeit werden allerdings keine Vergleiche gezeigt, da die Implementierung eines Huffman-Baumes über den Rahmen dieser Ausarbeitung hinausgehen würde.

Durch die Hashtabelle werden bei sehr großen Mengen an Zufallsvariablen viele komplizierte Berechnungen gespart, da nur bei der Initialisierung die möglicherweise aufwändige Invertierung berechnet wird. Durch das Zusammenfallen von Werten durch nur endlich viele Einträge in der Hashtabelle, eignet sich die hashbasierte Inversion allerdings nur, wenn die Berechnung und Auswertung der Dichtefunktion sehr aufwändig ist und man nicht die Ressourcen oder Zeit besitzt, diese für jeden Wert auszuführen.

V. Implementierung

Um einen Eindruck der hashbasierten Inversionsmethode und ihrer Arbeitsweise zu bekommen, wird jetzt eine konkrete Implementierung dieser angesprochen und E rgebnisse bei unterschiedlichen Eingaben gezeigt. Geschrieben wurde das Programm in Python mithilfe der Pakete *numpy* und *matplotlib*.

Schretter [5] stellt am Ende der Arbeit eine C++-Methode zur Generierung einer zweidimensionalen Goldenen-Schnitt-Sequenz zur Verfügung. Diese Methode wurde in ein Python-Script übertragen und alle, in diesem Kapitel erwähnte, zufällige, uniformen Variablen U werden mit diesem Script generiert.

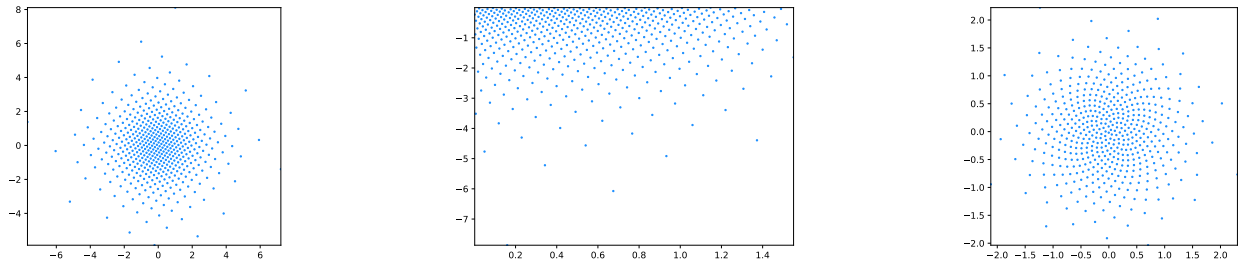
Zum Testen der vorgestellten Methode wurde das Konzept von Devroye [2] implementiert. Dabei wurde kein großer Wert auf optimale Laufzeiten gelegt, sondern darauf, eine Demonstration des Verfahrens geben zu können. Am Ende dieser Arbeit 1 ist in einer Mischung aus Python und Pseudocode der Kern der Implementierung zu sehen. Zusätzlich dazu ist der gesamte Algorithmus in diesem Git-Repositoryzu finden.

Die eigentliche Inversionsmethode mit Hashtabelle ist in `guidetable.py` implementiert. Dabei beschreiben die Methoden `__init__`, `hash` und `search_single` das Invertieren und Hashen.. In `__init__` wird die Hashtabelle mit ihren Werten populiert, wie es in dem Kapitel Inversion mit Hashing beschrieben wurde. Die Funktion `search_single` sucht ab dem gehashten Wert der eingegebenen Zufallsvariable solange in der Partialsummenliste, die davor entweder berechnet oder übergeben wurde, bis die Ungleichung zum Finden des zugehörigen X-Wertes (1) erfüllt ist. Damit nicht jeder Punkt einzeln berechnet werden muss, kann mit der Funktion `search` ein zweidimensionales Feld übergeben werden, für das dann die korrespondierenden Werte berechnet und zurückgegeben werden. Eine andere wichtige Datei ist `golden_ratio_sequence.py`. Diese Datei fasst Methoden zur Generierung von Goldenen-Schnitt-Sequenzen zusammen und wurde von Schretter übernommen und nach Python übertragen.

In 4 sind drei Punktemengen zu sehen, bei denen eine Goldene-Schnitt-Sequenz in der vorgestellten Implemtierung generiert und dann mit `guidetable.py` verformt wurden. Alle drei Abbildungen zeigen 500 Punkte, wobei einmal 4a auf beiden Achsen eine logistische Dichte I angewendet wurde. Im Gegensatz dazu ist bei 4b die X-Koordinate mit einer Sinusfunktion verschoben und die Y-Koordinate nach einer exponentiellen Verteilung I. Man kann leicht die sinusförmigen Wellen sehen, bei denen die Punkte enger aneinander liegen. In der dritten Abbildung 4c sind beide Koordinatenanteile standardnormalverteiltverformt, wodurch das bekannte Spiralmuster entsteht.

VI. Zusammenfassung

Mit der vorgestellten Methode können vor allem große Datenmengen sehr schnell generiert werden. Solange die Daten nicht zu dicht beieinander liegen und die Größe der Hashtabelle gut genug gewählt ist, werden sogar alle Eigenschaften der



(a) Logistische Dichte auf der X- und Y-Achse

(b) Sinusodial auf der X- und exponentiell auf der Y-Achse verteilt

(c) Standardnormalverteilung in beide Richtungen

Abbildung 4: Mehrere inverse Funktionen mit je 500 Punkten.

Funktion behalten. Dadurch kann man die hashbasierte Inversiosmethode mit Modellen zur uniforem Generierung von Zufallszahlen, wie zum Beispiel Golden Ratio Sequences [5] oder Fibonnaci Grids [3], verbinden. Je nach gewünschter Genauigkeit ist der Verwaltungsaufwand und der Speicherverbrauch in höheren Dimensionen auch nicht annehmbar, wenn man genügend Kapazitäten zur Verfügung hat. Falls dies der Fall ist, benötigt man, nach einer anfänglichen Initialisierungsphase nur $O(1)$ Zeit, um Zufallszahlen zu generiern. Darum kann die Hash-basierte Inversionsmethode beinahe überall eingesetzt werden. Da die vorgestellte Methode auf einem einfachen Prinzip beruht und dieses in dieser Ausarbeitung ausführlich erläutert wird, kann jeder eine naive Implementierung der hash-basierten Inversionsmethode selbst verwenden.

A. *Verbesserungsmöglichkeiten*

Wie in dem Abschnitt Dichte angesprochen, liefert die Methode nur noch suboptimale Ergebnisse, wenn die Punkte an wenigen Stellen sehr dicht beieinander liegen, ansonsten aber weiter voneinander entfernt. Aus Gründen der Optimierung wird dann meist eine kleinere Hashtabelle gewählt, wodurch diese eng beianderliegenden Punkte zusammenfallen. Dieses Problem könnte man dadurch lösen, dass in der Hashtabelle die Punkte nicht mehr gleichmäßig verteilt eingesetzt werden, sondern an Stellen mit höherer Dichte mehr Einträge vorhanden sind. Diese Anpassungen kann die Initialisierung der Hashtabelle allerdings sehr viel langsamer oder größer machen, weshalb in solchen Fällen abgewogen werden muss, welcher Aspekt für das gegebenere Problem wichtiger ist.

LITERATUR

[1] Hui-Chuan Chen and Yoshinori Asau. On Generating Random Variates from an Empirical Distribution. *A I I E Transactions*, 6(2):163–166, 1974.

[2] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer New York, 1986.

[3] Daniel Frisch and Uwe D. Hanebeck. Deterministic gaussian sampling with generalized fibonacci grids. In *Proceedings of the 24th International Conference on Information Fusion (Fusion 2021)*, South Africa, November 2021.

[4] Gopal Prasad. Strong rigidity of Q-rank 1 lattices. *Inventiones mathematicae*, 21(4):255–286, 1973.

[5] Colas Schretter, Leif Kobbelt, and Paul-Olivier Dehaye. Golden Ratio Sequences for Low-Discrepancy Sampling. *Journal of Graphics Tools*, 16:9, 06 2012.

[6] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with Hammersley and Halton Points. *Journal of Graphics Tools*, 2(2):9–24, 1997.

[7] Dong-Ming Yan, Jian-Wei Guo, Bin Wang, Xiao-Peng Zhang, and Peter Wonka. A survey of blue-noise sampling and its applications. *Journal of Computer Science and Technology*, 30(3):439–452, 2015.

Algorithm 1 Implementation of the hash-based inversion method (Part 1)

Input
N Size of the hash table
f_inv inverse function for this table
function __INIT__($self$, N, f_inv)
$self.T \leftarrow [\,]$ ▷ hash table
$self.PS \leftarrow$ np.array($[\,]$) ▷ list of partial sums
$self.N \leftarrow$ N
$self.f\_inv \leftarrow$
for i in range$(0, self.N)$ **do**
$self.T$.append$(self.f_inv(\frac{i}{self.N}))$
end for
$self.T \leftarrow$ np.array$(self.T)$
end function

Input
f inverse function of $self.f\_inv$
max amount of partial sums
function SETUP_PARTIAL_SUMS_UNIFORM($self$, f, max)
$ps \leftarrow [\,]$
$i \leftarrow 0.0$
$step \leftarrow \frac{\text{max}}{self.N}$
while $i <$ max **do**
if $i == 0.0$ **then**
ps.append(f(0))
else
ps.append$(ps[-1] +$ f$(i))$
end if
$i \leftarrow i + step$
end while
$self.PS \leftarrow$ np.array(ps)
end function

Algorithm 2 Implementation of the hash-based inversion method (Part 2)

Input
U random variate
Output
h hashed value of U
function HASH($self$, U)
return $\lfloor self.N * \mathrm{U} \rfloor$
end function

Input
U random variate
Output
X from hash table calculated value corresponding to U, so that $\mathrm{f}^{-1}(U) = X$
function SEARCH_SINGLE($self$, U)
$Z \leftarrow self.hash(\mathrm{U})$
while $self.PS[Z] \leq \mathrm{U}$ **do** $Z \leftarrow Z + 1$
end while
return $self.T[Z]$
end function