HSR Hochschule für Technik

Studienarbeit

# Optimierung und Überprüfung auf Korrektheit und Parallelisierbarkeit eines Clustering-Algorithmus

*Autoren:*
Johnathan Stolz
Thomas Jutzi

*Betreuer:*
Prof. Dr. Ruedi Stoop

31. Mai 2012

# Abstract

Zur Erkennung von Clustern gibt es heute unterschiedliche Algorithmen. Die gängigsten z.B. k-Means [1], Ward [1] eignen sich nicht, um komplexere Formen zu erkennen. Das Hebbian Learning Clustering (HLC) Verfahren [1] [2] ist ein neues mächtigeres Verfahren, welches komplexe Formen problemlos erkennen kann.
Das Ziel unserer Studienarbeit war es, den HLC-Algorithmus von Mathematica nach C/C++ zu portieren und seine Korrektheit zu überprüfen. Diese Portierung sollte anschliessend optimiert werden. Mit der Lösung sollten grosse Vorgaben möglichst schnell geclustert werden können.
Wir portierten den HLC-Algorithmus von einer Mathematica Vorlage nach C/C++. Unser Hauptaugenmerk haben wir auf die Optimierung unserer Portierung gelegt.
Als Anwendung wurde die Portierung verwendet, um Phoneme aus der TIMIT [3] Datenbank zu clustern. Gefundene Cluster wurden auf shrimp-ähnliche Formen untersucht, welche sich aufgrund theoretischer Voraussage darin finden sollten.
Der HLC in der Ausgangslage benötigte zu viel Zeit, um grössere Vorgaben zu clustern. Mit der optimierten Portierung sind wir nun in der Lage auch komplexere Vorgaben effizient zu clustern.

# Danksagung

Für die Unterstützung während der Studienarbeit möchten wir folgenden Personen einen besonderen Dank ausrichten:

**Prof. Dr. Ruedi Stoop** für seine Unterstützung und sein wertvolles, konstruktives und motivierendes Feedback.

**Prof. Dr. Josef Joller** für die unterstützenden Inputs während der Arbeit sowie die Hilfe bei der O-Notation.

**Silvia Stolz** und **Michèle Fröhlich** für das Korrekturlesen der Studienarbeit.

**Rolf Schönenberger** und **Claude Baumann** für die anregenden Diskussionen während der Arbeit.

# Inhaltsverzeichnis

# 1

# Einführung

In diesem Kapitel beschreiben wir zuerst die Aufgabenstellung. Als Zweites betrachten wir die Ausgangslage für unsere Studienarbeit genauer. Wir zeigen die Mächtigkeit des HLC-Algorithmus anhand von Beispielen auf. Als Letztes beschreiben wir die von uns selbstdefinierten Ziele.

## 1.1 Aufgabenstellung

Zur Erkennung von Clustern gibt es eine Vielzahl von Algorithmen. Unter diesen Algorithmen sticht der HLC durch seine Fähigkeit hervor, komplexe Gebilde selbstständig zu erkennen. Um grosse Datenmengen schnell mit dem HLC zu Clustern, sollte dieser Algorithmus in unserer Studierarbeit optimiert werden.

## 1.2 Ausgangslage

Der Cluster-Algorithmus HLC [2] ist ein mächtiger Algorithmus, welcher es ermöglicht, vorgegebene Objekte autonom in verschiedene Klassen von Objekten einzuteilen, die gemeinsame Eigenschaften aufweisen. Im Gegensatz zu klassischen Verfahren, wie z. B. k-Means, ist es nicht notwendig, dem Algorithmus anzugeben, nach wie vielen Clustern gesucht werden soll, da diese selbständig gefunden werden. Des Weiteren sind klassische Verfahren für rauschbehaftete Daten oftmals ungeeignet, da diese versuchen, sämtliche Datenpunkte zu clustern, was oftmals unerwünscht ist, weil sich dadurch die Qualität des Ergebnisses erheblich verschlechtert.
Für unsere Arbeit stand eine komplette Implementation des HLC-Algorithmus in Mathematica zur Verfügung.

### 1.2.1 Vorteile des HLC

Die Vorteile des HLC zeigen sich schnell, wenn man einfach Datenmengen clustert. In Abbildung 1.1 wird jeweils eine 2D- und 3D-Vorgabe geclustert. Die Unterschiede bei den Clustering-Algorithmen sind klar ersichtlich. Die Ergebnisse des HLC gegenüber dem k-Means sind unübersehbar besser. Die Auseinanderhaltung der zwei 3D-Kleeblattknoten erfolgt im HLC einwandfrei, wo hingegen der k-Means das Objekt nur halbiert.
Sogar die Augen beim 2D-Smiley werden korrekt erkannt (Trennung in Pupille und Iris), was beim k-Means nicht funktioniert. Auch sind die Rauschpunkte beim HLC-Ergebnis verschwunden.

## 1.3 Ziele

Das Ziel dieser Arbeit ist es, den HLC-Algorithmus von Mathematica nach C/C++ zu portieren. Der bestehende Ablauf der Portierung soll analysiert werden, damit der Code und der von ihm verursachte Aufwand auf ein Minimum reduziert wird. Diese optimierte Version gilt es sodann, auf Möglichkeiten zur Parallelisierung zu untersuchen. Die Vor- und Nachteile des Nvidia GPGPU-Frameworks CUDA [4] [5] sollen denen des normalem C/C++ Threading gegenübergestellt werden. Die optimalere Methode wird anschliessend implementiert.

Mit dieser optimierten Portierung sollen Phoneme in den TIMIT Daten schneller geclustert werden können. Die gefundenen Cluster werden auf shrimp-ähnliche Formen hin untersucht. Dies würde aufzeigen, dass die Phoneme keine Gaussschen Wolken darstellen und es deshalb natürlich ist, dass klassische Clusteringverfahren Phoneme schlecht unterscheiden können.

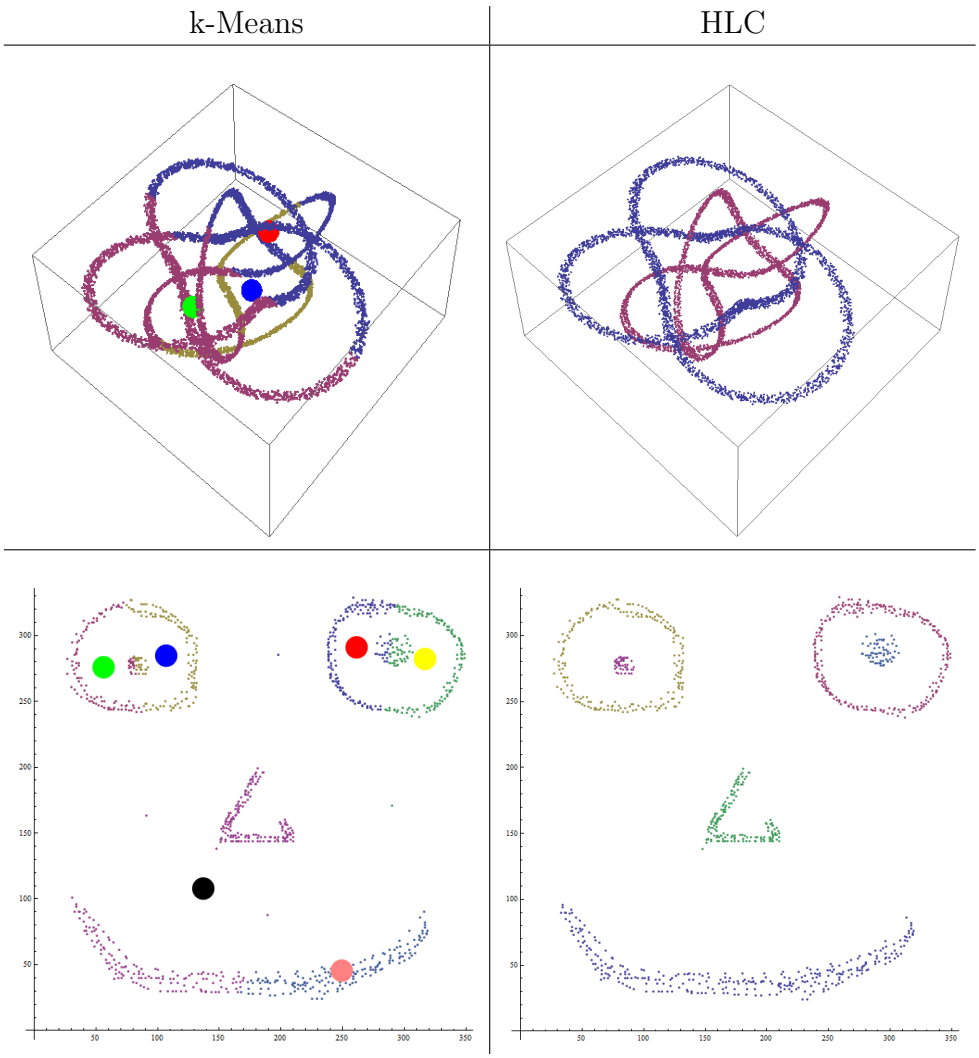


Abbildung 1.1: Unterschiede beim Clustering: Links Ergebnisse mittels k-Means; Rechts die gleiche Vorgabe mittels HLC. Klassische Verfahren wie k-Means gehen von Gaussschen Clusterformen aus (welche sie perfekt trennen). Die obigen Datenmengen sind aber konvex-konkav begrenzt, also nicht-Gauss'sch.

# 2

# Umsetzung der Portierung

Dieses Kapitel trennen wir in zwei Teile auf. Im ersten Teil beschreiben wir verschiedene Hürden und deren anschliessenden Lösungen, die bei der direkten Portierung aus der Mathematica Vorlage nach C/C++ aufgetaucht sind. Im zweiten Teil erklären wir die verschiedenen Optimierungen an unserer Portierung genauer. Dieser Teil baut auf unseren Erkenntnissen aus der Portierung auf.

## 2.1 Hürden

### 2.1.1 Numerische Sortierung

Um die k-nächsten Nachbarn von einem Punkt zu finden, wurde die Mathematica Funktion Ordering [6] verwendet. Diese Funktion gibt eine neue Liste mit Indizes auf die originale Liste zurück. Diese Indizes ergeben beim Auslesen die sortierte Liste.

Ein Beispiel:

| Liste L | Ordering[L] |
|---|---|
| 1, 7, 9, 2 | 1, 4, 2, 3 |

Das Auslesen erfolgt über die Indizes von Ordering:
L[1],L[4],L[2],L[3] = 1, 2, 7, 9

Dabei ist uns aufgefallen, dass diese Funktion mit Brüchen nicht gleich umgeht wie mit numerischen Zahlen. Die Zahlen wurden nicht nach deren Wert sortiert, was zu fehlerhaften Resultaten führte.
Die folgende Tabelle zeigt das Ordering auf die gleiche Liste angewendet, einmal mit Brüchen und einmal numerisch:

| Darstellung | Liste | Ordering | Ausgelesen |
|---|---|---|---|
| Bruch | $\{4\sqrt{2}, 2\sqrt{2}, 0, 2\sqrt{5}\}$ | 3, 2, 1, 4 | $\{0, 2\sqrt{2}, 4\sqrt{2}, 2\sqrt{5}\}$ |
| Numerisch | {5.65, 2.82, 0, 4.47} | 3, 2, 4, 1 | {0, 2.82, 4.47, 5.65} |

An diesem Beispiel ist klar zu erkennen, dass die zwei letzten Stellen beim Resultat der Brüche nicht korrekt sortiert wurden.
Da dies eindeutig zu fehlerhaften Resultaten führt, wurde in der Portierung immer das numerische Format verwendet.

Die Abbildung 2.1 zeigt, welche Resultate das Clustering mit und ohne korrekt formatierte Zahlen liefert. Die Iris wurde bei der korrekt formatierten Sortierung erkannt.

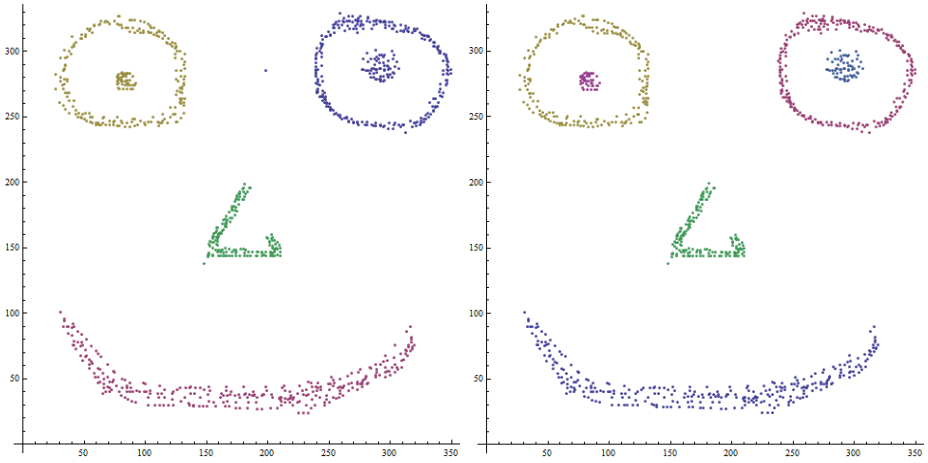

Abbildung 2.1: Sortierung: Links mit Brüchen (4 Cluster); Rechts numerisch (6 Cluster).

### 2.1.2 Präzision

Mathematica verwendet die GMP Library [7] [8], um grosse Zahlen mit hoher Genauigkeit zu verarbeiten. C/C++ verwendet bei float/double eine exponentielle und gerundete Repräsentation von nachkommabehafteten Zahlen. Dies führt zu Problemen beim Test auf Gleichheit:

```
if (number1 == number2)
```

Dieser Vergleich auf Gleichheit führt in C/C++ nicht zum gleichen Resultat wie in Mathematica. Wir haben festgestellt, dass der HLC-Algorithmus schon

mit wenigen Nachkommastellen gute Resultate liefert. Darum wird die Vergleichsoperation nur auf wenige Nachkommastellen beschränkt. Somit kann mit den herkömmlichen Datentypen von C/C++ gearbeitet werden, und es muss keine zusätzliche Abhängigkeit zu einer anderen Library eingegangen werden. Dies würde auch eine allfällige Portierung nach CUDA oder anderen hardwarenahen Lösungen vereinfachen.
Um zwei Zahlen auf Gleichheit zu prüfen, haben wir den Vergleich auf fünf Nachkommastellen beschränkt. Dazu verwenden wir das folgende Konstrukt:

```
if (fabs(number1-number2) < 0.00001)
```

### 2.1.3 Matrix Umspiegelung

Für die Erstellung der $k$ nächsten Nachbarschaften wurde in der Mathematica Vorlage eine Matrix der Grösse $N * N$ ($N$=Anzahl Datenpunkte) verwendet. Um die $k$ nächsten Nachbarn für einen Punkt einzutragen, wurden diese nicht nur horizontal abgelegt, sondern auch vertikal umgespiegelt. Dies führte zu nicht korrekten $k$ nächsten Nachbarschaftsbeziehungen.
Am folgenden Beispiel kann dies einfach erklärt werden. In Abbildung 2.2 werden die zwei nächsten Nachbarn für P1 (grün markierte Zeile) gesucht. Diese werden als Nachbarn für P1 markiert ($X_1$) und auch gerade gespiegelt eingetragen ($Y_1$).

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| P1 | | | $X_1$ | $X_1$ | |
| P2 | | | | | |
| P3 | $Y_1$ | | | | |
| P4 | $Y_1$ | | | | |
| P5 | | | | | |

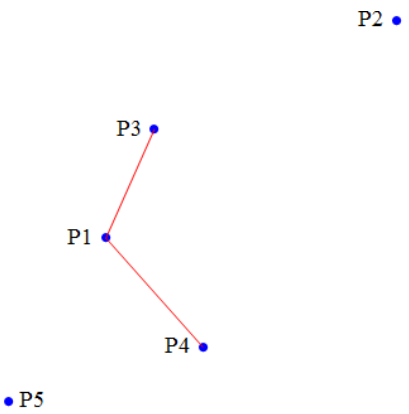


Abbildung 2.2: Schritt 1: Die zwei nächsten Nachbarn für P1 werden eingetragen.

In Abbildung 2.3 werden die nächsten zwei Nachbarn für P2 gesucht. In der Zeile P2 werden die gefundenen Nachbarn mit $X_2$ markiert.

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| P1 | | | $X_1$ | $X_1$ | |
| P2 | $X_2$ | | $X_2$ | | |
| P3 | $Y_1$ | | | | |
| P4 | $Y_1$ | | | | |
| P5 | | | | | |

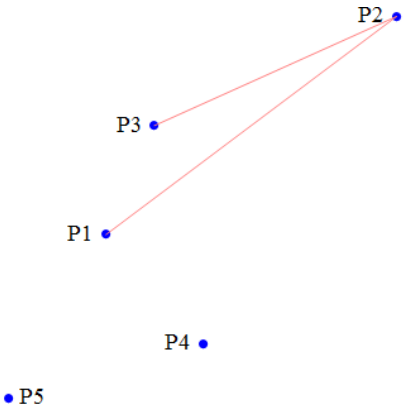


Abbildung 2.3: Schritt 2: Die zwei nächsten Nachbarn für P2 werden eingetragen.

Abbildung 2.4 zeigt wie die Punkte $X_2$ gespiegelt werden. Dies führt dazu, dass bei P1 und P3 der Punkt P2 als Nachbar markiert wird ($Y_2$).

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| P1 | | $Y_2$ | $X_1$ | $X_1$ | |
| P2 | $X_2$ | | $X_2$ | | |
| P3 | $Y_1$ | $Y_2$ | | | |
| P4 | $Y_1$ | | | | |
| P5 | | | | | |

Abbildung 2.4: Schritt 3: Die Nachbarn von P2 werden umgespiegelt.

Durch das Spiegeln der Nachbarschaftsmatrix wird nun P2 ein Nachbar von P1, obwohl der Punkt P5 für P1 näher gewesen wäre. Der fehlerhafte Punkt $Y_2$ ist in Abbildung 2.5 rot eingefärbt. Dieser Extrapunkt bedeutet letztlich, dass für Punkt P1 mehr als k Nachbarn definiert werden.

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| P1 | | $Y_2$ | $X_1$ | $X_1$ | |
| P2 | $X_2$ | | $X_2$ | | |
| P3 | $Y_1$ | $Y_2$ | | | |
| P4 | $Y_1$ | | | | |
| P5 | | | | | |

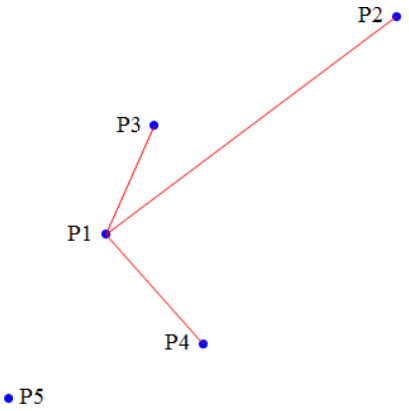


Abbildung 2.5: Das Ergebnis der Umspiegelung: P1 hat nun drei Nachbarn.

Dieses Beispiel zeigt auf, dass die Matrix mit den Nachbarn nicht gespiegelt werden darf. Daher kann die Matrix auch nur immer von einer Dimension aus gelesen werden, da es eine gerichtete Verbindung darstellt.
Durch diese Korrektur konnte der HLC neu auch die komplexe Vorgabe aus Abbildung 2.6 richtig erkennen.

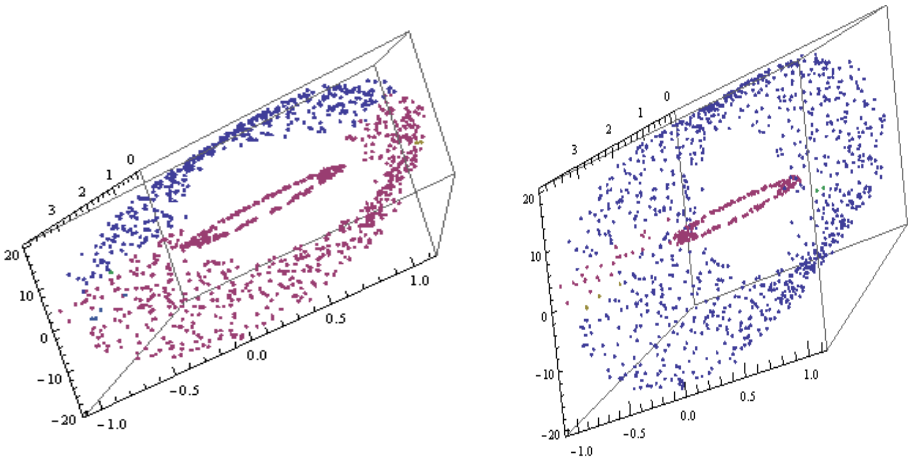

Abbildung 2.6: Ergebnis: Links mit umspiegeln; Rechts ohne umspiegeln.

Dadurch, dass die Nachbarschaftsmatrix nicht mehr gespiegelt wird, kann auch deren Dimension von $N * N$ auf $N * k$ verringert werden.

## 2.2 Optimierung

### 2.2.1 Kompaktierung von Matrizen

Die Berechnung der Matrizen für die Distanz, SynConnection (synaptische Stärke zwischen einzelnen Neuronen), Nachbarschaften sowie StrongComponents wurde in der Ausgangslage jeweils separat berechnet. Alle diese Matrizen waren von der Grösse $N*N$, wobei $N$ die Anzahl der zu clusternden Datenpunkte darstellt. Nach eingehender Analsyse konnte folgendes festgestellt werden: Die Distanz-Matrix wurde nur verwendet, um die SynConnection-Matrix zu füllen. Dazu wurden alle Distanzen von einem Datenpunkt $i$ zu allen anderen Datenpunkten sortiert, um dann die $k$ nächsten Nachbarn zu bestimmen. Wichtig hierbei ist, dass der eigene Punkt $(i, i)$ ausgeschlossen wird, da dieser zu sich selbst immer eine Distanz von 0 aufweist. Für diese $k$ nächsten Nachbarn wurde in der Nachbarschafts-Matrix dann markiert, um welche Datenpunkte es sich handelt. Danach wurde die Nachbarschafts- sowie Distanzmatrix nicht mehr benötigt.

Um den ganzen Prozess zu optimieren, werden die Matrizen zu einer einzigen, kompakten Matrix zusammengefasst. Diese Matrix hat nicht wie in der Ausgangslage eine Grösse von $N*N$, sondern nur noch $N*k$ (Erklärung dazu ist in Kapitel 2.1.3). Das schont den Speicherverbrauch und verbessert auch die Durchlaufzeit nachfolgender Berechnungen (namentlich der Integration-Loop) massgebend. Weiter kann die Nachbarschaftsmatrix komplett weggelassen werden, da diese nur temporär von Bedeutung war. Dafür muss ein neues Feld für den Zugriff auf die entsprechenden Punkte eingeführt werden, da dies vorher über den Index in der $N * N$-Matrix erfolgte.
Die initiale Berechnung der kompakten Matrix erfolgt nun komplett in einer einzigen Schleife. Der abstrakte Ablauf sieht wie folgt aus:

```
for(first k points in pointList)
{
   kNearestList.addLast(point)
}
SortByDistanceAscending(kNearestList)
currentMaxDistanceValue = kNearestList.Last.DistanceValue
while(pointList has more elements)
{
   currPoint = pointList.Next
   if(currPoint.distanceOfElement < currentMaxDistanceValue)
   {
      ReplaceLast(kNearestList, currPoint)
      SortByDistanceAscending(kNearestList)
      currentMaxDistanceValue = kNearestList.Last.DistanceValue
   }
}
```

### 2.2.2 Min-/Max anstelle von Ordering

Im Integration-Loop aus der Mathematica Vorlage wurde an mehreren Stellen die Funktion Ordering aufgerufen. Diese Funktion ist ziemlich aufwendig, weil intern die ganzen Daten sortiert und dann für die Originaldaten die Indizes gemäss der Sortierung berechnet werden müssen. Dieser Prozess hat typischerweise ein Worst-Case Verhalten von $O(N^2)$. Da aber eigentlich davon entweder das Minimum oder das Maximum gebraucht wird, ist es sinnvoller dafür eine entsprechende Min-/Max-Funktion zu verwenden. Diese haben typischerweise ein Worst-Case Verhalten von $O(N)$ was eine deutliche Steigerung bedeutet. Deshalb haben wir sämtliche Aufrufe von Ordering durch entsprechende Min-/Max-Funkionen ersetzt.

### 2.2.3 Leaky Bucket und Hebbian-Learning

Bei Vorgaben mit vielen Datenpunkten haben wir festgestellt, dass der Datentyp float/double im Laufe des Integration-Loop den Wertebereich übersteigt. Dies passierte, weil die SynConnection verdoppelt wurde ohne diese Werte zu begrenzen [2]. Nun wurde eine Begrenzung eingebaut, welche die Werte auf maximal 1.1 begrenzt.

Einen weiteren Punkt den wir beobachten konnten, war, dass sich bei rauschbehafteten Vorgaben mit hoher Punktezahl, die Objekte, die sich im Rauschen versteckten, sich nicht mehr erkennen liessen. Das Ergebnis war ein

einziger Riesen-Cluster. Wir konnten zusammen mit Herrn Prof. Dr. Stoop das fehlende Codefragment im Algorithmus ausfindig machen und erfolgreich einbauen [9]. Nun können auch Objekte in grösseren Vorgaben korrekt erkannt werden. Getestet haben wir dies mit zwei ineinander verhängten Kreisen, welche aus jeweils 24'000 Punkten bestehen, sowie von 8'000 Punkten Rauschen umgeben sind. Die Abbildung 2.7 zeigt die eindrücklichen Ergebnisse.

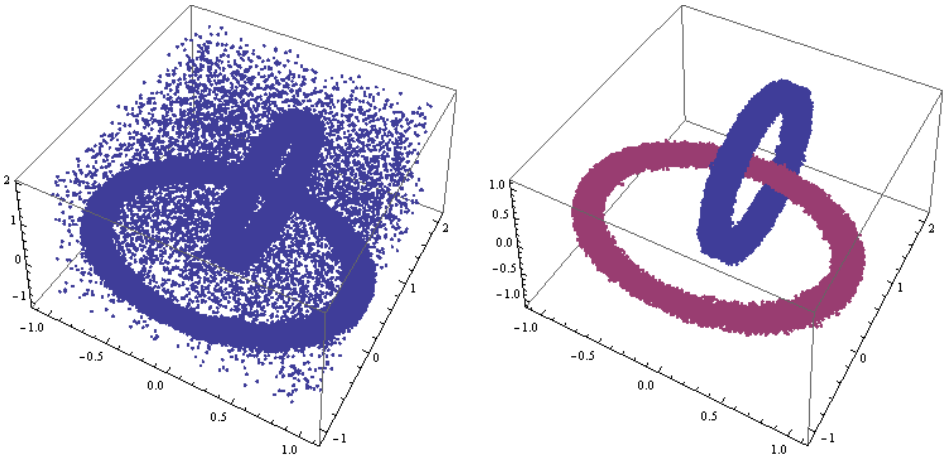

Abbildung 2.7: Leaky Bucket: Links verrauschte Vorgabe mit total 56'000 Punkten; Rechts das Ergebnis nach HLC mit eingebautem Leaky Bucket.

### 2.2.4 Extraktion der Cluster

Bei vielen Punkten benötigt die Extraktion der Cluster in der Mathematica Vorgabe mehr Zeit als der Main-Loop. Aus diesem Grund haben wir die Extraktion durch eine komplette Neuentwicklung ersetzt. Dazu haben wir uns überlegt, wie man die Cluster möglichst schnell auslesen kann.
Das Ergebnis ist ein Algorithmus, welcher sehr viel schneller durchläuft als das Original. Das Laufzeitverhalten mittels O-Notation wird in Kapitel 3.3 erörtert. Die Extraktion ist gemäss folgendem Pseudocode umgesetzt:

```
foreach(point in points)
{
   if(point.IsNotClustered)
   {
      combinedPointsList.addLast(point)
      while(combinedPointsList has more elements)
      {
         newPoint = combinedPointsList.removeFront()
         newPoint.IsClustered = true
         actualCluster.add(newPoint)
         foreach (neighbourPoint in newPoint.NeighbourPoints)
         {
            if(neighbourPoint.IsStrongComponent &&
               neighbourPoint.IsNotClustered &&
               actualCluster.ContainsNot(neighbourPoint) &&
               combinedPointsList.ContainsNot(neighbourPoint))
             {
                  combinedPointsList.addLast(neighbourPoint)
             }
         }
      }
      allClusters.add(actualCluster)
   }
}
```

### 2.2.5 Parallelisierung

Die initiale Berechnung der Distanzen ist voll parallelisierbar. Dabei kann jeder Datenpunkt unabhängig von den anderen berechnet werden. Diese Berechnung haben wir zu Testzwecken mit CUDA und C/C++-Threads implementiert. Jedoch stellte sich heraus, dass CUDA für dieses Problem eine weniger geeignete Lösung ist. Die folgenden Nachteile wurden bei CUDA erkannt:

- Die Grafikkarte benötigt mindestens CUDA Version 1.3, um mit doppelter Genauigkeit rechnen zu können [10]
- Eine einfache Desktopgrafikkarte bietet zu wenig Performance, um diese Berechnungen schneller als mit der CPU durchzuführen. Es müsste eine sehr leistungsstarke GPGPU-Karte wie eine Tesla M2090 beigezogen werden, um schneller als die CPU zu werden

- Um die Berechnungen auf der Grafikkarte durchzuführen, müssen zuerst alle Daten vom Hauptspeicher auf die GPU kopiert werden und anschliessend wieder zurück. Dieser Kopiervorgang benötigt relativ viel Zeit, sodass sich eine Parallelisierung erst ab einer entsprechend langen Berechnung lohnt

- Ein einzelner GPU-Core ist wesentlich langsamer als ein CPU-Core. Deshalb lohnt es sich nicht die parallel gerechneten Daten auf der GPU zusammenzuführen (Divide and Conquer). Zudem fehlen dafür geeignete Kontrollstrukturen auf der GPU. Dies führt dazu, diesen Prozess auf der CPU durchzuführen, was eigentlich sämtlichen Geschwindigkeitsvorteil wieder vernichtet

Diese oben genannten Nachteile bei CUDA führten dazu, dass wir diese Berechnung entgültig mittels klassischen Threads implementierten.

# 3

# Performance-Messungen

Damit die Resultate der Portierung und anschliessenden Optimierung vergleichbar werden, haben wir die Laufzeiten der verschiedenen Versionen gemessen. Diese Messungen und deren Auswertungen betrachten wir in diesem Kapitel. Im letzen Abschnitt gehen wir noch auf das O-Verhalten einiger Stellen unserer Implementierung ein.

## 3.1 Vorgehen

Um die Performance der verschiedenen Implementierungsstufen zu vergleichen, haben wir mehrere gleiche Vorgaben verschiedener Grösse mit jeweils 15% Rauschen generiert. Jede Messung beinhaltet Zeitmessungen über die Bereiche Distanz-Berechnung, Integration-Loop und Cluster-Extrahierung. Dies ermöglicht eine präzise Aussage über das interne Laufzeitverhalten.
Die Parameter wurden so gewählt, dass das Ergebnis jeweils dem erwarteten Resultat entspricht. Die Abbildung 3.1 zeigt die Vorgabe sowie das erwartete Ergebnis.

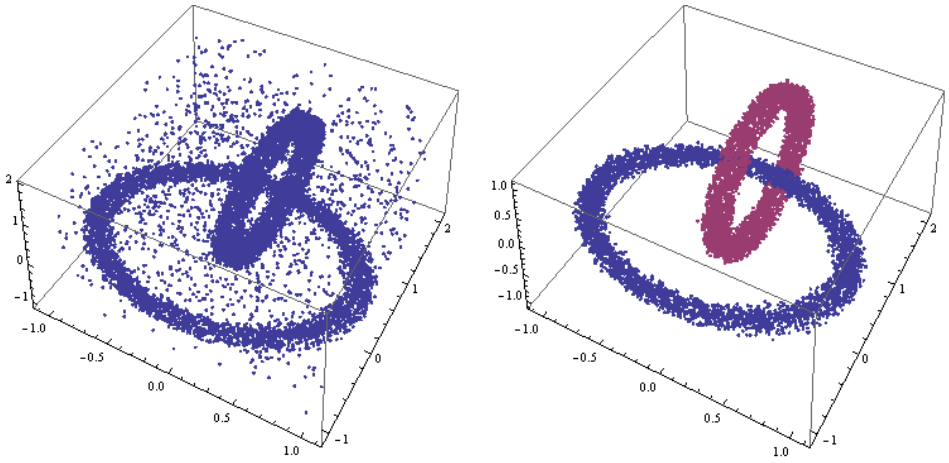

Abbildung 3.1: Performance-Messung: Links verrauschte Vorgabe; Rechts das erwartete Resultat.

Die Messungen haben wir mit den vier Implementationen aus Abbildung 3.2 durchgeführt:

| Version | Beschreibung |
|---|---|
| A | Originale, unveränderte Version in Mathematica |
| B | Abgeänderte Version in Mathematica<br>Gemäss Änderungen von Kapitel 2.1 |
| C | Abgeänderte Version in Mathematica<br>Gemäss Änderungen aus Kapitel 2.1 sowie 2.2.3 |
| D | C/C++-Version<br>Mit allen Änderungen aus Kapitel 2 |

Abbildung 3.2: Implementierungen: Die vier verglichenen Versionen des HLC. Version D stellt unsere finale Portierung dar.

## 3.2 Ergebnisse

In diesem Teil zeigen wir die Messungen für die Versionen aus Abbildung 3.2 auf. Diese Daten sind in Abbildung 3.3 aufgelistet und dienen als Grundlage für die nachfolgenden grafischen Darstellungen.

| Anzahl Punkte | A[s] | B[s] | C[s]* | D[s] | Faktor: $\frac{A}{D}$ | Faktor: $\frac{B}{D}$ |
|---|---|---|---|---|---|---|
| 200 | 5.01 | 13.57 | 114.82 | 0.30 | 16.76 | 45.38 |
| 400 | 21.50 | 45.38 | 1'377.86 | 0.23 | 91.95 | 194.13 |
| 600 | 46.74 | 105.44 | 6'009.63 | 0.41 | 113.98 | 257.13 |
| 800 | 84.76 | 194.94 | – | 0.38 | 221.47 | 509.39 |
| 1000 | 136.64 | 236.82 | – | 0.57 | 241.81 | 419.10 |
| 1200 | 214.99 | 368.93 | – | 0.65 | 331.62 | 569.08 |
| 1400 | 323.05 | 465.55 | – | 0.80 | 402.58 | 580.17 |
| 1600 | 432.65 | 586.08 | – | 1.37 | 315.80 | 427.79 |
| 1800 | 577.05 | 874.56 | – | 2.49 | 231.32 | 350.58 |
| 2000 | 779.21 | 1'853.25 | – | 3.38 | 230.44 | 548.07 |
| 2500 | 1'401.08 | 3'280.24 | – | 4.83 | 289.95 | 678.84 |
| 3000 | 2'374.49 | 4'395.24 | – | 9.63 | 246.55 | 456.38 |
| 3500 | 3'469.02 | 9'419.22 | – | 10.44 | 332.40 | 902.56 |

Abbildung 3.3: Performance-Messung: Tabelle mit Messwerten für jede Version. (*) Messung nur bis 600 Punkte durchgeführt, da der Vorgang für 800 Punkte zu lange brauchte.

### 3.2.1 Laufzeitverhalten aller Versionen

In diesem Plot werden die Laufzeiten der verschiedenen Versionen gegenübergestellt. Die Version D ist um einen grossen Faktor schneller als die anderen Versionen. Eine detailliertere Darstellung dieser Version ist in Abbildung 3.5 zu sehen.

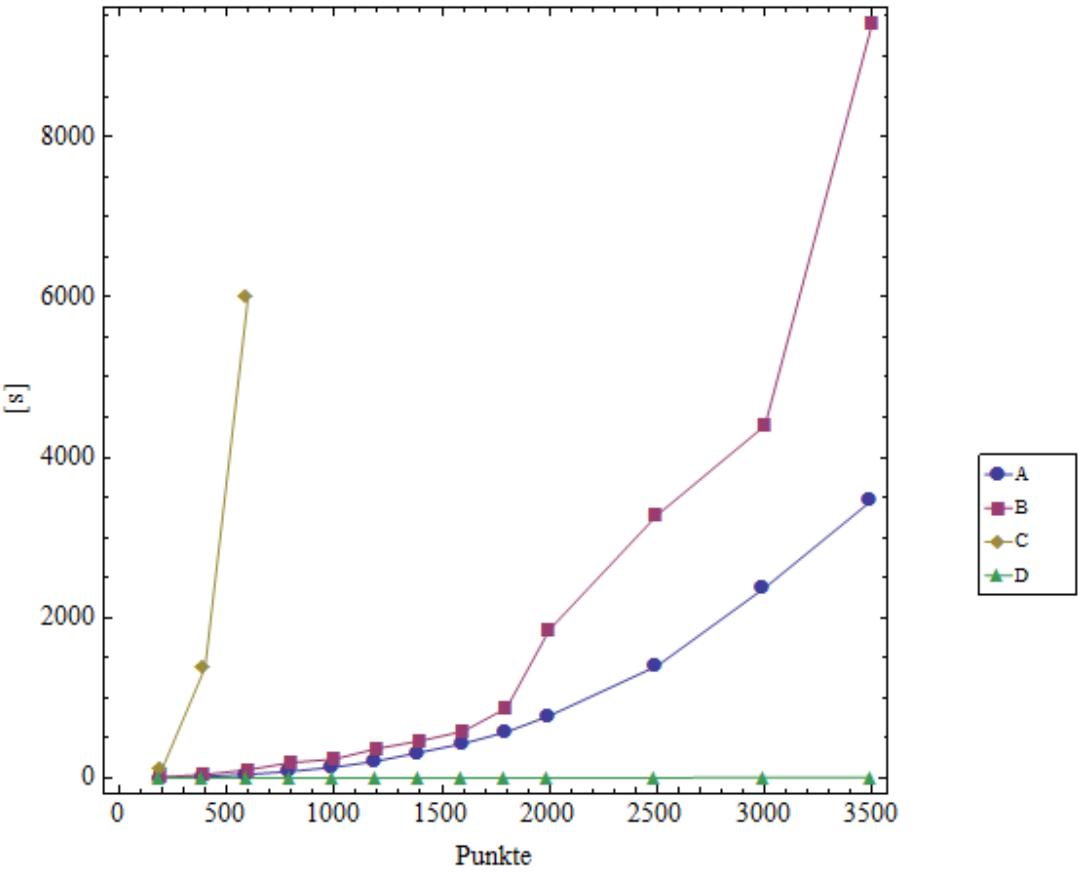


Abbildung 3.4: Laufzeiten der verschiedenen Versionen.

### 3.2.2 Laufzeitverhalten der Versionen A und D

In Abbildung 3.5 wird nur noch die Version A mit der Version D verglichen. Wichtig hierbei ist, dass für die Version A nur noch die lineare Steigung zwischen Messwert eins und zwei bis $\approx 10'000$ Punkte eingetragen wurde, da sonst die Kurve für Version D flachgedrückt würde. Es zeigt relativ deutlich, dass auch Version D einer exponentiellen Steigung unterliegt.

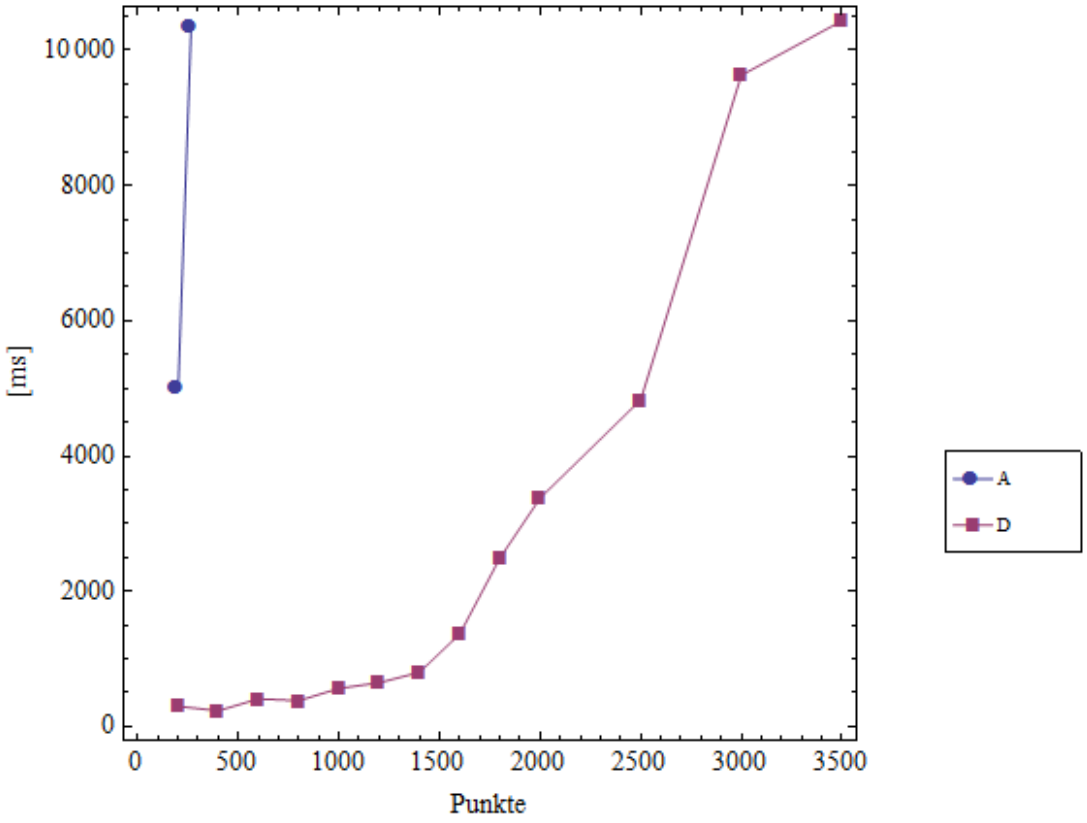


Abbildung 3.5: Laufzeitverhalten der Versionen A und D.

### 3.2.3 Verbesserung Version D gegenüber A und B

Der Geschwindigkeitszuwachs der Version D verglichen mit den beiden Versionen A und B ist in Abbildung 3.6 dargestellt. Der Einbruch bei $\approx 1800$ Punkten hat mehrere Gründe: Einerseits ist Version D sehr viel kürzer in der Abarbeitungsdauer, sodass sich die Scheduling-Strategie des Betriebssystems direkt auf die Dauer auswirkt, sobald ein Prozesswechsel stattfindet. Andererseits weisen die Versionen A und D unterschiedliche Präzisionen der Genauigkeit auf, wodurch der Integration-Loop nicht gleich viele Lernschritte umfasst.

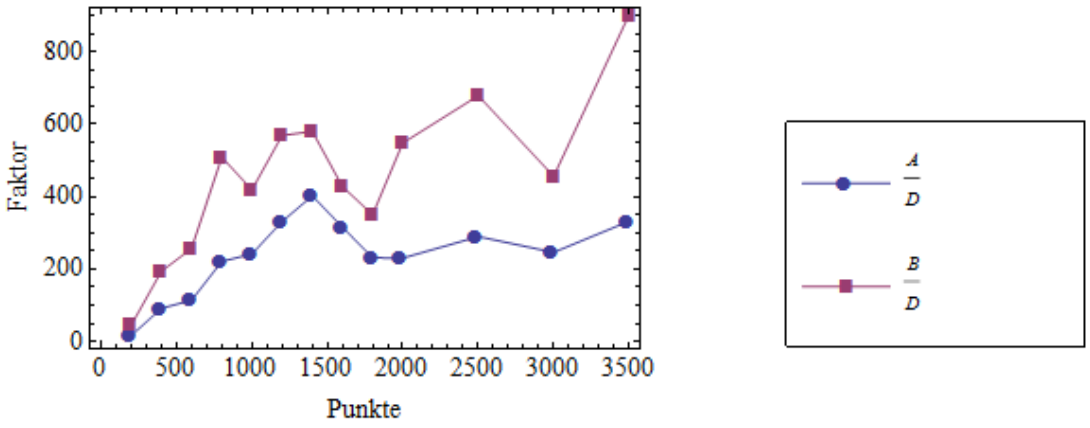


Abbildung 3.6: Performancegewinn der Version D verglichen mit den Versionen A und B.

### 3.2.4 Zeitverteilung, Version A intern

Die interne Zeitverteilung für Version A in Abbildung 3.7 zeigt deutlich, wo die meiste Zeit verbraucht wird. Der Extrahierungsprozess nimmt mit zunehmender Punktzahl exponentiell zu.

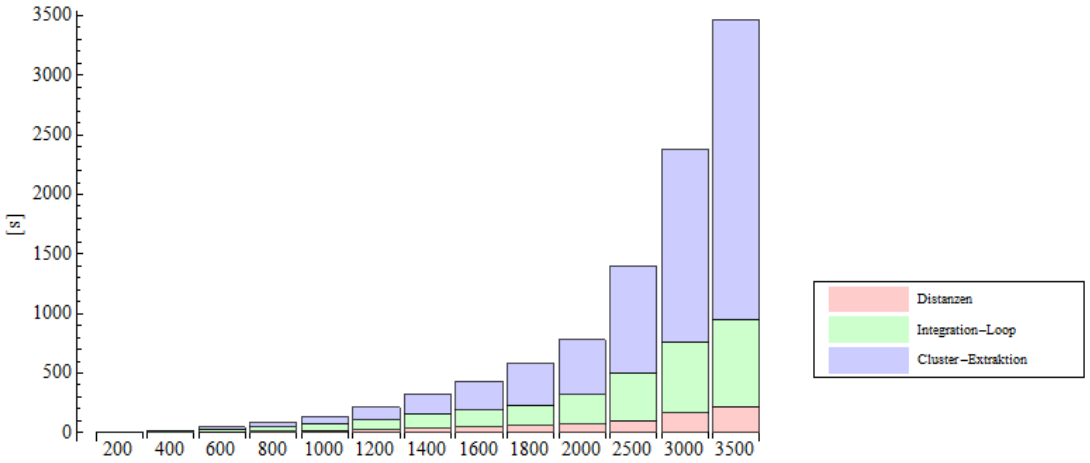


Abbildung 3.7: Zeitaufwendungen der Version A für Teilbereiche des Clusterings.

### 3.2.5 Zeitverteilung, Version D intern

Bei der Version D sieht die interne Zeitverteilung etwas anders aus, wie Abbildung 3.8 zeigt. Hier wird die meiste Zeit im Integration-Loop aufgewendet. Die Extraktion der Cluster hingegen macht nur noch einen Bruchteil des ganzen Prozesses aus.

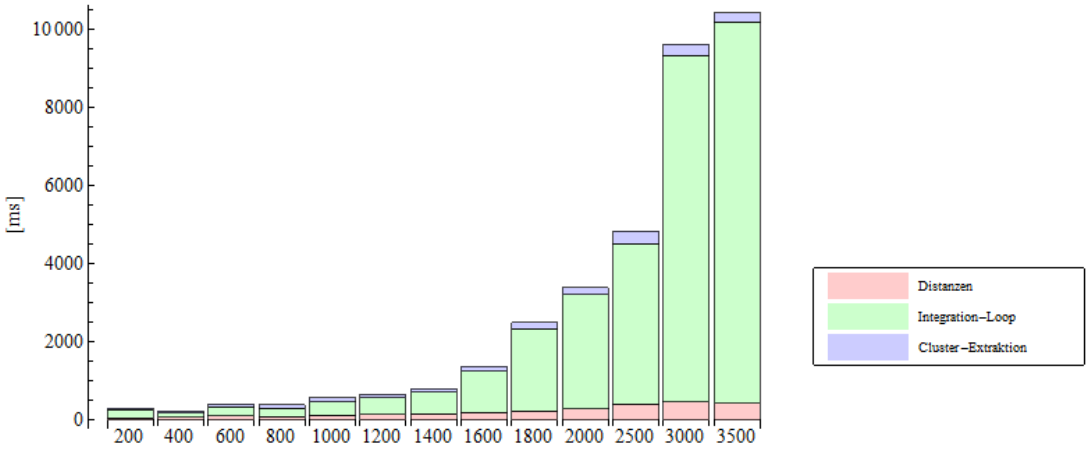


Abbildung 3.8: Zeitaufwendungen der Version D für Teilbereiche des Clusterings.

## 3.3 Theoretische Laufzeit der Version D

In dieser Sektion wird das Laufzeitverhalten von zwei Teilen unserer Portierung beleuchtet. Diese zwei Stellen sind die Distanz-Berechnung und die Extraktion der Cluster. Analysiert wird jeweils das Worst-Case Verhalten.

### 3.3.1 Problemgrössen

Die Berechnungen wurden mit den Problemgrössen gemäss Abbildung 3.9 durchgeführt.

### 3.3.2 Gewichtung der Steuerbefehle

Um das O-Verhalten so präzis wie möglich anzugeben, haben wir den Befehlen Gewichte zugeordnet. Zudem haben wir noch Kürzel für die Befehle eingeführt, damit der Ausdruck in O-Notation kürzer wird. Die Tabelle ist

| Grösse | Beschreibung |
|---|---|
| $N$ | Anzahl Datenpunkte aus der Vorgabe |
| $d$* | Dimension der Vorgabe |
| $k$* | Anzahl der Nachbarschaftsneuronen |

Abbildung 3.9: Problemgrössen in der O-Notation.
(*) Es ist zu beachten, dass $d$ und $k$ vernachlässigbar kleine Werte gegenüber $N$ darstellen.

in Abbildung 3.10 ersichtlich. Einige Befehle wurden hierbei zusammengenommen, wie z. B. die Subtraktion eine Addition ist oder die Division eine Multiplikation ist.

| Beschreibung | Kürzel | Gewicht | C++-Äquivalente Befehle |
|---|---|---|---|
| Zuweisung | ZUW | 1 | $Var = 42;$ |
| Addition | ADD | 2 | $Var+ = 42; Var[]; a - b$ |
| Multiplikation | MUL | 2 | $Var* = 42; Var/ = 42;$ |
| Funktion | FNC | 3 | $exp(..); pow(..); max(..);$ |
| Listen-Funktion [11] | COF | 1 | $push\_back(..); front();$ $pop\_front();$ |
| Logik-Test | LIF | 1 | $if(..); a < b;$ |
| Sortierung | SRT(x) | $x * log(x)$ | $sort(..); heap\_sort(..);$ |
| Suchen [11] | FND(x) | $x$ | $find(..);$ |

Abbildung 3.10: Gewichtung der Steuerbefehle zur Berechnung der O-Notation.

### 3.3.3 Abstands-Berechnung

Die Berechnung der Abstände zwischen den Datenpunkten kann in folgenden Ausdruck der O-Notation überführt werden:

$$O(N(3ADD + LIF + k(5ADD + LIF) + 2ZUW + N(ADD(11 + 8d) + (4 + d)FNC + 6LIF + dLIF + 3MUL + SRT(k) + 7ZUW)))$$

Wenn nun die Gewichtungen aus Abbildung 3.10 eingesetzt werden, kann auf den folgenden Ausdruck vereinfacht werden:

$$O(N(9 + 11k + N(20d + 53 + k * log(k))))$$

Diese Formel sieht im Worst-Case wie folgt aus: $O(N^2(d + k * log(k)))$.

Durch weglassen von $k$ und $d$ kann die Laufzeit mit $\approx O(N^2)$ angegeben werden.

### 3.3.4 Cluster Extraktion

Die Extraktion der Cluster kann in folgenden Ausdruck der O-Notation überführt werden:

$O(N(5ADD + 2COF + 2LIF + 2ZUW + N(2ADD + 3COF + LIF + k(14ADD + 7COF + 2FND(N) + 6LIF) + 2ZUW)))$

Wenn nun die Gewichtungen aus Abbildung 3.10 eingesetzt werden, kann auf den folgenden Ausdruck vereinfacht werden:

$O(N(16 + (10 + k(41 + 2k))N))$

Diese Formel sieht im Worst-Case wie folgt aus: $O(k * N^3)$. Durch weglassen von $k$ kann die Laufzeit mit $\approx O(N^3)$ angegeben werden.

# 4

# Anwendung: Suche von Shrimps

In diesem Kapitel beschreiben wir eine Anwendung des HLC Algorithmus. Diese Anwendung besteht darin, die TIMIT Daten zu clustern und in diesen Clustern shrimp-ähnliche Formen zu finden. Zuerst erklären wir unsere Vorbereitungen für diese Suche. Weiter zeigen wir Probleme bei der Suche auf und untersuche mögliche Lösungen dazu.

## 4.1 Vorgehen

Die TIMIT Daten sind eine grosse Sammlung an Tonaufnahmen gesprochener Sätze von Männern und Frauen quer durch die USA. Aus diesen Daten haben wir die einzelnen Phoneme *s* und *sh* der Männer und Frauen mittels Wavelet-Zerlegung herausgetrennt. Die Männer und Frauen haben wir separat behandelt, da wir uns durch die Geschlechtertrennung eine klarere Clusterabgrenzung erhofften.
Beide Dateien haben jeweils fünf Dimensionen. Damit diese Daten einfach darstellbar wurden, haben wir alle zehn möglichen 3D-Darstellungen davon erstellt. Diese Daten haben wir anschliessend mit dem HLC verarbeitet.
Die jeweiligen Resultate haben wir mittels Mathematica so dargestellt, dass wir einen 3D-Plot und drei 2D-Projektionen jeder Seite des 3D-Raums sehen konnten. Die 2D-Darstellungen dienten nur als Hilfe für ein schnelleres Erkennen der möglichen Muster.

## 4.2 Vorläufige Ergebnisse

In den TIMIT-Daten lassen sich Shrimps nach ersten Erkenntnissen nicht so leicht auffinden. Es sind zwar immer wieder ähnliche Muster vorhanden wie in Abbildung 4.1 dargestellt. Jedoch sind sie oftmals erst richtig erkennbar, wenn man diese künstlich verstärkt.

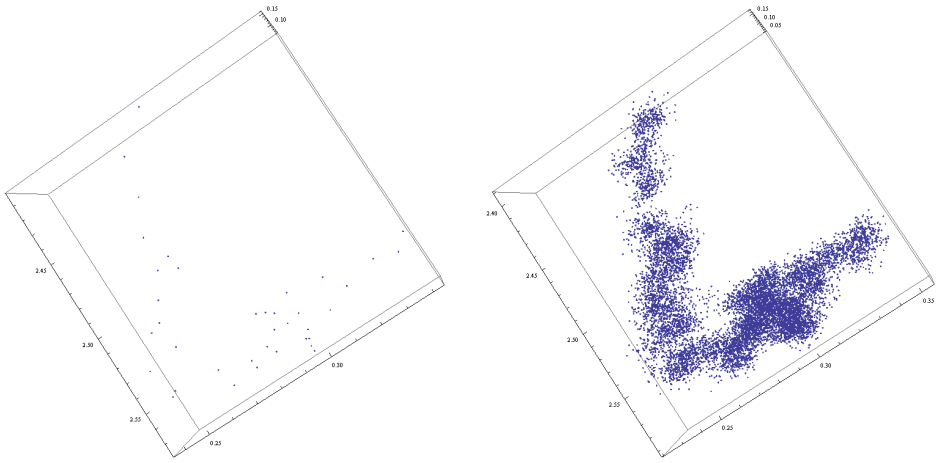

Abbildung 4.1: Ein möglicher Shrimp: Links die gefundene Struktur; Rechts die künstliche Verstärkung.

Eine andere Möglichkeit die Struktur im 2D-Raum besser erkennbar zu machen, ist die einzelnen Datenpunkte mit den jeweils nächsten $k$ Nachbarpunkten zu verbinden. Eine solche Darstellung der gleichen Struktur aus Abbildung 4.1 ist gezeigt in Abbildung 4.2.

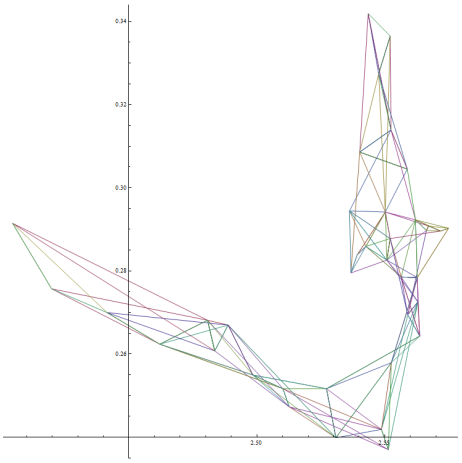

Abbildung 4.2: Nachbarn: Die nächsten sechs Nachbarn jeweils mit einer Linie verbunden.

### 4.2.1 Rauschanteil

Grundsätzlich kann nicht ausgeschlossen werden, dass die TIMIT-Daten einen grossen Rauschbeitrag beinhalten. In unseren Untersuchungen ist uns aufgefallen, dass die Extrahierung zwar das Rauschen erfolgreich rausfiltert, es jedoch oft kleinere Cluster gibt, die auch gefunden werden. Ein Beispiel dazu ist in Abbildung 4.3 zu sehen, wobei links die verrauschte Vorgabe ist und rechts alle 39 gefundenen Cluster dargestellt werden. Natürlich sind die zwei grössten Cluster die beiden Halbringe.

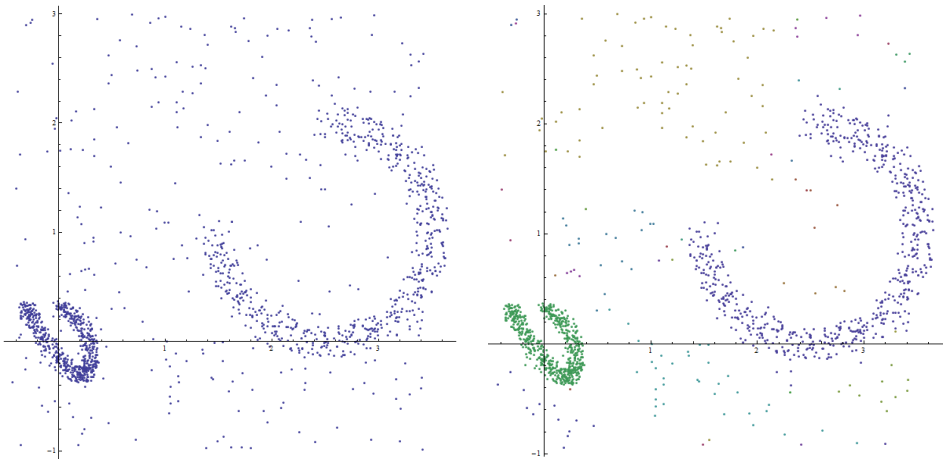

Abbildung 4.3: Rauschproblem: Links die verrauschte Vorgabe; Rechts alle 39 gefundenen Cluster farblich getrennt.

Wir haben uns deshalb gefragt, ob man das etwas schöner hinbekommen könnte. Dazu haben wir als Erstes Heat-Maps [12] über die synaptischen Stärken nach dem Clusteringprozess erzeugt, welche in Abbildung 4.4 gezeigt werden. Diese Heat-Maps erhält man dadurch, dass man die Punkte mit hoher synaptischer Stärke heisser (röter) darstellt, als solche mit tiefer synaptischer Stärke (weiss). Dabei sieht man insbesondere beim kleinen Halbkreis sehr schön, dass einige Stellen heisser sind als andere.

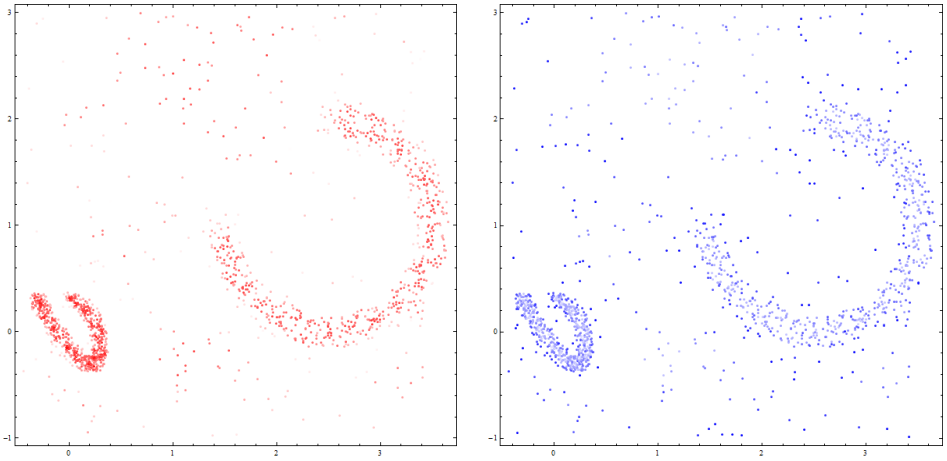

Abbildung 4.4: Heat-Map: Links nach heissen Punkten (je heisser desto röter, je kühler desto weisser); Rechts nach kalten Punkten (je kälter desto blauer, je heisser desto weisser).

Weiterführend haben wir den Extraktionsprozess dahingehend erweitert, als Alternative die HeatMap zur Extrahierung zu verwenden. Dabei werden kühlere Punkte nicht mit extrahiert, heissere hingegen schon. Dieselbe Vorgabe, wie bei Abbildung 4.3, geclustert, ergibt dann nur noch fünf Cluster. Das Ergebnis dazu ist in Abbildung 4.5 gezeigt.
Leider hat auch die Heat-Map Extrahierung zum Auffinden von shrimpähnlichen Strukturen noch nicht den gewünschten Erfolg gebracht. Weiterführend haben wir festgestellt, dass dieser Weg der Extrahierung nicht über alle Zweifel erhaben ist. Beim Smiley aus Abbildung 1.1 zum Beispiel können die Augen nicht mehr in Iris und Pupille unterteilt werden.

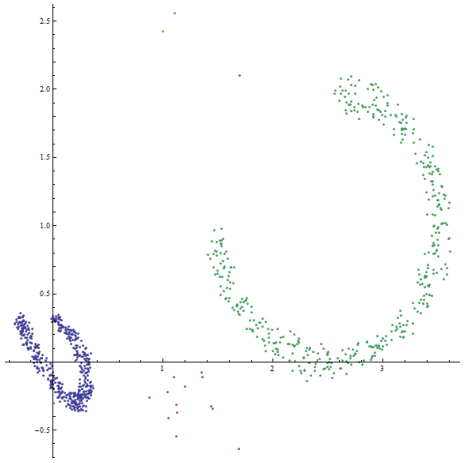


Abbildung 4.5: Heat-Map Extrahierung: Es werden weniger kleine, rauschartige Cluster gefunden.

# 5

# Schlussfolgerung

## 5.1 Offene Fragen

Die Abbruchbedingung, respektive das stoppen des Lernvorgangs durch den Stopp-Koeffizienten im Zusammenspiel mit dem abflachen der Lernkurve bei zunehmender Anzahl von Lernschritten, könnte bei grossen Vorgaben möglicherweise besser optimiert werden, sodass der Lernprozess schneller fertig wird.
In unserer Implementierung wurde der Speicherverbrauch zwar enorm reduziert, dennoch stellt sich die Frage, ob der Verbrauch klein genug ist, um den Algorithmus in Hardware zu implementieren, für den er sich im Prinzip von seiner Struktur her anbietet. Der Verbrauch liegt z. Z. für 10'000 Punkten bei ≈15MB (64bit).

## 5.2 Ausblick

Die Parallelisierung könnte ebenfalls weiter ausgebaut werden. Wie man in den Performancemessungen 3.2.5 sieht, ist der Integration-Loop der zeitintensivste Teil. Dieser Bereich ist noch nicht parallelisiert und könnte möglicherweise noch weiter optimiert werden.

# 6

# Persönliche Berichte

## 6.1 Jonathan Stolz

Clustering war ein mir bis zur Studienarbeit nur mittelmässig bekanntes Thema. Mit unserer Arbeit konnte ich den HLC besser kennen lernen. Ich war erstaunt, wie stark dieser Algorithmus bei der Erkennung von komplexen Objekten ist. Die Portierung und Optimierung dieses Algorithmus waren packende und abwechslungsreiche Tätigkeiten. Bei der Arbeit schätzte ich die spannende Aufgabenstellung aus den verschiedenen Bereichen der neuronalen Netze, des Clusterings, der Performance Optimierung und der Spracherkennung. Auch die Besprechungen mit Herrn Prof. Dr. Stoop waren für mich jedes Mal wieder eine weitere Motivation. Es würde mich interessieren und freuen, noch weitere Ansätze in diesem Gebiet erforschen zu können.

## 6.2 Thomas Jutzi

Das Thema Clustering finde ich ausgesprochen interessant. Vor Beginn der Studienarbeit war ich mir aber nicht über die Tragweite des Clusterings bewusst. Daher fand ich es sehr spannend, den HLC genauer unter die Lupe zu nehmen. Der HLC zeigte mir deutlich, was alles möglich sein kann in diesem Bereich. Was mir besonders gut gefallen hat, war das freie Arbeiten sowie die fortwährende motivierende Art von Herrn Prof. Dr. Stoop. Gerne würde ich weiterhin in diesem Bereich neue Ansätze verfolgen und umsetzen, da ich überzeugt bin, dass es hier noch sehr viel Neues zu entdecken gibt.

# 7

# Appendix

Der Übersicht halber verzichten wir an dieser Stelle auf das beifügen des Source Codes in C/C++ im Appendix. Dieser Code befindet sich nur digital auf der CD, welche mit der Arbeit zusammen abgegeben wurde.

# Literaturverzeichnis


[1] R. Stoop. Wissensbasierte Systeme, 2012.

[2] R. Stoop and F. Landis and Th. Ott. Hebbian Self-Organizing Integrate-and-Fire Networks for Data Clustering. *Neural Computation*, 22:273–288, 2010.

[3] Linguistic Data Consortium. http://www.ldc.upenn.edu/. Zuletzt besucht am 25.05.2012.

[4] J. Sanders and E. Kandrot. *CUDA BY EXAMPLE. An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[5] J. Sanders and E. Kandrot. *CUDA BY EXAMPLE. An Introduction to General-Purpose GPU Programming*, chapter 1, page 1. Addison-Wesley, 2011.

[6] Ordering - Wolfram Mathematica 8 Documentation. http://reference.wolfram.com/mathematica/ref/Ordering.html. Zuletzt besucht am 24.05.2012.

[7] Some Notes on Internal Implementation - Wolfram Mathematica 8 Documentation. http://reference.wolfram.com/mathematica/tutorial/SomeNotesOnInternalImplementation.html. Zuletzt besucht am 10.04.2012.

[8] The GNU Multiple Precision Arithmetic Library Official Website. http://gmplib.org/. Zuletzt besucht am 24.05.2012.

[9] F. Landis. A Self Organising, Spiking Neuron-Mediated Clustering Approach. Master's thesis, ETH Zürich, 2007.

[10] J. Sanders and E. Kandrot. *CUDA BY EXAMPLE. An Introduction to General-Purpose GPU Programming*, chapter 3, page 33. Addison-Wesley, 2011.

[11] B. Stroustrup. *Die C++ Programmiersprache*, volume 4. Addison-Wesley, 2009.

[12] Heat map. http://en.wikipedia.org/wiki/Heat_map/. Zuletzt besucht am 31.05.2012.