Optimierung einer Merkmalsextraktionsmethode

Oliver Hess
Christian Singer

Semesterarbeit SA-2004-32

Sommersemester 2004

Institut für Technische Informatik
und Kommunikationsnetze

Forschungsgruppe für Sprachverarbeitung

Betreuer: Ulrike Glavitsch, Beat Pfister

Verantwortlicher: Prof. Dr. L. Thiele

02.07.2004

Zusammenfassung

In der Sprachererkennung versucht man durch die Extraktion von Merkmalsvektoren die charakteristischen Eigenschaften eines Sprachsignals zu erfassen und somit die wichtigsten Informationen herauszufiltern, um in einem weiteren Schritt mit diesen Merkmalsvektoren statistische Modelle zu trainieren, welche später für die Spracherkennung verwendet werden.

Ziel dieser Arbeit ist die Verbesserung eines MATLAB Programms zur Merkmalsextraktion von Sprachsignalen. Dies umfasst im ersten Teil das vorhandene Programm hinsichtlich Laufzeiteffizienz zu optimieren und im zweiten Teil experimentell die Parameter dieser Methode so einzustellen, dass mit den verwendeten statistischen Modellen (Hidden Markov Modelle) eine möglichst gute Worterkennungsrate erzielt wird.

Um die Experimente in einem vernünftigen zeitlichen Rahmen durchführen zu können ist es notwendig, eine Testumgebung einzurichten. Darin wird ein Teil der SpeechDat(II)[1] Datenbank verwendet um die verschiedenen Parametereinstellungen und ihre Auswirkungen auf die Erkennungsrate genau und effizient zu untersuchen. Diese Datenbank umfasst im Wesentlichen eine grosse Menge von Sprachsignalen, die sowohl hinsichtlich Inhalt als auch Sprechereigenschaften von Testpersonen eine für die Spracherkennung repräsentative Auswahl darstellt.

Inhaltsverzeichnis

1 Einführung

1.1 Konzept der automatischen Spracherkennung

Um ein Sprachsignal automatisch in Text umzuwandeln muss man zuerst das Sprachsignal in eine Form bringen, welche möglichst nur die phonetisch relevanten Informationen enthält. Dies erreicht man, indem man eine Kurzzeitspektrums-Analyse [2] vornimmt, das heisst zu vielen verschiedenen Ausschnitten des Sprachsignals die Fourier-Transformation berechnet. Die Position und Länge dieser Abschnitte hängt vom Verfahren ab, welches man verwendet. Wie man aus den so gewonnenen Frequenzspektren die geeigneten Merkmale extrahiert ist ebenfalls eine Eigenschaft des Verfahrens. Als wohl etabliertes Verfahren sei hier die MFCC [3] Methode genannt, welche in dieser Arbeit auch als Referenzverfahren verwendet wird. Man erhält durch die Merkmalsextraktion aus den Kurzzeitspektren eine Sequenz von Vektoren, wobei jeder Vektor einen Abschnitt des Signals charakterisiert. Das Sprachsignal liegt nun in einer kompakteren Form vor, wobei die phonetischen Eigenschaften des Signals so gut wie möglich in diesen Merkmalsvektoren codiert sein sollten.

Mit diesen Merkmalsvektoren können nun statistische Modelle (Hidden Markov Modelle oder Neuronale Netze) trainiert werden. Das Ziel ist, dass nach dem Training mit einer genügend hohen Anzahl Lernbeispiele durch die statistischen Modelle eine möglichst fehlerfreie Übersetzung von einem Sprachsignal in textuelle Form durchgeführt werden kann (siehe Fig. 1).

1.2 Merkmalsextraktion basierend auf Formanten

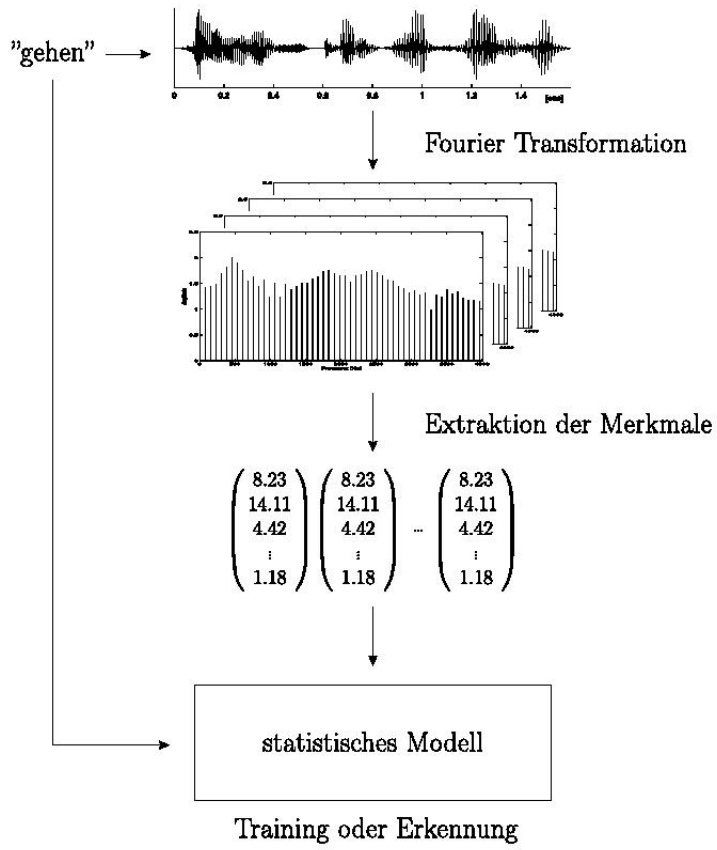
Untersuchungen der menschlichen Sprachwahrnehmung haben gezeigt, dass die wohl wichtigste in einem Sprachsignal enthaltene Information die spektralen Spitzen sind. Diese sogenannten *Formanten* entsprechen Resonanzfrequenzen im Vokaltrakt[1]. Aufgrund dieser Erkenntnis hat U. Glavitsch in ihrer Doktorarbeit eine Methode zur Merkmalsextraktion implementiert, welche auf den eben genannten Formanten basiert. Das Verfahren funktioniert grob beschrieben wie folgt:

Im ersten Schritt wird aus dem Sprachsignal, welches im Wesentlichen einer Folge von Abtastwerten entspricht, alle 10 Millisekunden ein pitch-synchrones Spektrum berechnet[2]. Die aktuellen Grundtöne zu diesen Zeitpunkten sind im Voraus berechnet worden und liegen in einem File vor, welches vorgängig eingelesen werden muss. Aufgrund des jeweils aktuellen Grundtones kann man nun berechnen wie lange der Signalabschnitt sein muss, damit das daraus berechnete Spektrum pitch-synchron ist. Das Signal wird auf diese Weise in mehrere überlappende Signalabschnitte aufgeteilt, die wir im Folgenden als *Frames* bezeichnen (siehe Fig. 2).

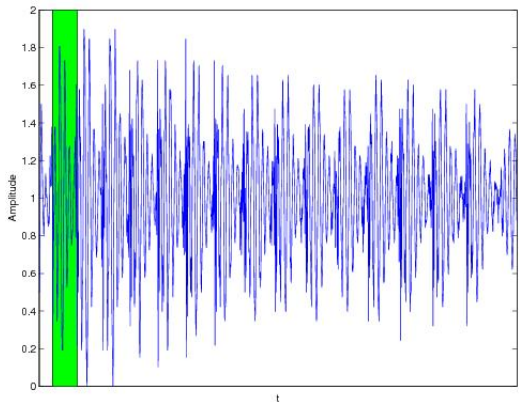
Wenn man das so berechnete pitch-synchrone Spektrum visualisiert, so kann man gewisse Formanten schon von Auge erkennen (siehe Fig. 3).

[1] Rachen, Mund und Nasenraum

[2] Von einem pitch-synchronen Spektrum spricht man, wenn die Länge des Signalabschnitts, für welchen das Spektrum berechnet wird, der Periode des Grundtons an der aktuellen Stelle im Signal entspricht.

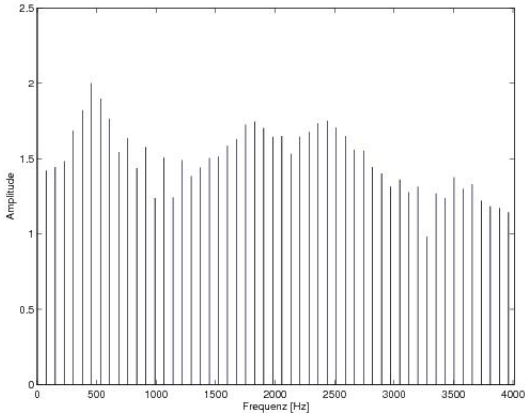


Figur 1: *Das Konzept der automatischen Spracherkennung, illustriert am Beispiel des Wortes "gehen"*



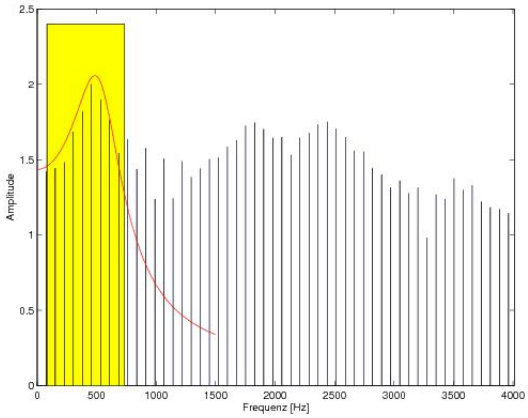
Figur 2: *Ein ausgeschnittenes Frame im digitalen Sprachsignal*

Die Herausforderung bei diesem Verfahren liegt in der möglist exakten Lokalisierung dieser Formanten. Deshalb wird im zweiten Schritt versucht, mit mathematischen Mitteln diese Formanten zu erkennen, um sie anschliessend abzuspeichern. Dazu wird das Spektrum an möglichst vielen Stellen mit einer *Templatefunktion* verglichen [4]. Diese kann man sich als Hutfunktion vorstellen, welche die Form eines Formanten abhängig von seiner Position



Figur 3: *Das pitch-synchrone Spektrum eines Frames. Die ersten drei Formanten sind bei 500, 1600 und 2500 Hz deutlich sichtbar.*

im Spektrum gut beschreibt (siehe Fig. 4).

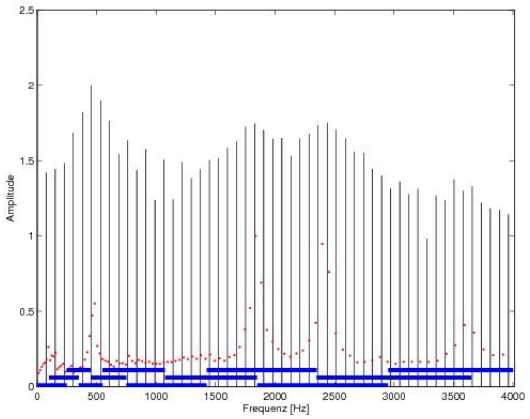


Figur 4: *Illustration der Template Funktion*

An welchen Stellen, d.h. Frequenzen, diese Funktion berechnet und mit dem Spektrum verglichen wird, wird in einem File spezifiziert, der Formant Bank. Für jeden *Gitterpunkt* in der *Formant Bank* wird nun die cepstrale Distanz[3] zwischen der Templatefunktion und dem Spektrum berechnet. Das Inverse dieser Distanz kann als Ähnlichkeitswert interpretiert werden, welchen wir im Folgenden als *Score* des jeweiligen Gitterpunktes bezeichnen. Des Weiteren wird für jeden Gitterpunkt ein Energiewert berechnet, der die lokale Energie des Spektrums an dieser Stelle wiedergeben soll. Diese zwei Werte pro Gitterpunkt in der Formant Bank werden nun in einem Score-Vektor abgespeichert.

Weil die Positionen der Formanten von verschiedenen Sprechern stark variieren können, definiert man *Toleranzbereiche*, d.h. sich überlappende Intervalle auf der Frequenzachse, welche gemäss den in [5] erforschten Formantbereichen gewählt werden. Pro Toleranzbereich wird immer nur der höchste darin liegende Score Wert gewählt und mit der dazugehörigen Energie gewichtet. Dieser Wert entspricht nun einem Eintrag im Merkmalsvektor

[3]Die cepstrale Distanz zwischen zwei Vektoren entspricht der euklidschen Distanz zwischen den Realteilen der inversen Fouriertransformierten der Logarithmen der beiden Vektoren. Weitere Informationen zum Cepstrum können in [2] nachgelesen werden



Figur 5: *Toleranzbereiche und Score Werte*

oder *Feature Vektor*. Wir haben so für jeden Toleranzbereich einen Wert berechnet, der aussagt, ob ein Formant im jeweiligen Bereich liegt oder nicht. Ist der Wert klein, so wurde bei keinem der betrachteten Gitterpunkte ein hoher Scorewert berechnet und folglich befindet sich auch kein Formant in diesem Bereich. Ist der Wert aber gross, so wurde bei mindestens einem der Gitterpunkte ein hoher Score berechnet und dieser kann nur durch einen Formanten verursacht werden. Also liegt in diesem Bereich ein Formant (siehe Fig. 5).

Pro Sprachsignal wird durch dieses Verfahren eine Sequenz von Merkmalsvektoren generiert, die - wie bereits erwähnt - verwendet werden kann, um statistische Modelle zu trainieren.

1.3 Aufgabenstellung

Das beschriebene Verfahren liegt als MATLAB Programm vor. Unsere Aufgabe besteht aus folgenden Teilen:

- Aneignen theoretischer Grundkenntnisse auf dem Gebiet der Sprachverarbeitung mit Hilfe der uns zur Verfügung gestellten Literatur [2]
- Studium des vorhandenen MATLAB Programms und Entwurf eines Konzepts für eine effiziente Implementation dieser Methode unter den Gesichtspunkten einer leichten Erweiterbarkeit und guten Wartbarkeit
- Implementation des Konzepts und Vergleich des Outputs mit der Ausgabe des bisherigen Programms, um Korrektheit zu überprüfen
- Einrichten einer Testumgebung um verschiedene Erkennungsexperimente durchzuführen, bei denen die Parameter variiert und deren Auswirkungen auf die Erkennungsrate analysiert werden.

2 Performance Optimierung

2.1 Problemstellung

Als ersten Schritt in der Optimierung des Merkmalsextraktionsverfahrens soll das bestehende MATLAB Programm in ein effizientes C oder Java Programm übersetzt werden.

Die wesentlichen Anforderungen an die Neuimplementation sind:

- Drastisch verminderte Laufzeit, um effizientes Experimentieren zu ermöglichen
- Erweiterbarkeit und Wartbarkeit

Ferner soll das neuimplementierte Programm selbstverständlich den selben Output produzieren, wie die bestehende Implementation.

2.2 Struktur des vorgegebenen Programms

Obwohl sich dieses Kapitel mit der Performance Optimierung des bestehenden Programms auseinandersetzt, möchten wir versuchen, den direkten Bezug auf grosse Programmcodestücke zu vermeiden und stattdessen die Ideen und Konzepte, welche wir für die Optimierung verwenden, herauszuarbeiten. Für ein Verständnis der verwendeten Techniken und der erzielten Resultate ist es jedoch trotzdem nötig, auf einer abstrakten Ebene eine Übersicht über die Struktur des bestehenden Codes zu gewinnen.

Die Feature Extraktion besteht im Wesentlichen aus zwei Funktionen, welche von einem Controller Programm aus aufgerufen werden (siehe Fig. 6):

1. Die erste Funktion `extract_features_all_phonemes(signal_file_list)` iteriert über alle Signaldateien in der als Argument übergebenen Dateiliste. Für jedes Signal wird wiederum über alle Frames iteriert, deren Länge berechnet und anschliessend mit der Funktion `cepstral_dist_frmt_bank(frame)` für jedes Frame der entsprechende Scorevektor generiert. Dieser enthält für jeden Gitterpunkt das Inverse der cepstralen Distanz (siehe Fig. 8). Alle Scorevektoren eines Signals werden anschliessend in einer Matrix zusammengefasst und zur späteren Weiterverarbeitung in ein File (Score File) auf Platte geschrieben (siehe Fig. 7).
2. Im zweiten Schritt der Feature Extraktion wird die Funktion `generate_feature_vecs(score_file_list)` verwendet, welche aus allen Score Files in der als Argument übergebenen Dateiliste mit Hilfe der Toleranzbereiche die endgültigen Features generiert (siehe Fig. 9).

Zu beachten ist, dass die in den Figuren 6, 7, 8 und 9 angegebenen Programmskizzen stark vereinfacht sind und lediglich die Information beinhalten, welche für das Verständnis dieses Kapitels notwendig ist.

```
controller(signal_file_list, score_file_list)
{
  extract_features_all_phonemes(signal_file_list);
  generate_feature_vecs(score_file_list);
}
```

Figur 6: *Programm zur Durchführung der Feature Extraktion*

```
extract_features_all_phonemes(signal_file_list)
{
  foreach(file in signal_file_list)
  {
    signal = read_signal(file);
    for(i=0; i<length(signal); i=i+10ms)
    {
      frame = get_frame(signal[i]);
      scores[i] = cepstral_dist_frmt_bank(frame);
    }
    save_scores_file(scores);
  }
}
```

Figur 7: *Aufbau der Funktion extract_features_all_phonemes*

```
cepstral_dist_frmt_bank(frame)
{
  foreach(gridpoint in formant_bank)
  {
    lb = get_lower_bound(gridpoint);
    hb = get_higher_bound(gridpoint);
    template = generate_template(gridpoint, lb, hb);
    score[i++] = 1 / cepstral_distance(template, frame(lb..hb);
  }
  return(score);
}
```

Figur 8: *Aufbau der Funktion cepstral_dist_frmt_bank*

```
generate_feature_vecs(score_file_list)
{
  foreach(file in score_file_list)
  {
    scores = read_scores_file(file);
    foreach(score_vector in scores)
    {
      features[i++] = score2feature(score_vector);
    }
    save_features_file(features);
  }
}
```

Figur 9: *Aufbau der Funktion generate_feature_vecs*

2.3 Ursprüngliches Konzept der Neuimplementation

2.3.1 Wahl der Programmiersprache

Bei der Wahl der Programmiersprache besteht unserer Ansicht nach ein klarer Trade-off zwischen Effizienz und Erweiterbarkeit/Wartbarkeit. Eine Implementation in C ist mit Sicherheit effizienter[4], wohingegen ein Java Programm einfacher verständlich ist und sich somit besser warten und erweitern lässt.

Wir haben uns für eine Neuimplementation in der Programmiersprache C entschieden, um einen maximalen Effizienzgewinn erzielen zu können. Da sich das Programm unserer Ansicht nach sehr gut auf saubere Art und Weise in einzelne Funktionen mit klar definierten Schnittstellen unterteilen lässt, ist trotz dieser Wahl auch die spätere Erweiterbarkeit problemlos gewährleistet. Ferner ist gemäss unserer Betreuerin U. Glavitsch, der Autorin des bestehenden Programms, davon auszugehen, dass an dem Verfahren keine wesentlichen strukturellen Veränderungen mehr durchgeführt werden, da eine Steigerung der Erkennungsrate ausschliesslich durch bessere Parametereinstellungen erreicht werden soll (siehe Abschnitt 3 und folgende).

2.3.2 Definition von Interfaces und Datentypen

Um Wartbarkeit und spätere Erweiterbarkeit des Programms sicherzustellen, wird ein top-down Approach verfolgt:

1. Analyse des bestehenden Programms und Indentifikation der verwendeten mathematischen Operationen, Funktionen und Datentypen
2. Design und Definition der benötigten Datenstrukturen zur Darstellung von
   - Parametern des Verfahrens

[4] ein Benchmark, welcher diese These unterstützt, kann unter [6] nachgelesen werden

- Mathematischen Datentypen
- Rückgabewerten von Funktionen

3. Design und Definition der Interfaces aller benötigten Funktionen in Header Files
4. Entwurf grober Skelette der wichtigsten und aufwändigsten Funktionen

Wir möchten an dieser Stelle vorwegnehmen, dass die hier beschriebene Idee der Neuimplementation nicht weiterverfolgt wird und dass daher die Resultate der oben beschriebenen Schritte einer Prüfung auf Konsistenz und Korrektheit unter Umständen nicht standhalten würden. Falls sich der Leser trotzdem für die Resultate dieser Designphase interessiert, so ist der entsprechende ausführlich kommentierte Code in Anhang A zu finden.

2.4 Laufzeitanalyse mit dem MATLAB Profiler

Durch das intensive Studium des bestehenden Programms während der Definition der Interfaces und Datentypen sind wir zu einem besseren Verständnis der Komplexität einiger der im Programm durchgeführten Operationen gekommen[5]. Wir denken, dass diese Operationen im Rahmen dieser Semesterarbeit selbst durch den Einsatz der sehr effizienten Programmiersprache C kaum um ein Vielfaches schneller implementiert werden könnten, als sie von MATLAB zur Verfügung gestellt werden.

Diese Einsicht, sowie Hinweise einiger Mitglieder der Forschungsgruppe für Sprachverarbeitung führten uns zur Idee, vor Beginn der Neuimplementation eine Laufzeitanalyse des bestehenden Programms durchzuführen, um allenfalls bestehende Bottlenecks zu finden und anschliessend nur gewisse Teile des bestehenden Programms einer Laufzeitoptimierung zu unterziehen[6].

Um die Dauer der Laufzeitanalyse in einem vernünftigen Rahmen zu halten, wurden sämtliche Analysen auf einer repräsentativen Stichprobe der SpeechDat(II) Datenbank durchgeführt. Diese Datenbank wird später auch für die Erkennungsexperimente mit dem neuen Verfahren verwendet.

Bereits ein erster Versuch einer Analyse mit den MATLAB Commands `tic` und `toc`[7], auf den wir hier nicht weiter eingehen möchten, zeigt, dass der grösste Teil der Laufzeit durch die Funktion `extract_features_all_phonemes` verursacht wird.

Um genauere und feingranularere Analysen durchführen zu können, wird im Folgenden das MATLAB Tool `profiler` verwendet. Um eine schnelle Auswertung der so entstehenden Resultate zu ermöglichen, wird eine selbst verfasste MATLAB Funktion zur massgeschneiderten graphischen Darstellung der Laufzeiten der einzelnen Programmteile verwendet, welche Balkendiagramme, wie sie beispielsweise in Figuren 10-12 zu sehen sind, erstellt. Die Diagramme zeigen pro Funktion einen mit dem Funktionsnamen beschrifteten Balken,

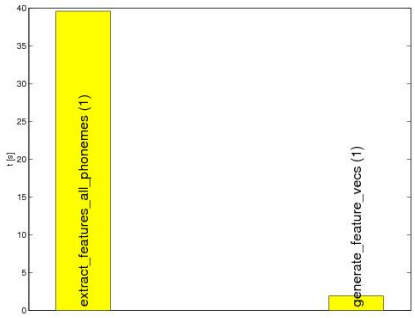
[5] Wir sprechen hier primär die diskete Fouriertransformation und deren Inverse, sowie diskrete Cosinustransformation an.

[6] Dies kann einerseits durch eine partielle Neuimplementation in einer effizienteren Sprache geschehen, andererseits auch durch eine Optimierung bestehender MATLAB Funktionen

[7] Diese Funktionen starten- bzw. lesen einen Stopwatch Timer

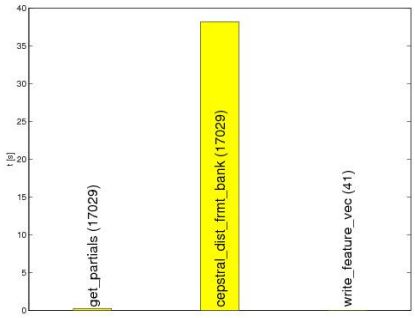
welcher die durchschnittliche Laufzeit der Funktion pro Signaldatei in Sekunden repräsentiert. In Klammern ist zusätzlich die Anzahl der Funktionsaufrufe angegeben. Damit die Ergebnisse der Laufzeitanalysen vergleichbar sind, werden die Evaluationen immer auf der gleichen Maschine durchgeführt[8]

Eine mit diesen Hilfsmitteln erstellte Analyse der beiden Hauptfunktionen `extract_features_all_phonemes` und `gen_feature_vecs` liefert eine Bestätigung der ersten Annahme: Die schlechte Laufzeit wird primär durch `extract_features_all_phonemes` verursacht (siehe Fig. 10).



Figur 10: *Die Laufzeiten der Funktionen extract_features_all_phonemes und generate_feature_vecs*

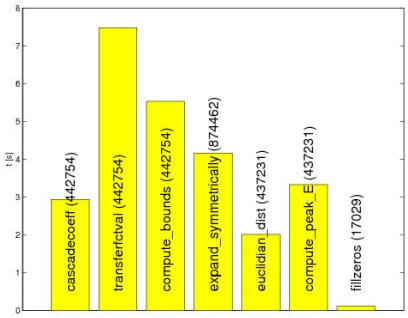
Dieses Resultat legt nahe, die Funktion `extract_features_all_phonemes` genauer zu analysieren. Hier zeigt sich, wie in Figur 11 ersichtlich, wiederum eine klare Dominanz einer einzigen Funktion, namentlich `cepstral_dist_frmt_bank`. Eine genauere Untersuchung dieser Funktion liefert jedoch kein so eindeutiges Bild mehr, wie man in Figur 12 erkennen kann.



Figur 11: *Eine genauere Analyse der Funktion extract_features_all_phonemes zeigt: der Hauptteil der Laufzeit entsteht durch eine einzige Funktion.*

Im Weiteren wird als Folge der oben erzielten Resultate lediglich noch die Funktion `cepstral_dist_frmt_bank` betrachtet und Schritt für Schritt optimiert. Auf die Funktionen, welche von `cepstral_dist_frmt_bank` aufgerufen werden, möchten wir an dieser Stelle nicht weiter eingehen, da eine tiefere Betrachtung für ein konzeptionelles Verständnis der Optimierung nicht notwendig ist.

[8]Wir haben unseren Arbeitsplatzrechner, eine Sun Blade 100, verwendet.



Figur 12: *Aus der Analyse der Funktion cepstral_dist_frmt_bank kann keine der aufgerufenen Funktionen mehr als Hauptverursacher für die hohe Laufzeit ermittelt werden.*

2.5 Optimierung der Funktion cepstral_dist_frmt_bank

Um dem Leser einen Eindruck zu vermitteln, wie wir bei der Performance Optimierung der Funktion `cepstral_dist_frmt_bank` vorgegangen sind, möchten wir die wesentlichen fünf Optimierungsschritte, welche uns zum Ziel geführt haben, in chronologischer Reihenfolge darlegen. Wir werden uns jedoch darauf beschränken, die verwendeten Techniken an einfachen Beispielen kurz zu erläutern und jeweils anschliessend die durch ihren Einsatz erreichten Laufzeitverbesserungen aufzuzeigen.

In den ersten drei Schritten werden durch Optimierung bestehender MATLAB Funktionen Verbesserungen in der Laufzeit erzielt, die letzten zwei Schritte beschäftigen sich mit dem Einsatz von C-MEX Funktionen zur Performance Steigerung.

2.5.1 Schritt 1: Inlining häufig verwendeter Funktionen, Loop Unrolling

Inlining von Funktionen, also ersetzen eines Function Call durch den Body der aufzurufenden Funktion, ist zwar hinsichtlich Übersichtlichkeit und strukturiertem Aufbau nicht sinnvoll, wohl aber bezüglich Performance. Wieviel Zeit im Extremfall durch das Einspaaren von Funktionsaufruf und Resultatrückgabe eingespart werden kann, illustriert das folgende einfache MATLAB Beispiel:

```
function s = add3(x, y, z)
  s = x + y + z;

>> tic;
>>   for(i = 1:100000) s = add3(i, i+1, i+2); end;
>> toc;

elapsed_time =

    7.4956


>> tic;
>>   for(i = 1:100000) s = i + (i + 1) + (i + 2); end;
```

```
>> toc;

elapsed_time =

    1.9974
```

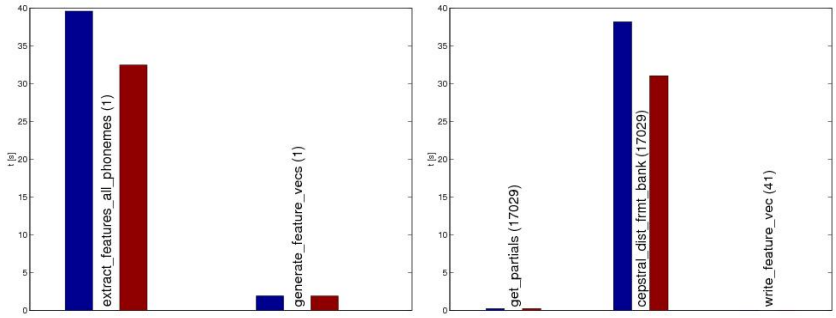
In dem bestehenden MATLAB Code kann die Laufzeit mehrerer kleiner Funktionen, welche nur an genau einer Stelle im ganzen Programm ausgeführt werden, durch Inlining verringert werden.

Loop Unrolling ist besonders einfach anzuwenden, wenn Abbruchbedingungen von Loops statisch sind. Der Code wird durch das Unrolling zwar grösser, jedoch auch deutlich schneller:

```
>> tic;
>>   for(i = 1:100000)
       s = 0;
       for(k=1:9) s = s + k; end;
     end;
>> toc;

elapsed_time =

    7.4400


>> tic;
>>   for(i = 1:100000)
       s = 0;
       s = s + 1; s = s + 2; s = s + 3;
       s = s + 4; s = s + 5; s = s + 6;
       s = s + 7; s = s + 8; s = s + 9;
     end;
>> toc;

elapsed_time =

    3.9054
```

Auch diese Technik kann für eine erste Optimierungsstufe im bestehenden Code angewendet werden.

Durch die beschriebenen Techniken kann die Laufzeit von `cepstral_dist_frmt_bank` auf 70.7% verringert werden. Dies entspricht einer Reduktion auf durchschnittlich 27.0s (siehe Fig. 13).



Figur 13: *Laufzeitvergleich: Originalprogramm vs. erste Optimierungsstufe*

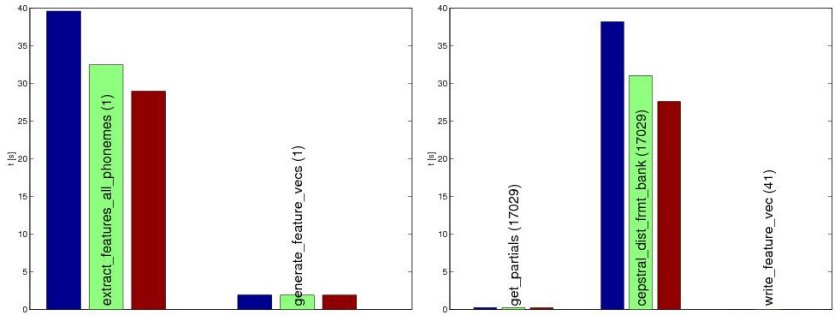
2.5.2 Schritt 2: Vektorisierung des Codes

Vektorisierung ist eine bekannte und äusserst effektive Art, MATLAB Code zu optimieren. Von vektorisiertem Code spricht man, wenn Operationen, welche man in einer regulären imperativen Programmiersprache durch Loops implementiert, durch MATLAB spezifische Vektor- und Matrixoperationen ersetzt werden. Wir möchten die Auswirkungen von Vektorisierung auf die Laufzeit anhand des konkreten Beispiels "Summe der Zahlen von 1 bis n" verdeutlichen:

```
>> one_to_n = (1:100000);
>>
>> tic;
>>   s = 0;
>>   for(i=1:100000) s = s + one_to_n(i); end;
>> toc;

elapsed_time =

    0.7232


>> tic;
>>   s = sum(one_to_n);
>> toc;

elapsed_time =

    0.0435
```

Diese Technik kann mit einigem Nachdenken auch auf grössere und kompliziertere Probleme angewendet werden. Zum allgemeinen Vorgehen bei der Vektorisierung von MATLAB Code, sowie zum technischen Hintergrund der daraus entstehenden extremen Laufzeitverminderung möchten wir auf die MathWorks Webseite [7] verweisen, wo Details und weitere Beispiele zum Thema "MATLAB Code Optimierung durch Vektorisierung" nachgelesen werden können.

Durch die Vektorisierung eines Teils des Loops, welcher in der Funktion `cepstral_dist_frmt_bank` über die Gitterpunkte iteriert, kann die Laufzeit von `cepstral_dist_frmt_bank` auf 45% verringert werden, was einer Reduktion auf durchschnittlich 17.2s entspricht. Der Code ist also, verglichen mit Schritt 1 um durchschnittlich weitere 9.8s schneller geworden (siehe Fig. 14).



Figur 14: *Laufzeitvergleich: Originalprogramm vs. erste und zweite Optimierungsstufe*

2.5.3 Schritt 3: Weitere Vektorisierung mit zusätzlichem Rechenaufwand

Vektorisierung kann in gewissen Fällen, wie in Schritt 2 gezeigt, auf einfache Art und Weise für das Entfernen von Loops verwendet werden. Es gibt jedoch auch Schleifen, welche sich nicht oder nur mit zusätzlichem Rechenaufwand vektorisieren lassen.

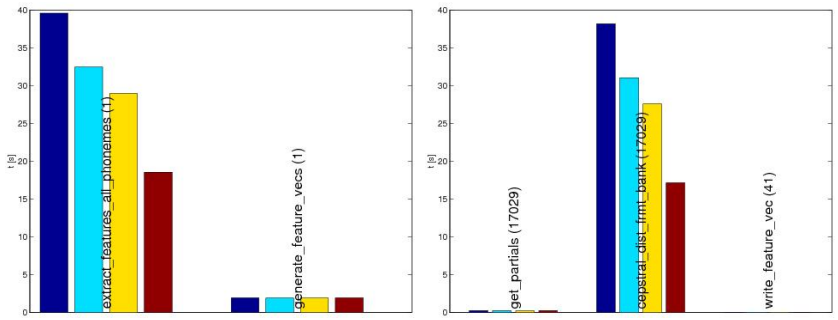
In der Optimierung von `cepstral_dist_frmt_bank` kann man einen solchen Fall beobachten: um das Generieren des Formant Templates, welches für jeden Gitterpunkt berechnet wird, aus dem Loop zu entfernen und mit Vektoroperationen zu realisieren, muss man das Template jeweils für die ganze Länge des Pitchsynchronen Spektrums berechnen, obwohl man die Werte eigentlich nur innerhalb des Bereichs benötigen würde, wo auch tatsächlich die cepstrale Distanz zwischen pitch-synchronem Spektrum und Template berechnet wird.

Durch dieses Vorgehen berechnet man zwar die Werte der Templatefunktion an viel zu vielen Stellen und extrahiert anschliessend innerhalb des restlichen Loops nur die tatsächlich verwendeten Werte mit entsprechender Vektorindizierung, spart aber trotzdem Zeit, da man die Berechnung in einem Schritt für alle Gitterpunkte vektoriell durchführen kann.

Die Umsetzung dieser Technik führt zu einer weiteren Reduktion der Laufzeit auf 31.9% beziehungsweise durchschnittlich 12.2s. Verglichen mit der zweiten Optimierungsstufe ist dies ein zusätzlicher Zeitgewinn von durchschnittlich 5.0s (siehe Fig. 15).

2.5.4 Schritt 4: Partielle Inplementation als MEX Funktion

C-MEX Funktionen (ein Beispiel wird in Fig. 16 gegeben) werden in C auf einem speziellen API geschrieben und anschliessend mit dem MATLAB Command `mex` kompiliert. Die so geschriebenen und für eine spezielle Plattform kompilierten Libraries können aus MATLAB wie normale, als `.m`-Files definierte Funktionen verwendet



Figur 15: *Laufzeitvergleich: Originalversion und Optimierungsstufen 1 bis 3*

werden. Das API unterstützt das Einlesen der an die Funktion übergebenen Parameter, sowie die Rückgabe der Resultate und definiert C Datentypen, welche mit denjenigen von MATLAB konsistent sind. Ferner bietet das API die Möglichkeit, aus der C-MEX Funktion andere MATLAB Funktionen aufzurufen. Da die C-MEX Funktionen plattformspezifisch kompiliert werden und folglich nicht von MATLAB interpretiert werden müssen, wie dies bei herkömmlichen Funktionen der Fall ist, sind sie in der Regel deutlich schneller.

```
>> v = crand(2000,2000);
>>
>> tic;
>>   s1 = sum(sum(real(v)));
>> toc;

elapsed_time =

    0.3438


>> tic;
>>   s2 = sum_sum_real_mex(v);
>> toc;

elapsed_time =

    0.1176
```

Wie man an diesem Beispiel sieht, kann in diesem Fall die C-MEX Funktion sogar die MATLAB-internen Vektorfunktionen bezüglich Effizienz schlagen. Viel deutlicher wird aber der Laufzeitunterschied erst, wenn man nicht vektorisierbare MATLAB Loops durch C-MEX Funktionen ersetzt.

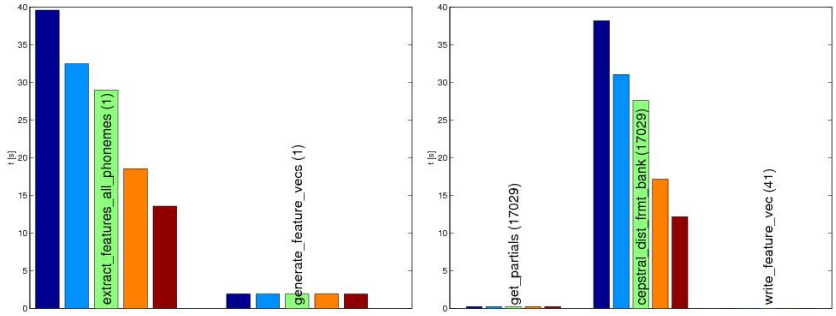
Für eine Einführung in die Programmierung von MEX Funktionen, sowie das dazugehörige API, möchten wir auf die entsprechende MathWorks Webpage[9] verweisen.

[9] `http://www.mathworks.com/access/helpdesk/help/techdoc/apiref/apiref.html`

```
// INCLUDES //
#include "mex.h"

// MEX FUNCTION //
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
  // GENERAL PURPOSE VARIABLES //
  int i;
  int n;
  double sum;

  // POINTERS FOR INPUT AND RESULT //
  double *in;
  double *out;

  // allocate memory for the result
  plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);

  // get the pointer to the real part of the input
  in  = mxGetPr(prhs[0]);

  // get the pointer to the result ([1x1] maxtix)
  out = mxGetPr(plhs[0]);

  // get the number of elements
  n   = mxGetN(prhs[0]) * mxGetM(prhs[0]);

  // initialise the sum
  sum = 0.0;

  for(i=0; i<n; i++)
  {
    // for all elements in the vector or matrix
    // (doesnt make a difference for C-MEX)
    sum += in[i];
  }
  *out = sum;
}
```

Figur 16: *Die C-MEX Funktion sum_sum_real_mex.c aus dem Beispiel in Schritt 5 kann durchaus mit einer Anwendung der entsprechenden Vektorfunktionen von MATLAB konkurrieren.*

Durch Ersetzen des restlichen, nicht weiter vektorisierbaren Teils des Loops in der Funktion `cepstral_dist_frmt_bank` durch eine C-MEX Funktion kann die Laufzeit der Funktion auf 15.4% oder durchschnittlich 5.9s reduziert werden. Die Laufzeit hat sich also, verglichen mit Schritt 3 um durchschnittlich weitere 6.3s verringert (siehe Fig. 17).

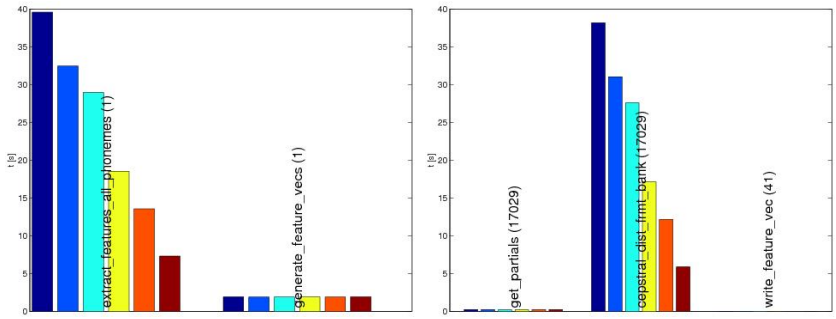


Figur 17: *Laufzeitvergleich: Das Originalprogramm im Vergleich mit den ersten vier Optimierungsstufen.*

2.5.5 Schritt 5: Vollständige Implementation als MEX Funktion

Die guten Resultate aus Schritt 4 führen zu der Idee, dass eine vollständige C-MEX Implementation der Funktion `cepstral_dist_frmt_bank` eine weitere Verbesserung bringen könnte. Um eine Implementation der inversen Fouriertransformation zu umgehen und stattdessen die MATLAB-interne Funktion `ifft` zu verwenden, kann die Funktion `MexCallMATLAB` aus `mex.h` verwendet werden, welche es ermöglicht, aus der C-MEX Funktion MATLAB Funktionen aufzurufen. Ferner wird die Adaption der Formant Bank auf den Grundton des Sprechers mittels `MexCallMATLAB` auf bestehende `.m`-Funktionen realisiert, da sie für spätere allfällige Veränderungen einfach modifizierbar bleiben soll.

Trotz des Overheads, welcher durch den Aufruf der MATLAB-internen `ifft` Funktion entsteht, kann durch die Implementation der ganzen Funktion `cepstral_dist_frmt_bank` als C-MEX Funktion die durchschnittliche Laufzeit auf 3.9% oder von ursprünglich 38.2s auf 1.5s verbessert werden. Dies entspricht verglichen mit Schritt 4 einer weiteren Ersparnis von 4.4s (siehe Fig. 18).



Figur 18: *Die Laufzeit der Originalversion im Vergleich mit allen durchgeführten Optimierungsschritten*

2.6 Funktionstest und Outputvergleich

Es bleibt nun zu zeigen, dass die optimierte Version der Funktion `cepstral_dist_frmt_bank` nicht nur wesentlich schneller ist, sondern auch den selben Output liefert, wie die ursprüngliche Version. Diese Aufgabe scheint auf den ersten Blick trivial, da die gesamte Prozedur der Feature Extraktion Files generiert, welche auf einfache Art[10] miteinander verglichen werden können. Das Problem dabei ist, dass die Outputs der Funktionen leicht voneinander abweichen, da durch die Berechnungen innerhalb von C und den Einsatz gewisser Funktionen aus der Library `math.h` kleine Abweichungen entstehen.

Um die Grösse dieser Abweichungen abschätzen zu können, kann unter Zuhilfenahme der ursprünglichen Funktion eine automatische Berechnung des kumulativen, relativen Fehlers über alle Scorevektoren durchgeführt werden:

$$E_{rel,cumulative} = \sum_{\substack{s \in \\ signals}} \sum_{\substack{frames \\ f \in s}} \frac{|v_{f,orig} - v_{f,new}|}{|v_{f,orig}|}$$

wobei $v_{f,orig}$ der Output der Originalversion und $v_{f,new}$ der Output der neuen Version von `cepstral_dist_frmt_bank` für ein Frame f sei.

Diese Berechnung liefert für die vorliegenden 40 Testdateien lediglich einen Wert von $1.9 \cdot 10^{-6}$, was im Einverständnis mit unserer Betreuerin U. Glavitsch als genügend genau erachtet werden kann.

2.7 Erreichte Verbesserung

Wir haben in diesem Kapitel gezeigt, wie durch die Anwendung von anfangs einfachen Optimierungsmethoden über kompliziertere Techniken wie Vektorisierung bis hin zu einer kompletten Neuimplementation als C-MEX Funktion die durchschnittliche Laufzeit der Funktion `cepstral_dist_frmt_bank` auf 3.9% verringert werden kann. Wir haben bisher jedoch in unserer Betrachtung vernachlässigt, dass diese Funktion nur einen Teil des Feature Extraktionsverfahrens ausmacht.

Tabelle 1 zeigt zusammenfassend die angewendeten Optimierungstechniken und die entsprechenden Verbesserungen, sowohl für die isoliert betrachtete Funktion `cepstral_dist_frmt_bank`, als auch für das ganze Programm. Die gesamte Laufzeit wurde von durchschnittlich 41.5s auf 4.8s vermindert. Hochgerechnet auf die circa 50'000 Dateien, welche für ein vollständiges Erkennungsexperiment verwendet werden, bedeutet dies eine Reduktion von 24 auf $2\frac{3}{4}$ Tage.

[10] Beispielsweise mit dem Unix Tool diff

Optimierungstechnik(en)	**∅ Laufzeit** `cepstral-dist_frmt_bank`	**∅ Laufzeit gesamtes Verfahren**	**∅ Verbesserung bezgl. vorheriger Stufe**
Keine	38.2	41.5	-
Inlining und Loop Unrolling	27.0	30.3	11.2
Vektorisierung	17.2	20.5	9.8
Vektorisierung (II)	12.2	15.5	5.0
Part. Impl. als C-MEX	5.9	9.2	6.3
Vollst. Impl. als C-MEX	1.5	4.8	4.4

Tabelle 1: *Zusammenfassung der Optimierungsschritte und der damit erreichten Laufzeitreduktion für cepstral_dist_frmt_bank und für das gesamte Verfahren. Die Zahlen verstehen sich in Sekunden pro Signaldatei, gerechnet auf einer Sun Blade 100.*

Da das komplette Training aller statistischen Modelle auf den zur Verfügung stehenden Maschinen ungefähr drei Tage dauert, ist die Extraktion der Features aus den Signaldateien nun nicht mehr der Bottleneck in einem Erkennungsexperiment. Obwohl man in der C-MEX Funktion sicherlich noch auf C basierende Optimierungen vornehmen könnte, haben wir die Optimierung eingestellt, da das Ziel, Erkennungsexperimente innerhalb nützlicher Frist durchführen zu können, durch die vorliegende Laufzeit bereits erreicht ist.

3 Visualisierung des Verfahrens

3.1 Motivation

Nachdem das Programm zur Merkmalsextraktion eine viel geringere Laufzeit aufweist, sind wir nun im Stande Experimente mit verschiedenen Parametereinstellungen durchzuführen um die Erkennungsrate des Verfahrens zu erhöhen. Um diese Parameter aber nicht willkürlich zu wählen, sondern aufgrund bestimmter Annahmen und Beobachtungen, versuchen wir einige Parameter des Verfahrens zu visualisieren um daraus einen Anhaltspunkt für die Wahl einer sinnvollen Anfangskonfiguration zu erhalten.

3.2 Parameter des Verfahrens

In dem Merkmalsextraktionsprogramm können folgende Parameter eingestellt werden:

- Anzahl der Gitterpunkte
- Bandbreiten der Templatefunktion
- Art der Gitterpunktanpassung an den Sprecher
- Lage und Anzahl der Toleranzbereiche
- Länge des maximalen Bereichs zur Berechnung des Score

Von Bedeutung für die hier beschriebene Visualisierung sind hier allerdings nur die Anzahl der Gitterpunkte und die Bandbreiten der Templatefunktion, da deren Einfluss am leichtesten von Auge abzuschätzen ist. Wir gehen grundsätzlich davon aus, dass die Anzahl der Gitterpunkte, also die Dichte des Rasters auf der Frequenzachse, den grössten Einfluss auf die Lokalisierung der Formanten hat.

Die Bandbreiten der Templatefunktion liegen, wie auch die Positionen der Gitterpunkte, in einem File vor, welches als Formant Bank bezeichnet wird. Durch das Variieren der Bandbreiten erreicht man, dass die Templatefunktion pro Gitterpunkt mehrere Formen[11] annimmt und sich somit besser auf verschiedene Ausprägungen von Formanten anpasst. Die Schwierigkeit liegt aber darin, diese Bandbreiten nicht zufällig zu wählen, sondern anhand von bestimmten Erfahrungswerten, da sowohl die Anzahl Gitterpunkte wie auch die Anzahl Bandbreiten pro Gitterpunkt einen direkten Einfluss auf die Laufzeit der Feature Extraktion haben[12].

Die Art der Gitterpunktanpassung besteht nur aus zwei gegebenen Verfahren [8] deren Einfluss auf die Erkennungsrate später mit je einem Erkennungsexperiment ausgewertet werden kann. Eine Visualisierung ist daher nicht notwendig.

Die Lage und Anzahl der Toleranzbereiche liegen ebenfalls in einem File vor. Dieser Parameter wird vorerst als fix betrachtet, da es als sinnvoll erachtet wird zuerst eine gute

[11] Das Verändern der Bandbreite entspricht von der Idee her einer Streckung des Template in Richtung der Frequenzachse

[12] Die Laufzeit verhält sich proportional zur Anzahl Bandbreiten und zur Anzahl Gitterpunkte

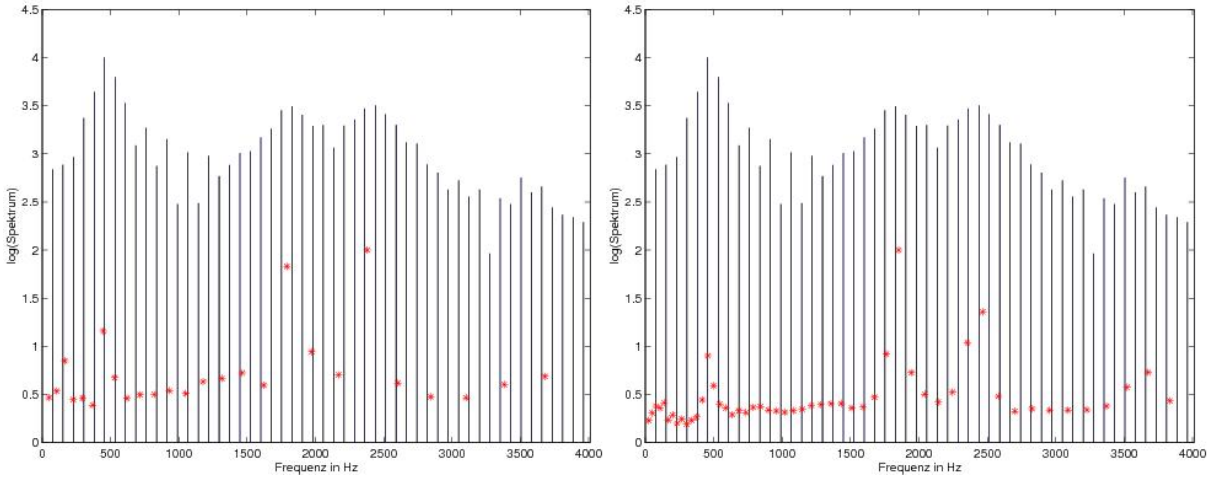
Einstellung der anderen Parameter zu finden um dann in einem letzten Schritt eventuell die Toleranzbereiche zu verändern. Eine Anpassung der Toleranzbereiche würde zudem eine statistische Auswertung von Formantpositionen erfordern.

Beim Vergleich des Spektrums mit der Templatefunktion lässt sich bestimmen, wie breit der Bereich zur Berechnung des Score Werts maximal sein soll (siehe Fig. 4). Je breiter dieser Bereich ist, desto mehr Spektralwerte unter der Templatefunktion werden benutzt um die cepstrale Distanz zu berechnen. Dies kann aber dazu führen, dass der Score an manchen Stellen trotz gutem Matching verschlechtert wird, da die Distanz umso grösser wird, je mehr Werte zu deren Berechnung verwendet werden.

Für die visuelle Auswertung der Parameter verwenden wir lediglich Vokale, da sie besonders ausgeprägte Formanten aufweisen und dadurch von Auge gut erkannt werden können. Die dazu benötigten Signaldateien werden generiert, indem man aus den Signalen ganzer Wörter die jeweiligen Bereiche, an denen Vokale gesprochen werden, ausschneidet. Auf diese Art erzeugte Signaldateien von Vokalen liegen uns bereits vor.

3.3 Einfluss der Anzahl Gitterpunkte

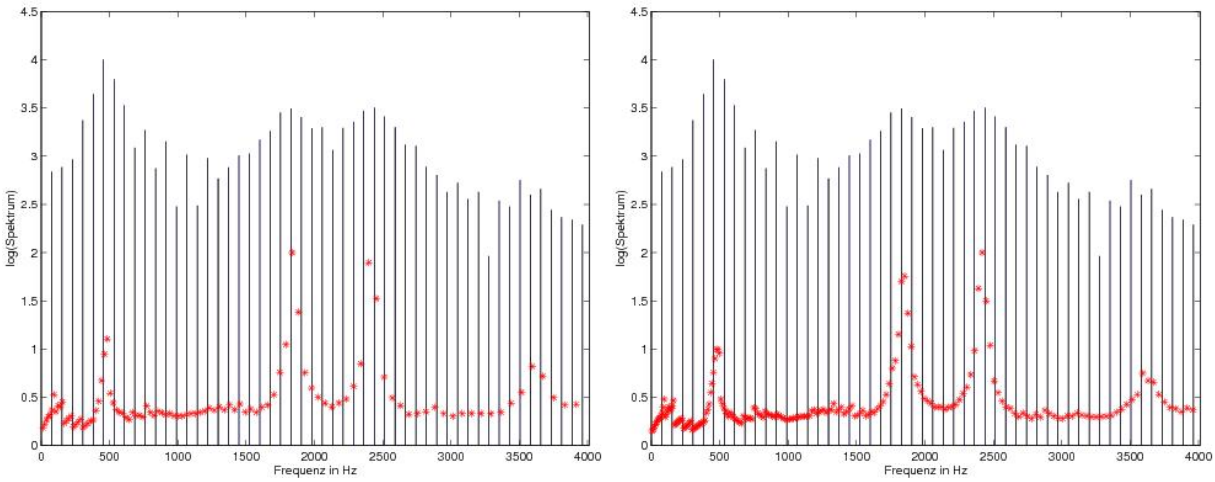
Für jedes File kann man alle Frames mit den berechneten Score Werten pro Gitterpunkt plotten und für 26, 52, 104 und 208 Gitterpunkte miteinander vergleichen:



Figur 19: *Erhöhung von 26 auf 52 Gitterpunkte führt zu einer genaueren Lokalisierung der Formanten*

Wie man in Fig. 19 sehen kann, werden die Formanten mit 52 Gitterpunkten genauer lokalisiert als mit 26. Noch präziser erscheint die Lokalisierung mit 104 Gitterpunkten. Da man bei erneuter Erhöhung der Anzahl Gitterpunkte auf 208 von Auge kaum noch eine Veränderung der Verteilung der Score Werte wahrnimmt, haben wir uns dazu entschieden 104 Gitterpunkte als gute Anfangskonfiguration zu wählen (siehe Fig. 20).

Vor allem bei Sprechern mit hohem Grundton (hauptsächlich Frauen) wird angenommen, dass eine erhöhte Präzision bei der Lokalisierung der Formanten durchaus einen grossen Einfluss auf die Qualität der Feature Extraktion hat.

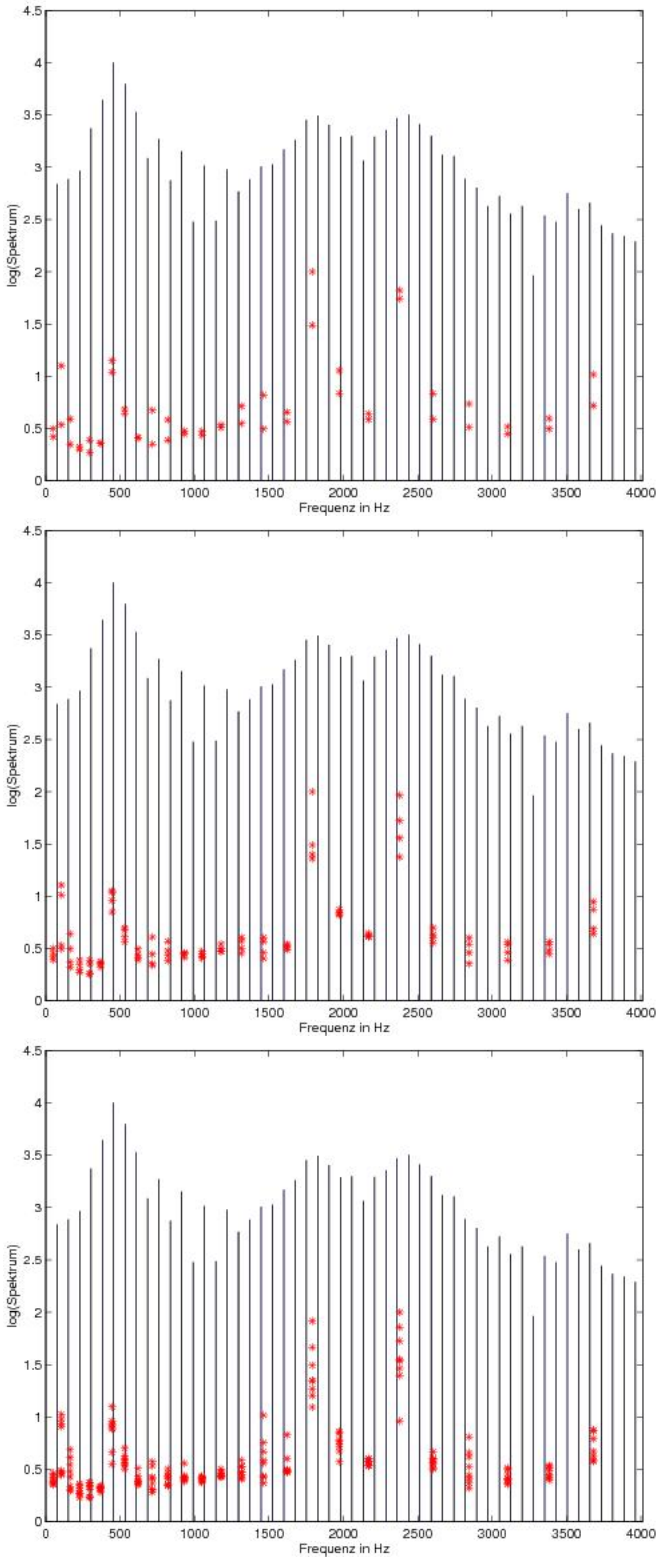


Figur 20: *Erhöhung von 104 auf 208 Gitterpunkte bringt keine wesentliche Verbesserung mehr*

3.4 Einfluss der Bandbreiten der Templatefunktion

Wenn man pro Gitterpunkt mehrere Bandbreiten verwendet, erhöht man die Wahrscheinlichkeit, dass ausgeprägte Formanten einen höheren Score Wert erhalten. Gleichzeitig nehmen wir an, dass sich an Stellen wo sich kein Formant befindet auch mit mehreren Bandbreiten kein wesentlich besserer Score ergibt.

Bei 2,4 und 8 verschiedenen Bandbreiten pro Gitterpunkt kann man in Fig. 21 erkennen, dass sich die Score Werte bei verschiedenen Formantenpositionen verbessern. Daher vermuten wir, dass eine Variation der Bandbreiten ebenfalls zu besseren Resultaten führen kann.



Figur 21: *2,4 und 8 verschiedene Bandbreiten pro Gitterpunkt*

4 Erstellung des Testenvironment

4.1 Vorgegebenes Environment

Das bisherige Erkennungssystem [1] besteht im Wesentlichen aus dem NoiseTrain Programm, welches verschiedene Hidden-Markov Modellarten trainieren kann. Standardmässig werden mit dem MFCC Verfahren aus allen Sprachsignalen auf den SpeechDat(II) CD-ROMs Merkmalsvektoren berechnet und diese werden dann zum trainieren verwendet. Welche Dateien zum trainieren und welche zum testen gedacht sind wird in Index-Dateien auf den CDROMs spezifiziert.
Zum Testen stehen verschiedene weitere Skripts zur Verfügung, welche ebenfalls mit dem MFCC Verfahren Merkmalsvektoren generieren und anschliessend versuchen, diese mit den trainierten Modellen zu erkennen. Die Testskripts unterscheiden sich lediglich im Inhalt der Sprachsignale die sie testen (siehe Tabelle 2). Als Output liefern sie eine Auflistung der Wortfehlerrate (*WER: Word Error Rate*) pro Modellart.
Das Erkennungssystem akzeptiert auch Merkmalsvektoren, die mit einer anderen Methode berechnet wurden, sofern diese ein vorgeschriebenes Dateiformat aufweisen [9]. Liegen in den entsprechenden Verzeichnissen bereits Merkmalsvektoren vor, so werden diese nicht durch die Skripts überschrieben.

Test-Name	**Inhalt des Tests**
mviptest_O	Ortschaftsnamen (city names)
mviptest_W	Worte mit vielen phonetischen Übergängen (phonetically rich words)
sviptest_A	30 einzelne Worte (application words)
sviptest_I	Einzelne Ziffern (isolated digits)
sviptest_Q	Ja / Nein Antworten (questions)

Tabelle 2: *Namen und jeweiliger Inhalt der einzelnen Erkennungstests*

4.2 Problemstellung

Da die Erkennungsexperimente bisher mit der ganzen SpeechDat(II) Datenbank und mit allen Hidden-Markov Modellen durchgeführt wurden, war es unmöglich verschiedene Parametereinstellungen auszutesten ohne dass man 2-3 Tage dafür aufwenden musste. Unsere Aufgabe besteht nun darin, eine Umgebung einzurichten, in der man in nützlicher Frist ein komplettes Erkennungsexperiment durchführen kann. Die Merkmalsextraktion, das Training der Modelle und das Testen mit den Testdaten soll innerhalb eines Tages durchgeführt werden können, um im zeitlichen Rahmen der Semesterarbeit effizient Parametereinstellungen zu testen und zu analysieren.

Damit die Testumgebung auch über die Semesterarbeit hinweg von Nutzen ist, liegt ein weiterer Schwerpunkt darin, diese so komfortabel wie möglich hinsichtlich der Übergabe von Parametern zu gestalten.

4.3 Adaption des Referenzerkennungssystems

Um die in der Problemstellung erwähnten Ziele zu erreichen gehen wir in 3 Schritten vor:

1. Einarbeiten in die vorhandenen Skripte des Erkennungssystems
2. Reduktion der verwendeten Datenmenge und Anzahl von Modellarten
3. Adaption des MATLAB Feature Extraktionsprogramms an neue Testumgebung

4.3.1 Reduktion der Datenmenge

Da das Erkennungssystem die Daten aus einem Verzeichnis ausliest, in das alle CDROMs der SpeechDat(II) Datenbank gelinkt werden, kann man die Datenmenge am einfachsten reduzieren indem man nur eine Teilmenge der Dateien in dieses Verzeichnis linkt. Das Perl Skript das diesem Zweck dient wird im Anhang genauer beschrieben. Es ergeben sich aber folgende zwei Schwierigkeiten: Erstens braucht das Erkennungssystem eine bestimmte Grundmenge an Daten um die verwendeten Modellarten zu initialisieren. Dieses sogenannte Bootset[13] muss also beibehalten werden, da das Skript ansonsten das Training der Modelle frühzeitig abbricht. Zweitens verwenden die Testprogramme individuelle Testdaten, welche ebenfalls hinzugelinkt werden müssen. Alles in allem verwenden wir also für unsere Erkennungsdurchläufe ca 22000 Dateien von den bisher ca 50000. Diese Reduktion um grob die Hälfte erscheint gering, hat aber den Vorteil, dass die Modelle mit genügend Daten trainiert werden und die Testresultate dadurch aussagekräftiger werden.

4.3.2 Reduktion der Modellarten

Da die verschiedenen Modellarten nacheinander im Skript ausgeführt werden kann man relativ komfortabel diejenigen Modellarten auskommentieren die man nicht trainieren will[14]. Dabei bleibt das Skript lauffähig und die Testprogramme verwenden automatisch nur die trainierten Modellarten.

4.3.3 Adaption des MATLAB Programms zur Feature Extraktion

Da das Merkmalsextraktionsprogramm eine Dateiliste mit allen zu bearbeitenden Dateien benötigt, ist es am einfachsten, wenn man aus dem Verzeichnis, welches das reduzierte Datenset beinhaltet, eine Dateiliste generiert. In dieser Liste sind alle Dateien enthalten welche in späteren Trainings- und Testprozessen benötigt werden.

Da jedoch die Score Werte für jedes Sprachsignal in einem File zwischengespeichert werden ergibt sich ein Speicherplatzproblem. Je nach Anzahl der Gitterpunkte und Bandbreiten kann der Speicherplatzbedarf auch für das reduzierte Datenset sehr gross werden. Das MATLAB Programm kann aber so modifiziert werden, dass nur noch die Merkmalsvektoren in den entsprechenden Dateien abgespeichert werden, was erheblich weniger Speicherplatz beansprucht.

[13] das Bootset enthält ca 10000 Dateien

[14] In unserem Fall sind das die komplexen Modellarten ab 8 mixtures

Nach diesen Änderungen ist die Testumgebung einsatzbereit, wobei die Dauer eines Erkennungsexperiments (inkl. Merkmalsextraktion) statt wie bisher ca 2 Wochen nur etwa einen Tag beträgt. Für weitere kleinere Anpassungen, wie z.B. die parallele Ausführung des Extraktionsprogramms werden diverse Hilfsskripts verwendet, welche im Anhang B noch genauer erläutert werden.

5 Durchführung und Auswertung einiger Erkennungsexperimente

Mit der funktionierenden Testumgebung wird nun mit verschiedenen Parametereinstellungen experimentiert. Für einige Experimente möchten wir eine kurze Begründung für die Parametereinstellungen, sowie eine Auswertung der Resultate präsentieren. Wir verzichten aber bewusst auf Spekulationen über die Gründe weshalb gewisse Einstellungen bezüglich der Erkennungsrate besser oder schlechter sind als andere. Pro Einstellung werden jeweils die Wortfehlerraten pro verwendete Modellart von allen Testprogrammen (siehe Tabelle 2) aufgeführt.

5.1 Standardverfahren MFCC als Vergleichsbasis

Um die Qualität der Parametereinstellungen zu beurteilen vergleichen wir die Erkennungsraten mit denen des Referenzverfahrens MFCC.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	17.45	28.46	6.90	14.43	0.87
mono_4_2	15.96	27.61	7.09	12.37	0.87
tied_4_1	8.51	13.93	1.55	3.92	0.65
tied_4_2	7.66	13.08	0.97	2.68	0.65

Tabelle 3: *MFCC Verfahren*

Dieses Verfahren wird in [3] erklärt und hat sich als Feature Extraktionsverfahren in den letzten Jahren am besten bewährt. Das Ziel ist es, mit der Formanten-basierten Featureextraktion eine bessere Erkennungsrate zu erzielen als mit diesem Verfahren. Da die menschliche Sprachwahrnehmung auf der Detektion der Formanten basiert und das MFCC Verfahren diese Information nicht speziell berücksichtigt, liegt die Annahme auf der Hand, dass das Formanten-basierte Verfahren zu besseren Ergebnissen führt.

5.2 Bisherige Konfiguration der Parameter

In diesem Experiment verwenden wir die selben Parametereinstellungen, die bereits für ein früheres Experiment von Frau Glavitsch verwendet wurden. Mit dieser Einstellung liegen zwar schon Resultate vor, diese beziehen sich jedoch auf die ganze SpeechDat(II) Datenbank. Um beurteilen zu können wie sich Änderungen an den Parametern auswirken, brauchen wir aber Resultate die mit unserem reduzierten Trainingsset entstanden sind.

Ein Vergleich mit den Resultaten in Tabelle 3 zeigt, dass die Wortfehlerraten beim Referenzverfahren MFCC deutlich geringer sind. Es wäre also eine deutliche Steigerung nötig, um mit besseren Parametereinstellungen ähnliche Resultate zu erreichen. Diese Tatsache überrascht aber nicht, da die bisherige Konfiguration der Parameter schon in früheren Experimenten mit allen Daten und Modellarten nicht zu wesentlich besseren Resultaten geführt hat.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	26.17	39.30	11.47	15.26	1.74
mono_4_2	25.11	38.14	10.98	14.23	1.30
tied_4_1	15.57	21.35	4.57	5.57	1.52
tied_4_2	15.53	21.03	4.57	5.98	1.84

Tabelle 4: *Formant-basierte Methode, 26 Gitterpunkte, eine Bandbreite pro Gitterpunkt*

5.3 Erhöhung der Anzahl Gitterpunkte auf 104

Das erste Erkennungsexperiment mit neuen Parametereinstellungen besteht aus der Erhöhung der Anzahl Gitterpunkte. Durch die Vervierfachung der Anzahl Gitterpunkte erhofft man sich eine exaktere Lokalisierung der Formanten, die sich in einer besseren Erkennungsrate äussert. Falls dies zutrifft, dann kann man sich überlegen das Gitterpunktraster noch enger zu wählen, andernfalls ist gezeigt, dass dieser Parameter einen sekundären Einfluss auf die Erkennungsrate hat und die anderen Parameter wie z.B. Bandbreiten der Templatefunktion relevanter sind.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	26.17	42.37	10.98	15.46	1.95
mono_4_2	25.96	41.37	11.08	14.43	1.41
tied_4_1	19.57	23.78	6.71	6.19	1.30
tied_4_2	17.66	23.15	6.80	5.57	1.30

Tabelle 5: *Formant-basierte Methode, 104 Gitterpunkte, eine Bandbreite pro Gitterpunkt*

Offensichtlich führt eine deutliche Erhöhung der Anzahl Gitterpunkte bei keinem Testprogramm zu verbesserten Resultaten. Die Werte liegen im Vergleich zu den Werten in Tabelle 4 im Durchschnitt etwa 2 Prozentpunkte höher. Eine erhöhte Präzision bei der Lokalisierung von Formanten scheint also nicht den angenommenen positiven Effekt zu haben.

5.4 Mit Isophone Anpassung

Von Interesse ist ebenfalls der Einfluss der Gitterpunktanpassung an den Sprecher. Zweck dieser Anpassung ist es, das Raster der Gitterpunkte je nach Grundton des Sprechers anzupassen. Daher verwenden wir die uns zur Verfügung stehende alternative Isophone Anpassung, um zu sehen, ob dadurch bessere Resultate entstehen.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	26.38	41.95	14.29	18.76	2.60
mono_4_2	26.17	41.15	13.02	16.70	2.17
tied_4_1	14.68	24.21	6.12	6.60	2.39
tied_4_2	14.47	23.68	6.03	6.80	2.06

Tabelle 6: *Formant-basierte Methode, Isophone Anpassung mit 26 Gitterpunkten, eine Bandbreite pro Gitterpunkt*

Leider bringt auch diese alternative Anpassung keine wesentliche Verbesserung, unterscheidet sich aber in den Werten nicht allzu stark vom ursprünglichen Verfahren.

5.5 Mit 4 verschiedenen Bandbreiten

In diesem Experiment definieren wir in der Formant Bank 4 verschiedene Bandbreiten pro Gitterpunkt. Das hat zur Folge, dass pro Gitterpunkt, also pro Frequenz in der Formant Bank, mehrere Scores berechnet werden. Gleichzeitig werden 26 Gitterpunkte und die Isophone Anpassung verwendet.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	27.23	43.96	13.31	19.38	2.82
mono_4_2	25.74	42.48	13.12	19.18	2.39
tied_4_1	17.87	25.37	7.87	5.98	2.39
tied_4_2	17.02	24.84	7.77	5.77	1.95

Tabelle 7: *Formant-basierte Methode, 4 Bandbreiten pro Gitterpunkt, 26 Gitterpunkte*

Da 4 Bandbreiten pro Gitterpunkt keine Verbesserung mit sich bringt, gehen wir davon aus, dass sich auch eine noch stärkere Variation der Bandbreiten nicht positiv auf die Erkennungsrate auswirkt.

5.6 Verkleinerung des Bereichs zur Berechnung der Score Werte

Ein weiterer Parameter des Programms ist die maximale Breite des Bereichs, der zur Berechnung der cepstralen Distanz und somit des Score Werts gebraucht wird. Dieses Experiment verwendet 26 Gitterpunkte, wie das Experiment mit variablen Bandbreiten ebenfalls.

	mviptest_O	mviptest_W	sviptest_A	sviptest_I	sviptest_Q
mono_4_1	23.40	38.24	10.79	16.91	0.76
mono_4_2	22.34	37.08	10.40	16.08	0.87
tied_4_1	14.68	21.19	4.08	5.15	1.74
tied_4_2	14.47	21.50	4.37	4.54	1.41

Tabelle 8: *Formant-basierte Methode, 26 Gitterpunkte, Bereich für cepstrale Distanz verkleinert*

Eine Verkleinerung dieses Bereichs hat eindeutig eine bessere Erkennungsrate zur Folge. Eine mögliche Begründung dafür haben wir bereits in 3.2 bei der Beschreibung dieses Parameters gegeben.

6 Schlussfolgerung

In diesem Kapitel wollen wir als Rückblick auf diese Semesterarbeit die wichtigsten Schlussfolgerungen aufführen.

6.1 Performance Optimierung

Im ersten Teil, der Performance Optimierung, ging es ursprünglich darum, ein vorgegebenes MATLAB Feature Extraktions Programm komplett neu zu implementieren. Wie sich durch detaillierte Analyse der Komponenten des Programms herausgestellt hat, war es sinnvoller, lediglich die zeitintensivste Funktion durch Übersetzung in eine C-MEX Funktion zu optimieren. Dies hat den Vorteil, dass das ursprüngliche Programm in seiner Struktur erhalten blieb und künftig bezüglich Wartbarkeit keinen Mehraufwand generiert. Durch die Optimierung konnte ein Performance Gewinn erreicht werden, der eine komplette Neuimplementation nicht mehr erforderlich macht, so dass auch die damit verbundenen Probleme nicht mehr entstehen. Eine Erweiterung des Programms um neue Komponenten ist beispielsweise in MATLAB einfacher zu erreichen als in C.

Rückblickend lässt sich sagen, dass wir die im ersten Teil gestellten Anforderungen erfüllen konnten.

6.2 Testumgebung und Parameteroptimierung

Bei der Einrichtung einer Testumgebung zur Durchführung einiger Experimente sind leider aufgrund der Komplexität des NoiseTrain Programms einige Verzögerungen aufgetreten. Dies hat es uns verunmöglicht, genügend viele Erkennungsexperimente mit der neuen Testumgebung durchzuführen um die Parameter zu optimieren und mit dem neuen Merkmalsextraktionssystem deutlich bessere Resultate zu erzielen, als diese mit MFCC erreicht werden.

Es ist uns jedoch gelungen, eine Umgebung für ein kleineres Erkennungssystem einzurichten und für weitere Experimente produktiv nutzbar zu machen. Dadurch haben wir eine Basis für weitere, schnell durchführbare Experimente geschaffen, welche mit Sicherheit zu einer deutlichen Verbesserung der Erkennungsrate der Formant-basierten Feature Extraktion führen werden.

7 Ausblick

In unserer Arbeit haben wir erreicht, dass innerhalb kurzer Zeit mit dem von U. Glavitsch entwickelten Feature Extraktions Verfahren Erkennungsversuche auf der SpeechDat(II) Datenbank durchgeführt werden können. Optimale Einstellungen für die Parameter des Verfahrens konnten wir leider aus Zeitgründen nicht finden.

In einem weiteren Schritt muss nun mit einer grossen Anzahl von Parametereinstellungen versucht werden, die Erkennungrate des Verfahrens zu maximieren. Ein erster Schritt dazu ist sicherlich, wie wir in Kapitel 3.2 im vierten Absatz beschrieben haben, eine statistische Analyse der Formantbereiche über einer grossen Anzahl von Sprechern, sowie eine statistische Auswertung der typischen Bandbreite der Templatefunktion in Abhängigkeit der Frequenz, da das Experimentieren mit den restlichen Parametern keinen wesentlichen Einfluss auf die Erkennungsrate zu haben scheint[15].

Wird mit dem Formant-basierten Verfahren eine signifikant bessere Erkennungsrate erreicht als mit dem MFCC Verfahren, so bleibt zu zeigen, dass auch die Laufzeit vergleichbar mit MFCC ist. Wir haben zwar im Rahmen dieser Arbeit eine massive Beschleunigung erreichen können, verglichen mit MFCC ist das Formant-basierte Verfahren jedoch immer noch relativ langsam[16]. Wir sind jedoch davon überzeugt, dass mit einer vollständigen C Implementation des Formant-basierten Verfahrens unter Verwendung von Performance optimierten Mathematik-Libraries eine Laufzeit in der Grössenordnung derjenigen von MFCC erreicht werden kann.

[15] Insofern ist es uns immerhin gelungen zu zeigen, welche Parametereinstellungen *keinen* wesentlichen Einfluss auf die Qualität der Featurevektoren und damit auch auf die Erkennungsrate haben.

[16] MFCC berechnet das Feature einer Signaldatei in durchschnittlich weit unter $\frac{1}{10}s$, das Verfahren ist jedoch auch wesentlich einfacher.

Literatur

[1] B. Lindberg, F. T. Johansen, N. Warakagoda, G. Lehtinen, et al. A Noise Robust Multilingual Reference Recogniser Based on SpeechDat(II). In *Proceedings of the ICSLP'2000*, Beijing (China), October 2000.

[2] B. Pfister und H.-P. Hutter. *Sprachverarbeitung I.* Vorlesungsskript für das Wintersemester 2003/2004, Departement ITET, ETH Zürich, 2003.

[3] P. Davis, S. B. Mermelstein. Comparison of Parametric Representations for Monosyllabic WordRecognition in Continuously Spoken Sentences. *IEEE Trans. on ASSP*, 1980.

[4] D. H. Klatt. Software for a cascade/parallel formant synthesizer. *Journal of the Acoustical Society of America*, 67(3):971–995, March 1980.

[5] A. Rausch. *Untersuchungen zur Vokalartikulation im Deutschen*, volume 30 of *IPK Forschungsberichte*, pages 35–82. Helmut Buske Verlag Hamburg, 1972.

[6] Y. F. Hu, R. J. Allan and K. C. F. Maguire. Comparing the performance of java with fortran and c for numerical computing, `http://www.dl.ac.uk/TCSC/UKHEC/JASPA/bench.html`, 2000.

[7] MathWorks. Product support, 1109 - code vectorization guide, `http://www.mathworks.com/support/tech-notes/1100/1109.html`, 2004.

[8] U. Glavitsch. Speaker normalization with respect to f0: a perceptual approach. TIK-Report 185, ETH Zurich, December 2003.

[9] D. Valtchev V. Young, S. Ollason and P. Woodland. The HTK book (for HTK Version 2.1). *Entropic Cambridge Research Laboratory*, March 1997.

A Design Draft der Neuimplementation

Datenstrukturen

```
  ///////////////////////
 // DATA_STRUCTURES.H //
///////////////////////

// DEFINES //

#define MAXPATHLENGTH 512
#define MAXEXTLENGTH 8

// VECTOR DATA STRUCTURE //

struct double_vec
{
  int size;     // size of the vector
  double *cont; // content of the vector (ptr to array)
}


struct short_vec
{
  int size;     // size of the vector
  short *cont   // content of the vector (ptr to array)
}


struct short_vec *creat_short_vec(int size);
// description:
// ------------
//   allocates memory for a short_vec data
//   structure holding a vector of size <size>
//
//   IMPORTANT: must explicitly be disposed with
//   the dispose_short_vec() function, to avoid
//   memory leaks.
//
// return value:
// -------------
//   pointer to a short vector data structure
//   of size <size>, dynamically allocated with
//   malloc() from stdlib.h.
//
// arguments:
// ----------
//   -> size: size of the vector.


struct double_vec *creat_double_vec(struct double_vec *v, int size);
// description:
// ------------
//   allocates memory for a double_vec data
//   structure holding a vector of size <size>
//
//   IMPORTANT: must explicitly be disposed with
//   the dispose_double_vec() function, to avoid
//   memory leaks.
//
// return value:
// -------------
//   pointer to a double vector data structure
//   of size <size>, dynamically allocated with
//   malloc() from stdlib.h.
//
// arguments:
// ----------
//   -> size: size of the vector.


void dispose_short_vec(struct short_vec *v);
// description:
// ------------
//   frees all memory of a previously created
//   short_vec data structure.
//
// return value:
// -------------
//   none
//
// arguments:
// ---------
//   -> v:    short vector, previously created
//            with creat_short_vec().


void dispose_double_vec(struct double_vec *v);
// description:
// ------------
//   frees all memory of a previously created
//   double_vec data structure.
```

```
//
// return value:
// -------------
//   none
//
// arguments:
// ---------
//   -> v:    double vector, previously created
//            with creat_doube_vec().

// END OF VECTOR DATA STRUCTURE //

struct formant_bank_struct
{
  // stores the information given in a
  // formant bank file
  double f0;                // base frequency of the formant bank
  double f1f2_border        // unused at the moment
  double f2f3_border        // unused at the moment
  struct double_vec *freq;  // frequency vector
  struct double_vec *bw;    // bandwidth vector
};


void dispose_formant_bank_struct(struct formant_bank_struct *fb);


struct formant_ranges_struct
{
  // stores the information given in a
  // formant ranges struct
  struct double_vec *lower;   // lower bounds of formant ranges
  struct double_vec *higher;  // higher bounds of formant ranges
};


void dispose_formant_ranges_struct(struct formant_ranges_struct *fr);


struct cfg_struct
{
  // additional config stuff goes here...
  char f0_ext[MAXEXTLENGTH];       // extension of the f0 files
  char f0_norm_ext[MAXEXTLENGTH];  // extension of the f0 files used for normalisation
  char score_ext[MAXEXTLENGTH];    // extension of the score files written by the program
  char feature_ext[MAXEXTLENGTH];  // extension of the feature files written by the program
  int fsG;                         // sampling frequency of the signals
  int interval;                    // shift of a signal window
  int e_included;                  // include mean energy as last component (score- and featurevec)
  int numceps;                     // maximal number of coefficients per feature vector
};


void dispose_cfg_struct(struct cfg_struct *c);


struct score_struct
{
  int nof_frames;                  // number of score vectors generated for the signal
  struct double_vec **cont;        // array of pointers to the score vectors of the signal for each frame
};


void dispose_score_struct(struct score_struct *s);


struct feature_vecs_struct
{
  int nof_frames;                  // number of feature vectors generated for the signal
  struct double_vec **cont;        // array of pointers to the feature vectors of the signal for each frame
};


void dispose_feature_vecs_struct(struct feature_vecs_struct *fv);
```

Interfaces

```
  ////////////////////
 // IO_FUNCTIONS.H //
////////////////////

struct short_vec *alaw2lin(const char *filename);
// description:
// ------------
//   reads an sza file from disk and returns a pointer to a
//   short_vec struct allocated by malloc() from stdlib.h,
//   holding the decompressed signal.
//
//   IMPORTANT: must explicitly be disposed with
//   the dispose_short_vec() function, to avoid
```

```
//   memory leaks.
//
// return value:
// -------------
//   returns a pointer to short_vec struct on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the sza file to convert


struct formant_bank_struct *read_normalized_formant_bank(const char *filename);
// description:
// ------------
//   internalizes a formant bank file from disk into a
//   formant_bank_struct struct using malloc() from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed.
//
// return value:
// -------------
//   returns a pointer to formant_bank_struct struct
//   on success, NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the formant bank file


struct formant_ranges_struct *read_formant_ranges(const char *filename);
// description:
// ------------
//   internalizes a formant ranges file from disk into a
//   formant_ranges_struct struct using malloc()
//   from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed.
//
// return value:
// -------------
//   returns a pointer to formant_ranges_struct on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the formant ranges file


struct cfg_struct *read_config(const char *filename);
// description:
// ------------
//   internalizes a config file from disk into a
//   cfg_struct struct using malloc() from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed.
//
// return value:
// -------------
//   returns a pointer to cfg_struct on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the config file


struct double_vec *read_f0_vec(const char *filename);
// description:
// ------------
//   internalizes a base frequency file from disk into a
//   double_vec struct using malloc() from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed using
//   dispose_double_vec().
//
// return value:
// -------------
//   returns a pointer to double_vec on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the base frequency file


int store_score_information(const char *filename, struct score_struct *score);
// description:
// ------------
//   stores a score_struct struct into a file with
//   name <filename>.
//
// return value:
// -------------
//   returns 1 on success, 0 otherwise.
//
```

```
// arguments:
// ----------
//   -> filename: name of the file for the score struct
//                to be written
//   -> score:    pointer to the score struct to be written


struct score_struct *load_score_information(const char *filename);
// description:
// ------------
//   loads a score information file with the name <filename>
//   and internalizes it into a score_struct struct, which
//   is created using malloc() from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed using
//   dispose_score_struct() to avoid memory leak.
//
// return value:
// -------------
//   returns a pointer to score_struct on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> filename: name of the file for the score struct
//                to be read


int store_feature_vecs(const char *filename, struct feature_vecs_struct *fv);
// description:
// ------------
//   stores a feature_vecs_struct struct into a file
//   with name <filename>.
//
// return value:
// -------------
//   returns 1 on success, 0 otherwise.
//
// arguments:
// ----------
//   -> filename: name of the file for the feature vecs struct
//                to be written
//   -> score:    pointer to the feature vecs struct to be written



  ////////////////////////////////////
 // SIGNAL_PROCESSING_FUNCTIONS.H //
////////////////////////////////////

struct score_struct *extract_score_information(struct short_vec *signal, struct double_vec *f0,
  struct double_vec *f0_norm, struct formant_bank_struct *fb, struct cfg_struct *c);
// description:
// ------------
//   computes the cepstral distances and energies for each
//   grid_point in each frame and returns a score_struct
//   struct allocated dynamically using malloc() from
//   stdlib.h.
//
//   IMPORTANT: must explicitly be disposed using the
//   dispose_score_struct() function.
//
// return value:
// -------------
//   returns a pointer to score_struct on success,
//   NULL otherwise.
//
// arguments:
// ----------
//   -> signal:   pointer to the signal, from which
//                the score information should be
//                extracted
//   -> f0:       pointer to a double vector containing
//                base frequencies of the signal for each
//                interval (to compute length of window)
//   -> f0_norm:  pointer to a double vector containing
//                base values of f0 (for normalisation)
//   -> fb:       pointer to a formant bank struct
//   -> c:        pointer to a configuration struct


struct feature_vecs_struct *generate_feature_vecs(struct score_struct *s, struct formant_bank_struct *fb,
  struct formant_ranges_struct *fr, struct cfg_struct *c);
// description:
// ------------
//   computes the features out of a given score_struct struct
//   and returns the pointer to the corresponding
//   feature_vecs_struct struct, dynamically allocated using
//   malloc() from stdlib.h.
//
//   IMPORTANT: must explicitly be disposed using the
//   dispose_feature_vecs_struct() function.
//
// return value:
// -------------
//   returns a pointer to feature_vecs_struct on success,
```

```
//   NULL otherwise.
//
// arguments:
// ----------
//   -> s:        pointer to a score struct, from which
//                the feature vector should be extracted
//   -> fb:       pointer to a formant bank struct
//   -> fr:       pointer to a formant ranges struct
//   -> c:        pointer to a configuration struct


void transform_filename(char *out, const char *prefix, const char *relative_fn, const char *ext);
  // ext == NULL: dont change extension
  // out is an array of fixed size (#define MAXPATHLENGTH)


struct double_vec *get_partials(struct short_vec *signal, int pos, double f0_value, int fsG, double *partial_distance);
// description:
// ------------
//   computes the spectrum and the distance between the overtones
//   of the actual signal frame.
//
//   allocates a double_vec struct using malloc() from stdlib.h,
//   to return the spectrum.
//
//   IMPORTANT: return value must explicitly be disposed using
//   dispose_double_vec().
//
// return value
// ------------
//   returns the magnitude spectrum (pitch-synchronous) in form
//   of a double_vec pointer on success, NULL otherwise.
//
// arguments
// ---------
//    -> signal:           pointer to signal vector
//    -> pos:              current position of signal window
//    -> f0_value:         base frequency of the signal at position <pos>
//    -> fgG:              sampling frequency of <signal>
//    -> partial_distance: distance between overtones (output)
```

Skelette

```
  ////////////////////////////
 // CONTROLLER_SKELETON.C //
////////////////////////////

// INCLUDES //
#include <stdio.h>
#include <stdlib.h>

// DEFINES //

// MAIN FUNCTION //
int main(int argc, char **argv)
{
  FILE *filelist_fd;               // filelist with relative paths to sza files

  char filename[MAXPATHLENGTH];    // name of the currently processed file
  char tmp[MAXPATHLENGTH];         // temporary file name

  char *sza_file_list;             // filename of sza filelist
  char *sza_prefix;                // path to the sza file directory
  char *f0_prefix;                 // path to the f0 file directory
  char *f0_norm_prefix;            // path to the f0_norm directory
  char *formant_bank_file;         // path to the formant bank file
  char *formant_ranges_file;       // path to the formant ranges file
  char *config_file;               // path to the config file
  char *target_prefix;             // the target directory

  struct formant_bank_struct *fb;  // struct holding the formant bank
  struct formant_ranges_struct *fr; // struct holding the formant ranges information
  struct cfg_struct *conf;         // struct holding the config file information

  struct short_vec *signal;        // the signal of the actual file
  struct double_vec *f0;           // the base frequencies of the actual file
  struct double_vec *f0_norm;      // the base frequencies for normalisation of the act. file

  struct score_struct *score;      // score struct as return value of extract_score_information()
  struct feature_vecs_struct *fv;  // feature vecs struct as return value of generate_feature_vecs()

  if(argc != 9)
  {
    // not enough input parameters specified.
    // print usage information
    fprintf(stderr, "usage: %s <sza_file_list> <sza_prefix> <f0_prefix> <f0_norm_prefix>\n",argv[0]);
    fprintf(stderr, "<formant_bank_file> <formant_ranges_file> <config_file> <target_prefix>\n");
    exit(1);
  }
```

```
// giving the parameters meaningfull names
sza_file_list       = argv[1];
sza_prefix          = argv[2];
f0_prefix           = argv[3];
f0_norm_prefix      = argv[4];
formant_bank_file   = argv[5];
formant_ranges_file = argv[6];
config_file         = argv[7];
target_prefix       = argv[8];

// open sza file list file
if((filelist_fd = fopen(sza_file_list, "r")) == NULL)
{
  fprintf(stderr, "could not open sza file list\n");
  exit(1);
}

// read the specified formant bank into
// a formant_bank_struct
if((fb = read_normalized_formant_bank(formant_bank_file)) == NULL)
{
  fprintf(stderr, "could not open formant bank file\n");
  exit(1);
}

// read the specified formant ranges file
// into a formant_ranges_struct
if((fr = read_formant_ranges_weights(formant_ranges_file)) == NULL)
{
  fprintf(stderr, "could not open formant ranges file\n");
  exit(1);
}

// read the specified configuration file
// into a config_struct
if((conf = read_config(config_file)) == NULL)
{
  fprintf(stderr, "could not open config file\n");
  exit(1);
}

// initializing the pointers to null, so
// cleanup wont crash the program
signal  = NULL;
f0      = NULL;
f0_norm = NULL;
score   = NULL;
fv      = NULL;

while(fgets(filename, MAXPATHLENGTH, filelist_fd) != NULL)  // for each file in the filelist
{
  cleanup(signal, f0, f0_norm, score, fv);
  transform_filename(tmp, sza_prefix, filename, NULL);
  if((signal = alaw2lin(tmp)) == NULL)
  {
    fprintf(stderr, "could not process %s\n", tmp);
    // continue and process next OR exit ???
  }
  transform_filename(tmp, f0_prefix, filename, conf->f0_ext);
  if((f0 = read_f0_vec(tmp)) == NULL)
  {
    fprintf(stderr, "could not process %s\n", tmp);
    // continue and process next OR exit ???
  }
  transform_filename(tmp, f0_norm_prefix, filename, conf->f0_norm_ext);
  if((f0_norm = read_f0_vec(tmp)) == NULL)
  {
    fprintf(stderr, "could not process %s\n", tmp);
    // continue and process next OR exit ???
  }
  if((score = extract_score_information(signal, f0, f0_norm, fb, conf)) == NULL)
  {
    fprintf(stderr, "computation failed for %s in extract_score_information()\n", tmp);
    // continue and process next OR exit ???

  }

      //   here one might use store_score_information()
      //
      //   transform_filename(tmp, target_prefix, filename, conf->score_ext);
      //   store_score_information(tmp, score);
      //
      //   ... continuing in another controller ...
      //   foreach(file)
      //   {
      //     load_score_information(file, score);
      //     generate_feature_vecs(...);
      //     store_feature_vecs(...);
      //   }

  if((fv = generate_feature_vecs(score, fb, fr, conf)) == NULL)
  {
    fprintf(stderr, "computation failed for %s in generate_feature_vecs()\n", tmp);
    // continue and process next OR exit ???
  }
  transform_filename(tmp, target_prefix, filename, conf->feature_ext);
```

```
    if(!store_feature_vecs(tmp, fv))
    {
      fprintf(stderr, "could not store feature_vecs file %s\n", tmp);
      // continue and process next OR exit ???
    }
  }

  // freeing all the datastructures
  cleanup(signal, f0, f0_norm, score, fv);
  dispose_cfg_struct(conf);
  dispose_formant_ranges_struct(fr);
  dispose_formant_bank_struct(fb);

  fclose(filelist_fd); // closing the filelist file
  exit(0);
}


void cleanup(struct double_vec *signal, struct double_vec *f0, struct double_vec *f0_norm,
  struct score_struct *score, struct feature_vecs_struct *fv)
{
  if(signal != NULL)
  {
    dispose_double_vec(signal);
  }
  if(f0 != NULL)
  {
    dispose_double_vec(f0);
  }
  if(f0_norm != NULL)
  {
    dispose_double_vec(f0_norm);
  }
  if(score != NULL)
  {
    dispose_score_struct(score);
  }
  if(fv != NULL)
  {
    dispose_feature_vecs_struct(fv);
  }
}


  ////////////////////////////////////////
 // EXTRACT_SCORE_INFORMATION_SKELETON.C //
////////////////////////////////////////

int extract_score_information(struct short_vec *signal, struct double_vec *f0,
  struct double_vec *f0_norm, struct formant_bank_struct *fb, struct cfg_struct *c, struct score_struct *s)
{
  // assumtions: all arguments are valid. check them, before
  // calling this function!
  int k = 0;                    // index for the f0 / f0_norm vectors
  int pos = 0;                  // start position of the current signal window
  double partial_distance;      // return value of get_partials()
  struct double_vec *partials; // return value of get_partials()
  double meanE;                 // mean energy of current spectrum

  while((k < f0->size) && (k < f0_norm->size) && (pos < signal->size))
  {
    // f0[k], f0_norm[k] and signal[pos] all exist
    if(!get_partials(signal, pos, f0->cont[k], c->fsG, &partial_distance, partials))
    {
      // get_partials() ran out of signal, while
      // calculating partial_distance and partials.
      // jumping out of while loop.
      break;
    }
    cepstral_dist_frmnt_bank();
    if(energy flag)
    {
      // do some energy related stuff
      // applied to return stuff of get_partials
      // I.E. append energy to return value of cepstral_dist_frmnt_bank
    }

    // add all the stuff to score, which is a
    // pointer to a double_vec array

    k++;
    pos += interval;
  }
  return(1);
}


  ////////////////////////////////////
 // GENERATE_FEATURE_VECS_SKELETON.C //
////////////////////////////////////

struct feature_vecs_struct *generate_feature_vecs(struct score_struct *s, struct formant_bank_struct *fb,
  struct formant_ranges_struct *fr, struct cfg_struct *c)
{
  int i,j,k;
  int N, M;
```

```
  int nof_ranges;
  int max_ind;                    // index of the maximal score in current formant range
  double max_val;                 // value of the maximal score in current formant range
  double max_energy;              // energy at the position with max score in current formant range
  struct feature_vecs_struct *fv;

  nof_ranges = fr->lower->size;  // number of formant ranges
  M = s->nof_frames;             // number of score vectors
  N = s->cont[0]->size;          // length of the score vectors
  if(c->e_included)
  {
    N--; // use length without mean energy for the loops
  }
  // create the feature vecs struct, however it looks
  for(i=0; i<M; i++)
  {
    // for each of the score vectors
    for(j=0; j<nof_ranges; j++)
    {
      // for each formant range
      max_ind = -1;
      max_val = -1.0; // assuming, score is never negative (???)
      for(k=0; k<N/2; k++)
      {
        // for each gridpoint
        if((fr->lower[j] <= fb->freq[k]) && (fb->freq[k] <= fr->higher[j]))
        {
          // the grid point lies in the formant range
          // check, if greater than current maximum
          if(/* score_vector[i]->component[k] > max_val */)
          {
            // set the newly found maximum value and index
            max_ind = k;
            max_val = /* score_vector[i]->component[k] */;
          }
        }
        if(fb->freq[k] > fr->higher[j])
        {
          // walked out of the formant range, continue
          // with the next
          break;
        }
      }
      if(max_ind == -1)
      {
        // if max_ind == -1, we have a problem (check formant ranges
        // and formant bank, they dont make much sense ...)
        return(NULL);
        // should we really abort, or just set max_val to zero (???)
      }
      max_energy = /* score_vector[i]->component[max_ind + N/2] */;
      // the maximum score can now be found at max_ind, its value
      // is max_val and the corresponding energy is max_energy
      if((max_val == 0) || (max_energy == 0))
      {
        // avoid computation of log(0)
        /* fv[i]->component[j] = EPS */
      }
      else
      {
        /* fv[i]->component[j] = log(max_val * max_energy) */;
      }
    }
  }
  // feature vectors for this signal computed so far,
  // decorrelate

  /* decorrelation goes here */
  return(fv);
}
```

B Test Environment Shell Scripts

Aufbau und Verwendung des Testenvironment

Im Verzeichnis `scripts` finden sich alle Skripts und Dateien, welche für die durchführung von Erkennungsexperimenten und die Konfiguration des Testenvironment benötigt werden. In diesem Anhang sind alle zur Verfügung stehenden Skripts, deren Verwendung und ihre Funktion beschrieben. Da das Testenvironment vollständig aus dem Verzeichnis `scripts` bedient werden kann, geben wir im Folgenden alle Pfade relativ zu `scripts` an.

Zuerst möchten wir jedoch zusammenfassen, wie die Installation der Scripts, das Erstellen eines Testenvironments und die Durchführung von Erkennungsexperimenten im Normalfall abläuft[17]:

Installation der Scripts

Die Umgebung kann, unter Beibehaltung der vorliegenden Verzeichnisstruktur, in einem beliebigen Verzeichnis installiert werden. Alle statischen Konfigurationsdaten können in der Datei `./config.txt` spezifiziert werden. Damit das Testenvironment funktioniert, muss die Datei `./config.txt` folgende Einträge enthalten:

- `LOCATION` spezifiziert das Directory, in welchem sich das Testenvironment befindet (Bsp. `/home/username/testenvironment`). Der Pfad darf *nicht* mit einem `/` enden.
- `SP_DAT_DIR` soll den Pfad zu den Daten der SpeechDat(II) Datenbank enthalten.
- `F0_DIR` muss auf das Verzeichnis gesetzt werden, welches die vorbereiteten Grundtondateien `*.f0` und `*.f0_base` enthält.
- `RESULT_DIR` ist das Basis-Verzeichnis für alle berechneten Resultate. Die mit `./SetDirectories` generierten Verzeichnisse werden alle in diesem Basis-Verzeichnis erstellt.
- `PASSWORD` spezifiziert das Passwort des Benutzers, welcher das Testenvironment verwendet. Diese Angabe wird von `./CreateFormantFeaturesParallel` für das Login auf die in `./machines.txt` angegebenen Server verwendet. Falls `./CreateFormantFeaturesParallel` nicht verwendet wird, kann für den Parameter `PASSWORD` der leere String angegeben werden.

Die Auswahl der Modelle, welche für Training und Tests verwendet werden sollen, muss durch ein- und auskommentieren der entsprechenden Zeilen in den Perlskripts per Hand durchgeführt werden.

Vorgehen bei Erstellung eines datenreduzierten Testenvironment

1. `./CreateLinks` (Erstellt das reduzierte Datenset)

[17] Details zu den jeweiligen Scripts werden in den folgenden Abschnitten beschrieben

2. `./CreateFileList` (Erstellt die Dateiliste für das Feature Extraktions Programm)

Vorgehen bei Durchführung von Erkennungsexperimenten

1. `./SetDirectories` (Erstellt neue Output-Directories)
2. `./CreateDirectoryStructure` (Generiert benötigte Verzeichnisstruktur)
3. `./CreateFormantFeaturesParallel` (startet die Parallele Feature Extraktion)
4. `./GetMissing` (Sucht nach fehlenden Feature Files)
5. `./CreateFormantFeatures` (Erstellt gegebenenfalls fehlende Features Files)
6. `./StartNoiseTrain` (Startet das Training des reduzierten Modellsets)
7. `./PerformAllTests` (Führt die Erkennungstests durch)
8. `./GetResults` (Wertet die Resultate aus)

CreateLinks

Verwendung

```
CreateLinks <nth>
```

Beschreibung

Nach Ausführung der Funktion `CreateLinks <nth>` wird für alle folgenden Erkennungsexperimente nur jede `<nth>`-te Datei aus der Speechdat(II) Datenbank verwendet. Diese Funktion kann also verwendet werden, um den Trade-off zwischen hoher Erkennungsrate und schneller Durchführung der Experimente einzustellen.

CreateFileList

Verwendung

```
./CreateFileList
```

Beschreibung

Nach der Anwendung der Funktion `CreateLinks` zur Erstellung einer neuen Datenauswahl für das Testenvironment, muss für das Formant Feature Extraktionsverfahren eine neue Dateiliste erstellt werden. Durch ausführen von `./CreateFileList` wird diese Dateiliste (`reduced_sp_dat_set.txt`) im dafür vorgesehenen Verzeichnis `formant_feature_extraction/params` erstellt. Dieses Script benötigt keine Argumente.

SetDirectories

Verwendung

```
SetDirectories <feature_dir_name> <model_dir_name> <work_dir_name>
```

Beschreibung

Diese Funktion wird verwendet, um die für das Testenvironment wesentlichen Verzeichnisse zwischen verschiedenen Erkennungsexperimenten umzustellen. Alle wichtigen Informationen zu einem Erkennungsexperiment werden in den drei Arbeitsverzeichnissen `feature_dir_name` (Features), `model_dir_name` (Trainierte Modelle) und `work_dir_name` (Arbeitsverzeichnis) gespeichert. Die drei Argumente sollten nicht als Pfade verstanden werden, sondern als Namen von Unterverzeichnissen im in `config.txt` spezifizierten Verzeichnis `base_dir`.

Besteht ein Verzeichnis mit dem angegebenen Namen bereits, so wird es verwendet. Wird ein Name spezifiziert, zu dem noch kein gleichnamiges Verzeichnis existiert, so wird ein Verzeichnis mit diesem Namen erstellt.

CreateDirectoryStructure

Verwendung

```
./CreateDirectoryStructure <target_dir>
```

Beschreibung

Wurde mit Hilfe von `SetDirectories` ein neues Feature Verzeichnis erstellt, so muss in diesem Verzeichnis vor Ausführen der Feature Extraktion zuerst die korrekte Verzeichnisstruktur erstellt werden. Dies wird im Normalfall durch die Ausführung des Befehls

```
./CreateDirectoryStructure ../features
```

erreicht.

CreateFormantFeatures

Verwendung

```
CreateFormantFeatures <frmt_bank> <frmt_ranges> <end_coef>
    <use_isophone> <use_f0_base> <file_list>
```

Beschreibung

Dieses Programm extrahiert die Features aus den in der Fileliste `<file_list>` enthaltenen .sza Dateien. Als Parameter werden die Formant Bank und die Formant Ranges, sowie andere für die Extraktion benötigte Parameter übergeben. Während der Ausführung werden keine Score Dateien erzeugt, d.h. im Verzeichnis `../features` befinden sich nur die Feature Files die ebenfalls Endung .sza haben. Die Flags `<use_isophone>` und `<use_f0_base>` dürfen nur die Werte 0 oder 1 annehmen. Wenn `<use_isophone>` auf 0 gesetzt wird, dann wird die linear shift Adaption verwendet. Wenn `<use_f0_base>` aktiviert ist, dann werden als die Basiswerte der Grundtöne für die Adaption verwendet, ansonsten die Momentanwerte [8].

CreateFormantFeaturesParallel

Verwendung

```
./CreateFormantFeaturesParallel <frmt_bank> <frmt_ranges>
   <end_coef> <use_isophone> <use_f0_base>
```

Beschreibung

Dieses Script verteilt die Arbeit der Formant Feature Extraktion der in `formant_feature_extraction/params/reduced_sp_dat_set.txt` spezifizierten Dateien auf ein vorher festgelegtes Set von Maschinen. Welche Maschinen verwendet werden sollen, kann in der Datei `scripts/ParallelExecution/machines.txt` spezifiziert werden. Die Datei `scripts/machines.txt` muss genau einen Eintrag pro Zeile enthalten und keine Maschine darf doppelt aufgeführt werden. Zur Identifikation der Maschinen können Name oder IP Adresse verwendet werden.

GetMissing

Verwendung

```
./GetMissing <directory> <missing_files_list>
```

Beschreibung

Falls während der Ausführung von `CreateFormantFeaturesParallel` ein Prozess auf einer der Maschinen vorzeitig terminiert hat, so kann es sein, dass nicht alle Feature Dateien die zum trainieren benötigt werden im Verzeichnis `<directory>` vorliegen. Da dieser Fall ab und zu eintrifft sollte man nach jedem Aufruf von `CreateFormantFeaturesParallel` dieses Skript ausführen. Es sucht in `<directory>` anhand der von `CreateFileList` erzeugten Dateiliste nach den fehlenden Dateien und generiert eine neue Liste `<missing_files_list>` von nicht vorhandenen Feature Dateien.

Diese Liste kann man danach dem Programm `CreateFormantFeatures` übergeben, welches die restlichen Feature Dateien erzeugt. Wenn `<missing_files_list>` leer ist, so sind alle Dateien erstellt worden und `StartNoiseTrain` kann ausgeführt werden.

StartNoiseTrain

Verwendung

```
./StartNoiseTrain
```

Beschreibung

Dieses Script startet nach abgeschlossener Extraktion der Features durch `CreateFormantFeatures` oder `CreateFormantFeaturesParallel` das Training des reduzierten Modellsets. Für die Ausführung müssen keine Parameter angegeben werden.

PerformAllTests

Verwendung

```
./PerformAllTests
```

Beschreibung

Nach dem abgeschlossenen Training der statistischen Modelle mit `./StartNoiseTrain` können mit diesem Programm alle zur Verfügung stehenden Erkennungsversuche durchgeführt werden. Anschliessend können die erzielten Resultate mit `GetResults` in Form einer Textdatei erstellt werden.

GetResults

Verwendung

```
GetResults <report_file_name>
```

Beschreibung

Nachdem mit dem Befehl `PerformAllTests` ein Erkennungstest durchgeführt wurde, können mit dem Script `GetResults <report_file_name>` alle Resultate zusammengefasst in ein Textfile namens `<report_file_name>` geschrieben werden. Zu beachten ist, dass `<report_file_name>` ein relativer Pfad sein muss. Das skript funktioniert *nicht* bei Angabe eines absoluten Pfades.

C Aufgabenstellung

Die folgenden Seiten enthalten die ursprüngliche Aufgabenstellung zu unserer Semsterarbeit.

Figurenverzeichnis

Tabellenverzeichnis