
Spracherkennung einzelner Befehlswoorte zur Steuerung eines Roboters

Projektarbeit

Sven Schwarz

13. November 1998

Inhaltsverzeichnis

1	Motivation	1
2	Einführung	2
2.1	Allgemeines zur Sprache	3
3	Realisierung	7
3.1	Die Realisierung in Kürze	7
3.2	Bestimmen der Wortgrenzen	10
3.3	Aufteilen der Sprache in mehrere Frames	11
3.4	Der Merkmalsvektor	13
3.4.1	Die Filterbänke	13
3.5	Der Prototyp	14
3.5.1	Dynamische Zeitverzerrung	15
3.6	Akzeptanzkriterien	16
4	Die Hear-Schnittstelle	19
4.1	Der Hear-Server	19
4.2	Allgemeines zur Schnittstelle	20
4.3	Die Konfiguration des Servers	20
4.4	Die Funktionsbibliothek	22
5	Trainieren der Befehle	28
5.1	Die Trainingsapplikation	28
6	Erkennen der Befehle	31
6.1	Einfaches Beispielprogramm zur Erkennung	31
6.2	Einfaches Beispielprogramm zur parallelen Erkennung	32
7	Fazit	35
	Literaturverzeichnis	36

Kapitel 1

Motivation

In Forschungsbereichen, wie zum Beispiel der Robotik-AG an der Universität Kaiserslautern, wird der Roboter, neben den laufenden Programmen, nur mittels Tastatur, Joystick oder anderen, auf dem Roboter befindlichen Tasten bedient. Der „Not-Aus“-Schalter sitzt sogar auf dem Roboter selbst. Wenn sich also die Tastatur nicht in Reichweite befindet, muß man diesen Schalter früh genug drücken, bevor sich der Roboter verselbstständigt.

Diese Projektarbeit soll dem Abhilfe schaffen, und es ermöglichen, den Roboter mit simplen Ein-Wort-Befehlen zu steuern.

Spracherkennung ist schon lange kein neues Thema mehr, und so kann man diese Arbeit keinesfalls als Forschungsdokument betrachten. Es gibt zwar noch genug in dieser Domäne zu forschen und zu entwickeln, aber dabei geht es um innovativere Mechanismen. Dennoch kann mit Recht behauptet werden, daß es sich zeitweilig zur reinen Forschung entwickelt hat, da es anfangs recht schwierig war, gute Literatur über Implementierungsdetails bezüglich Spracherkennung zu finden.

Kapitel 2

Einführung

Das Stichwort „Spracherkennung“ ist bereits im Kapitel Motivation gefallen. Spracherkennung wird aber für eine Vielzahl von Anwendungen benutzt. Daher gibt es auch verschiedene Anforderungen, die man an die Spracherkennung stellt. Bei einem Diktiermodul, zum Beispiel, muß fließende Sprache erkannt werden. Dies bedeutet einen wesentlich höheren Aufwand als wenn einzelne, getrennt voneinander gesprochene Worte erkannt werden sollen. Hinzu kommt noch eine eventuelle Notwendigkeit für Lexika oder gar Grammatikprüfungen.

Hier soll Spracherkennung benutzt werden, um einem Roboter Ein-Wort-Befehle zu erteilen. Es soll dabei nicht als Joystick-Ersatz dienen. Vielmehr sollen die Befehle höhere Ziele symbolisieren. Zum Beispiel sagt man dem Roboter STOP, ZURÜCK oder KÜCHE. Es sollen also Ein-Wort-Befehle sein, die ausreichen, um ein zuvor damit verknüpftes (höheres) Ziel zu verfolgen. Der Befehl KÜCHE könnte demnach ein Kommando sein, das den Roboter dazu veranlaßt, die Küche zu suchen und dorthin zu fahren. Es geht hier also nicht um ein korrektes Diktat komplexer Sätze, sondern um ein Dutzend Befehls Worte. Fassen wir die daraus resultierenden Fakten noch einmal zusammen:

Positive Fakten:

- einzelne Worte
- keine zusammenhängenden Worte
- vor und nach jedem Wort herrscht (mehr oder weniger) Stille
- geringe Anzahl der Worte (typisch: zwischen 10 und 30)
- Worte werden so gewählt, daß sie sich gut unterscheiden (kooperativ)
- sprecherabhängiges Verstehen kann toleriert werden, da die Anzahl der Wörter sehr klein ist – gegebenenfalls muß also jeder Sprecher seine eigene Aussprache der Befehls Worte trainieren

Guten Morgen, Herr Hauptkommissar Thanner. Gibt es irgendetwas Neues im Fall "Verbmobil"?	der Text in "Schönschrift"
Morgen, Thanner. Irgendwas Neues im Fall "Verbmobil"?	spontan gesprochene Sprache
morgen thanner irgendwas neues im fall verbmobil	Großschreibung? Satzzeichen?
morgenthannerirgendwasneuesimfallverbmobil	kontinuierliche Sprache
moangtannaingwasneuesimfalwerpmobiel	Aussprachevarianten
<i>moangtannaingwasneuesimfalwerpmobiel</i>	artikulatorische Verschleifung
<i>moangtannaingwasneuesimfalwerpmobiel</i>	Störungen & Verzerrungen des akustischen Kanals
<i>moangtannaingwasneuesimfalwerpmobiel</i>	Überlagerung d. Fremdstimmen „Cocktailparty-Effekt“

Abbildung 2.1: Warum ist Spracherkennung schwierig?

Negative Fakten:

- es gibt Umgebungsgeräusche (z.B. andere leisere Stimmen)
- es steht evtl. nicht immer ein „Head-Phone“ zur Verfügung, d.h. es sollte auch mit einem normalen (relativ schlechten) Mikrofon funktionieren
- Zurufen auf größere Entfernung wird im Allgemeinen nicht funktionieren

2.1 Allgemeines zur Sprache

Dazu möchte ich einen Auszug aus dem Buch [ST95] zitieren:

Wir erleben das Verstehen gesprochener Mitteilungen als eine alltägliche und selbstverständliche menschliche Fertigkeit, die selbst von kleinen Kindern, Berufspolitikern und Tennisprofessionals mit überraschender Leichtigkeit vollzogen wird. Um so befremdlicher muß uns daher das Faktum erscheinen, daß für die maschinelle Realisierung dieser Wahrnehmungsleistung auch nach Jahrzehnten intensiver Forschungstätigkeit bis zum heutigen Tage nur rudimentäre Teillösungen angeboten werden konnten.

Unserem subjektiven Eindruck nach besteht eine Sprachäußerung aus einer Folge von Wörtern, diese wiederum aus einer Folge von Lauten; insgesamt ergibt sich ein zeitlich gerichteter Strom diskreter Einheiten, aneinandergereiht wie die Perlen einer Kette, und jedes Lautsegment erscheint ausgestattet mit invarianten, vom jeweiligen Kontext völlig unabhängigen Klangeigenschaften. Dieser Eindruck täuscht jedoch; er resultiert lediglich aus einer hochkomplexen neurophysiologischen und kognitiven internen Verarbeitung des Sprachschalls. Dieser Sprachschall stellt eine hochreduzante Kodierung der zu übermittelnden Nachricht dar. Die hervorstechendsten Phänomene, die bei der artikulatorischen Verschlüsselung des „Klartextes“ einer Mitteilung auftreten, sind in der Abbildung 2.1 wiedergegeben. Zur Veranschaulichung wurde dabei das Medium gesprochener Sprache durch das geschriebener Sprache ersetzt.

Die Schwierigkeiten, welche zum Zwecke maschineller Spracherkennung zu bewältigen sind, lassen sich grob in vier Gruppen gliedern: diese betreffen die Kontinuität, die Variabilität, die Komplexität und die Ambiguität gesprochener Sprache.

Kontinuität: Entgegen dem oben beschriebenen Eindruck existieren im Sprachschall selbst im allgemeinen *keine* sichtbaren Diskontinuitäten, welche die Grenzen zwischen Wörtern, Silben oder Lauten eindeutig markieren. Selbst eine sichere Detektion von Wortgrenzen ist nur in Sonderfällen gewährleistet, wenn nämlich hervorzuhebende Phrasengrenzen intonationsbedingt mittels kurzer Stilleintervalle realisiert wurden.

Variabilität: Ein und dieselbe Spracheinheit kann akustisch auf vielfältigste Weise realisiert sein. Es ist daher ungemein schwierig, für Wörter oder Laute geeignete akustische *Prototypen* zu finden, welche die Referenzeinheit über stark variierende Begleitumstände hinweg korrekt repräsentieren. Ein solcher Prototyp muß einerseits hinreichend generell gewählt werden, um alle Realisierungen zu umfassen, er muß die Bezugseinheit aber andererseits auch scharf gegen konkurrierende Einheiten abgrenzen. Ursachen für diese Variabilität sind in spezifischen Eigenschaften des *Aufnahmekanals* (Typ und Position des Mikrofons, räumliche Reflektionseigenschaften, Diskretisierungsrauschen), oder *akustische Störquellen* (Stimmen weiterer Personen, Verkehrsgeräusch, Büro- oder Fertigungsumgebung) zu suchen. Ferner stellen sich Variationen ein aufgrund der *Sprechweise* (Tempo, Artikulationsdruck, Anspannung, Emotionen, Kooperativität), *individueller* Sprechermerkmale (Alter, Geschlecht, Vokaltraktanatomie, Gesundheitszustand) und *habituellem* Sprechermerkmalen (Dialekt, Soziolekt). Der vielleicht massivste Einfluß ist in der *kontextuellen Aussprachevariation* zu sehen. Abhängig von ihrer unmittelbaren lautlichen Umgebung und der Satzprosodie erhalten wir akustische Ausprägungen mit unterschiedlichsten Klang-, Intensitäts- und Rhythmusseigenschaften.

Komplexität: Die automatische Spracherkennung erfordert hohe Rechenleistung und Speicherkapazität. Gründe dafür sind die Datenrate der Spracheingabe (typisch sind 8000 bis 20000 Signalabtastwerte je Sekunde), die umfangreichen Inventare von Erkennungseinheiten und die ausufernde Kombinatorik bei der Satzbildung

(aus K verschiedenen Wörtern lassen sich bekanntlich K^L unterschiedliche Wortfolgen der Länge L bilden). Die Erkennungseinheiten schlagen gleich zweimal zu Buche: ihre Prototypen oder Modelle tragen zum Speicherbedarf bei, und sie sind überdies für die Anzahl der erforderlichen Mustervergleiche während der Analysephase verantwortlich.

Ambiguität: Zwischen Spracheinheiten und ihren Realisierungen besteht im allgemeinen keine eindeutige Zuordnung; insbesondere können unterschiedliche Wörter oder Sätze unter Umständen gleich oder sehr ähnlich artikuliert werden. Diese Mehrdeutigkeiten treten auf allen sprachlichen Ebenen auf: als *Homophonien* ('Rad' und 'Rat'), an Wortgrenzen ('Stau-becken' und 'Staub-ecken'), im syntaktischen ('... das Tonband, das Nixon vernichtete ... ') und semantischen ('Bienenhonig' und 'Imkerhonig') Bereich.

Zur Dekodierung einer gesprochenen Nachricht aus ihrem akustischen Korrelat nutzen wir Menschen Restriktionen aus einer Fülle unterschiedlicher Wissensquellen, etwa der Artikulatorik, Phonetik, Morphologie, Syntax und Semantik, um die durch Verschlüsselungseffekte verschüttete Originalinformation zu rekonstruieren. Die hervorragende Performanz des Menschen bei der Erkennung von Wörtern und Sätzen ist also undenkbar ohne unsere Fähigkeit zur Unterscheidung wohlgeformter Satzbauten von Fehlkonstruktionen, ohne die Interaktion von Allgemeinwissen, Situationseinschätzung und Inferenzmechanismen zur Plausibilitätsbewertung potentieller Bedeutungen. Diese Tatsache war bereits im ausgehenden 19. Jahrhundert wohlbekannt:

„When we listen to a person speaking or read a page of print much of what we think we see or hear is supplied from our memory.”

und läßt sich unschwer am Beispiel fremdsprachlicher Kommunikation belegen, die empfindlich unter Störungen des Übertragungskanal wie Straßenlärm oder schlechter Telefonverbindung leidet. In letzter Konsequenz impliziert diese (nicht ganz unumstrittene) These für die maschinelle Sprachverarbeitung, daß die Realisierung eines Diktierautomaten nur auf der Basis eines Computers denkbar wäre, der über eine derjenigen des Menschen nahekommende Intelligenz verfügt.

Was dem Hörer recht ist, ist dem Sprecher billig. Im festen Vertrauen auf die Rezeptionsleistung unseres Zuhörers suchen wir unser Kommunikationsziel (nämlich verstanden zu werden) mit dem Minimum des dafür unbedingt erforderlichen Aufwandes zu erreichen. Dabei steht die Redundanz der Mitteilung in umgekehrt proportionalem Verhältnis zur Qualität ihrer Darbietung. Die Dosierung unserer „Nachlässigkeit“ bei der Realisierung eines kompliziert strukturierten Eigennamens am Telefon wird erheblich vorsichtiger ausfallen als beim alltäglich wiederkehrenden Morgengruß im Büro. Besonders präzise formulieren und deutlich artikulieren werden wir nur dann, wenn die Verständigungsbedingungen schlecht sind, unser Anliegen wesentliche Neuinformation beinhaltet, oder wenn wir bei unserem Gegenüber eine mangelnde Vertrautheit

mit der aktuell verwendeten Sprache vermuten. Aus diesem adaptiven Sprecherverhalten ergibt sich eine unmittelbare Folgerung für die Auslegung einer Mensch-Maschine-Schnittstelle: nur das Bewußtsein, mit einer *Maschine* zu kommunizieren, spornt den Benutzer an, eine Spracheingabe hoher Qualität zu produzieren. Der gutgemeinte Versuch hingegen, einen menschlichen oder mit nahezu menschlichen Fähigkeiten ausgestatteten Gesprächspartner vorzutäuschen, führte wohl zwangsläufig zu einer anfänglichen Überschätzung der Systemleistung, dadurch bedingter Fehladaptation, Einbruch der Erkennungsrate und schließlich Frustration beim Benutzer.

Kapitel 3

Realisierung

3.1 Die Realisierung in Kürze

Grundsätzlich geschieht die Worterkennung bei mir in folgenden Schritten (hervorgehobene Begriffe sind einstellbare Parameter der Spracherkennung):

1. Warten bis Lautstärkepegel *hoch genug* ist
2. Aufnahme der Sprache bis Lautstärkepegel *für längere Zeit* nicht mehr *hoch genug*
3. Zerschneiden der aufgenommenen Sprache in *gleich lange Stücke* („Frames“)
4. Für jeden solchen Frame:
 - (a) Herausfiltern von Frequenzinformationen
 - (b) Berechnen der Outputs für eine *Anzahl* von *Frequenzbändern* und quantisieren der Pegel auf ganzzahlige Skalare innerhalb eines *kleinen Intervalls*

Nach Schritt 4 erhält man für ein gesprochenes Wort also eine Liste von Vektoren mit ganzzahligen Skalaren. Diese Vektoren mit ganzzahligen Skalaren werden im folgenden „Merkmalsvektoren“ genannt. Eine Liste dieser Merkmalsvektoren wird „Prototyp“ genannt. Nach diesem Schema verfährt man mit jedem aufgenommenen Wort und legt eine Fallbasis aller dieser Prototypen mit dazugehörigen Zeichenketten an.

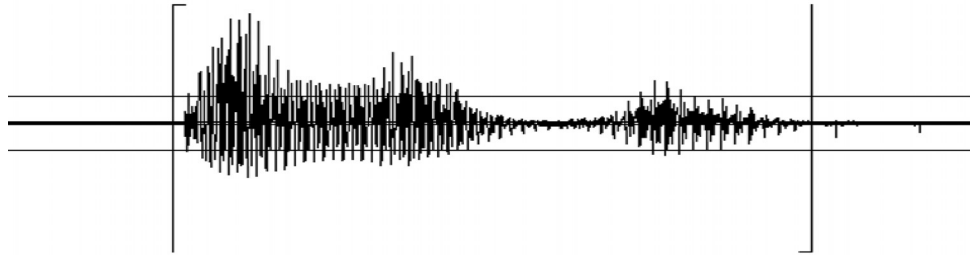
Wird nun eines der trainierten Worte noch einmal ins Mikrofon gesprochen und daraus ein Prototyp generiert (Durchlaufen der Schritte 1 bis 4), so kann die Fallbasis dazu benutzt werden, die dazugehörige Zeichenkette zu liefern. Dies geschieht, indem man denjenigen Prototyp aus der Fallbasis ermittelt, der die geringste Abweichung zum Prototypen des gerade gesprochenen Wortes hat. Man sucht also den „nächsten Nachbarn“. Das Wort (die Zeichenkette), welches mit dem gefundenen Prototypen gespeichert wurde, ist nun das von der Spracherkennung *erkannte* Wort. Wurde das Wort nicht korrekt erkannt (die ermittelte Zeichenkette entsprach dem gesprochenen Wort *nicht*), fügt man

den neuen Prototyp mit dazugehöriger Zeichenkette ebenfalls zur Fallbasis dazu. Das bedeutet, daß sich auch mehrere Prototypen ein und desselben Wortes in der Fallbasis befinden können.

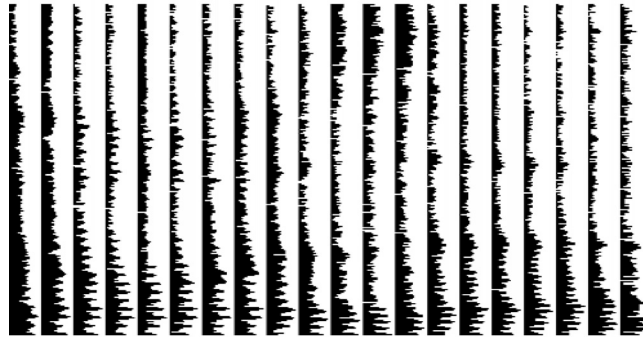
Kurz zur Suche des „nächsten Nachbarn“: Da die gesprochenen Worte unterschiedlich lang sein können, sind auch die daraus resultierenden Prototypen unterschiedlich lang. Normen, wie zum Beispiel die euklidische Norm, kann man daher nicht zur Ermittlung des Abstands zweier Prototypen zu Rate ziehen. Stattdessen wird ein Algorithmus namens Dynamische Zeitverzerrung (Dynamic Timewarping) benutzt. Wie dieser Algorithmus funktioniert wird später erläutert.

Auf der nächsten Seite wird das Erstellen eines Prototypen aufgrund einer Sprachaufnahme noch einmal grafisch erläutert.

Schritt 1 - 3: Bestimmung Anfang, Ende der Sprache (*hier: das Wort „Dose“*) und in gleich lange Frames zerlegen (*hier: Frames zu je 40ms*)



Schritt 4 (a): FFT über die Samples jedes Frames (*hier: 256 reellwertige Koeffizienten*)



Schritt 4 (b): Quantisieren der FFT-Koeffizienten auf eine kleine Anzahl von Koeffizienten (*hier: 16*), sowie Quantisieren der Koeffizienten auf ganzzahlige Werte innerhalb eines kleinen Intervalles (*hier: [0;3]*)



Hier noch ein Beispiel für eine Liste von Prototypen mit dazugehörigen Zeichenketten:

	dose
	halt
	licht
	los
	stop
	tür

3.2 Bestimmen der Wortgrenzen

Bei der Einzelworterkennung gibt es zunächst zwei grundlegende Probleme:

1. Wie erkennt man die Wortgrenzen (Anfang, Ende) eines einzelnen gesprochenen Wortes?
2. Woher weiß man, daß dieses gesprochene, einzelne Wort auch wirklich als Befehl an den Computer gedacht war und nicht bloß ein Teil der Unterhaltung zweier Personen?

Der zweite Punkt ist so gut wie gar nicht zu lösen, es sei denn, man zwingt den Benutzer, vor oder während der Spracheingabe stets eine Tastenkombination zu drücken. Da wir aber nur sehr wenige Befehle erkennen wollen, können wir diejenigen „erkannten“ Worte zurückweisen, die keinem vorhandenen Prototypen ähnlich genug sind. Diese Akzeptanzkriterien sind in Abschnitt 3.6 erläutert.

Doch nun zum eigentlichen Problem dieses Abschnitts: Die Erkennung der Wortgrenzen. Auch hier gibt es die einfache Möglichkeit, den Anfang und das Ende des zu erkennenden Wortes vom Benutzer zu markieren. Dies kann zum Beispiel durch Drücken einer Taste, mehrerer Tasten oder Buttons geschehen.

Wir wollen hier jedoch *keine* Hilfe vom Benutzer erwarten! Daher sind wir gezwungen, ständig am Mikrofon zu lauschen und den Anfang und das Ende eines gesprochenen Wortes ausschließlich anhand des Lautstärkepegels zu ermitteln. Genauer gesagt wird ständig ein Block von Samplewerten vom Mikrofon gelesen und der durchschnittliche Lautstärkepegel für diesen Block ermittelt. Die Größe dieses Blocks ist ein Parameter und kann eingestellt werden. Ist der Lautstärkepegel eines aufgenommenen Blocks größer als ein bestimmter Wert L_{Stille} , so beginnt ab diesem Block wahrscheinlich ein gesprochenes Wort. Warum hier „wahrscheinlich“ geschrieben wurde, wird gleich klar.

Zunächst zur Berechnung eines „Lautstärkepegels“. Er errechnet sich aus dem Durchschnitt der ersten Ableitung bezüglich der Samplewerte. Wenn also n Samplewerte eingelesen wurden und die $\{x_i\}$ die Samplewerte sind, dann errechnet sich der Lautstärkepegel L aus:

$$L = \frac{1}{n} \sum_{i=1}^{n-1} |x_{i+1} - x_i| \quad (3.1)$$

Bevor mit dem Training bzw. der Erkennung der Befehls Worte begonnen wird, muß zumindest einmal der Lautstärkepegel gemessen werden, wenn mehr oder weniger Stille herrscht. Dieser Lautstärkepegel wird leicht angehoben (mit einem Faktor wie 1.5 multipliziert) und als L_{Stille} gespeichert. Das leichte Anheben des Lautstärkepegels ist notwendig, da sonst die Erkennung des Wortbeginns zu empfindlich ist.

Wurde nun der Beginn eines gesprochenen Wortes festgestellt, werden weiter Sampleblöcke eingelesen und gespeichert. Wie wird nun das Ende des Wortes erkannt?

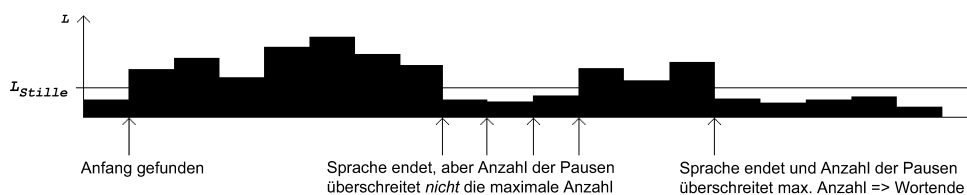


Abbildung 3.1: Bestimmung der Wortgrenzen

Sehr einfach wäre es, den Abfall des Lautstärkepegels unterhalb von L_{Stille} als Maß für das Wortende zu nehmen. Dies würde aber nicht funktionieren, da sich in vielen Worten kurze „Innerwortpausen“ befinden. Diese Innerwortpausen gilt es also zu überlesen. Daher wurde ein Parameter eingeführt, der die Anzahl der maximal zu überlesenden Pausenblöcke (Sampleblöcke mit Lautstärkepegel unterhalb von L_{Stille}) angibt. Werden mehr Pausen erkannt, so wird dies als Ende des gesprochenen Wortes interpretiert. Die letzten Pausenblöcke werden nicht beachtet, da es sich ja nicht um Innerwortpausen, sondern um Pausen am Ende eines Wortes zu handeln scheint. Zur Verdeutlichung soll Abbildung 3.1 dienen.

3.3 Aufteilen der Sprache in mehrere Frames

Es gibt zunächst zwei verschiedene Möglichkeiten, das ganze gesprochene Wort in Frames zu zerteilen:

1. Die Anzahl der Frames ist fix. Damit richtet sich die *Länge* der einzelnen Frames nach der gesamten Länge des gesprochenen Wortes.
2. Die Länge der Frames ist fix. Das bedeutet, daß die *Anzahl* der Frames variabel ist.

Die Vor- und Nachteile beider Alternativen liegen klar auf der Hand: Die erste Alternative ist, abgesehen von der FFT, leicht zu implementieren. Die meisten FFT-Algorithmen arbeiten allerdings nur korrekt, wenn die Anzahl der Stützstellen eine Zweierpotenz ist. Man müßte also nach speziellen, optimierten Algorithmen suchen, die auch mit beliebigen Anzahlen von Stützstellen arbeiten. Dafür ist aber der Vergleich zweier Prototypen sehr einfach, da zwei Prototypen immer die gleiche Anzahl an Merkmalsvektoren besitzen. Man kann leicht beide Prototypen mittels einer beliebigen Metrik vergleichen. Wenn viele der zu trainierenden Befehls Worte unterschiedlich lang sind, muß man bei dieser Alternative jedoch starke Erkennungsschwächen in Kauf nehmen.

Die zweite Alternative ist natürlicher, da sie eher dem entspricht, was das menschliche Ohr leistet. Innerhalb eines kleinen Zeitintervalles (ca. 10ms) kann das menschliche Ohr keine Veränderungen wahrnehmen. Das bedeutet, man kann Geräusche innerhalb

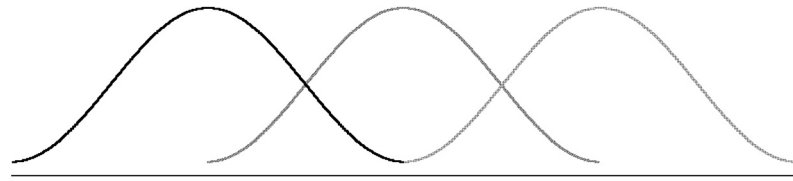


Abbildung 3.2: Drei überlappende Frames mit Hammingfenster.

dieses Intervalles als *einen* einzigen Laut auffassen. Dies wird nun genutzt, in dem man das gesamte gesprochene Wort in gleichlange Frames (zu je 10ms) zerteilt. Für jeden Frame wird ein Merkmalsvektor berechnet (siehe folgenden Abschnitt). Es resultiert ein Prototyp – eine Liste dieser Merkmalsvektoren, deren Länge von Wort zu Wort variieren kann. Ein Vergleich von Prototypen dieser Art ist natürlich erheblich schwieriger und aufwendiger zu realisieren. Darauf wird jedoch in Abschnitt 3.5.1 eingegangen.

Obwohl die Geräusche innerhalb eines kleinen Zeitintervalles als einzelne Laute aufgefaßt werden können, wäre es eine zu voreilige Methode, die gesamte Äußerung *gleichmäßig* in Stücke (Frames) zu je 10ms aufzuteilen. Das Problem liegt an den Grenzen der Frames! Manche Äußerungen werden vielleicht zwischen dem Ende eines Frames und dem Anfang seines Nachfolgers verschluckt oder werden zumindest verzerrt gespeichert. Daher werden zwei Maßnahmen ergriffen:

- Über jeden Frame wird ein sogenanntes „Hammingfenster“ gelegt. Das heißt, jeder Samplewert innerhalb eines Frames wird nun – bevor er weiter verarbeitet wird – zunächst mit dem Funktionswert des Hammingfensters an der entsprechenden Stelle multipliziert. Mit dieser Vorgehensweise wird die Unempfindlichkeit am Anfang und am Ende der Frames eliminiert.
- Die Frames überlappen sich. Dadurch kommt es nicht so leicht zu dem Effekt, daß Laute verschluckt oder verzerrt werden. Es bietet sogar die Möglichkeit, die Größe der Frames höher zu setzen (auf 20ms oder gar 40ms) und damit die Geschwindigkeit der Erkennung zu steigern.

In Abbildung 3.2 ist ein Beispiel für die Nutzung der Hammingfenster und der Überlappung von drei Frames gezeigt. Als Hammingfenster wurde die Gaußglocke (siehe Abbildung 3.3) verwandt. Als Versatz der Überlappung zweier Frames wurde die Hälfte der Framegröße gewählt.

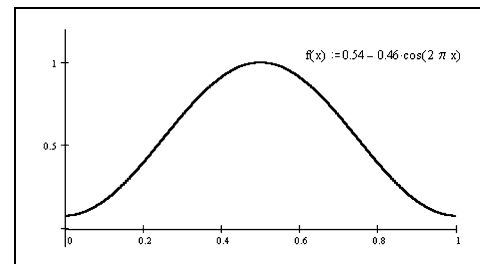


Abbildung 3.3: Gaußglockenkurve

3.4 Der Merkmalsvektor

Hat man ein Frame eines gesprochenen Wortes extrahiert, soll nun daraus ein Merkmalsvektor gebildet werden. Wie der Name schon sagt, soll ein Merkmalsvektor *Merkmale* speichern. Merkmale verstehen sich als Ausprägungen eines Objektes, mit deren Hilfe dieses eine Objekt oder ähnliche Objekte wiedererkannt werden können. Das bedeutet, die Merkmale sollen typisch (für ein Objekt) sein. Typische Merkmale zur Erkennung eines Apfels sind beispielsweise die Farbe, die Form und eventuell die Größe, aber keinesfalls die Anzahl der Blätter am Zweig oder gar die Anzahl der Wurmlöcher.

Im vorigen Abschnitt wurde davon gesprochen, daß ein Frame eines gesprochenen Wortes einem einzigen Laut entspricht. Nun gilt es, Merkmale zu finden, die es ermöglichen, bestimmte Laute zu identifizieren bzw. unterscheiden zu können. Die Natur hilft bei der Suche nach diesen Merkmalen: Die menschlichen Laute bestehen, vereinfacht ausgedrückt, aus einer Überlagerung mehrerer Frequenzen zwischen 100Hz und 4kHz. Eine genauere Analyse der menschlichen Lauterzeugung kann man in [Wit82] nachlesen. In diesem Buch werden die Merkmalsvektoren durch drei „Formanten“ erzeugt. Ein Formant ist die stärkste Frequenz innerhalb eines definierten Frequenzintervalles. Da die Suche nach den Formanten nicht ganz trivial ist, wurde hier ein alternativer, ebenfalls effektiver Weg beschritten. Anstatt drei bestimmte Frequenzen zu *suchen*, berechne ich den Output von *fest vorgegebenen* Filterbänken. Diese Outputs werden skalar quantisiert und bilden den Merkmalsvektor. Die skalare Quantisierung wird vollzogen, um die reellwertigen Outputs der Filterbänke in ganzzahlige Werte umzuwandeln, die unabhängig von der Lautstärke der Äußerung sind. Es findet also eine Normierung bezüglich der Lautstärke statt.

Ein *sehr* wichtiger Faktor für die Erkennungsrate ist die Wahl der Filterbänke. Dies betrifft die Frequenz und die Breite jeder Filterbank. Da sich menschliche Laute im tiefen Frequenzspektrum wesentlich mehr unterscheiden als im hohen Bereich, müssen die Filterbänke für die tiefen Frequenzen schmal und die Filterbänke für die hohen Frequenzen breit sein. Die daraus resultierende, auditorisch begründete Verteilung der Filterbänke wird auch oft als *melscale*-Verteilung beschrieben.

3.4.1 Die Filterbänke

Aus auditiven Gesichtspunkten heraus sollten die Filterbänke in einer *melscale*-Verteilung angeordnet werden: Also viele schmale Filterbänke im tiefen Frequenzbereich und wenige breite Filterbänke im hohen Frequenzbereich.

Es gibt mehrere Arten, Filterbänke zu implementieren. In diesem System wurde folgender Weg eingeschlagen: Zunächst wird eine Fast-Fourier-Transformation (FFT) über die einzelnen Frames berechnet und daraus reellwertige, diskrete Fourierkoeffizienten ermittelt. Aus diesen relativ vielen Koeffizienten wird dann für jede Filterbank ein „Output“ berechnet.

Jede Filterbank hat drei Parameter, die ihren Frequenzbereich genau beschreibt: eine Mittelfrequenz, eine linke Grenze und eine rechte Grenze. Mit diesen Parametern wird eine Dreiecksfunktion gebildet, wie sie in Abbildung 3.4.1 zu sehen ist. Sie hat also ihr Maximum an der Mittelfrequenz, fällt von dort aus zur linken sowie zur rechten Grenze linear auf Null ab und bleibt jenseits der linken bzw. rechten Grenze gleich Null.

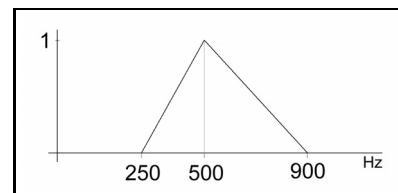


Abbildung 3.4.1

Die Koeffizienten, welche die FFT geliefert hat, werden nun für jede Filterbank mit der zugehörigen Dreiecksfunktion multipliziert und diese Faktoren addiert. Es ergibt sich also ein einziger Wert – ein sogenannter Output. Diese Output-Werte werden in einem Vektor gespeichert und bilden nun schließlich den Merkmalsvektor für den Frame.

Anmerkung: Dieses System arbeitet mit unnormierten Merkmalsvektoren. Wollte man normierte Merkmalsvektoren verwenden, läge es nahe, darauf zu achten, daß die Flächeninhalte der Dreiecksfunktionen normiert sind. Man müßte in diesem Falle die Funktionswerte durch das Integral der Dreiecksfunktion teilen. Die Dreiecksfunktionen würden in diesem Falle mit zunehmender Frequenz nicht nur breiter, sondern auch flacher werden.

3.5 Der Prototyp

Ein Prototyp eines gesprochenen Wortes besteht aus einer endlichen Liste von Merkmalsvektoren. Die Anzahl der Merkmalsvektoren ist im allgemeinen von Prototyp zu Prototyp verschieden, da auch die Aussprache von verschiedenen Wörtern im allgemeinen verschieden lang ist.

Die Berechnung und Zusammensetzung der Merkmalsvektoren wurde im vorangegangenen Abschnitt 3.4 beschrieben. Daher soll hier nicht erneut auf Details der Merkmalsvektoren eingegangen werden. Es genügt für diesen Abschnitt, zu wissen, daß alle Merkmalsvektoren aus einer fest vorgegebenen, gleichen Anzahl von Werten bestehen. Demnach können je zwei Merkmalsvektoren mittels des euklidischen Maßes verglichen werden.

Da die Prototypen zweier Worte im allgemeinen *nicht* gleich lang sind – also die Anzahl der Merkmalsvektoren nicht identisch ist – ist ein Vergleich von zwei Prototypen kein triviales Problem. Dieser Vergleich ist aber von grundlegender Wichtigkeit, da er uns sowohl ermöglicht, ein gesprochenes Wort zu klassifizieren, als auch eine Abschätzung über die Güte der Klassifikation zu treffen. Letzteres ist wichtig für das Abweisen von Wörtern, die nicht trainiert wurden und wahrscheinlich lediglich Gesprächsteile einer Unterhaltung zweier Personen sind, die sich vor dem Mikrophon befinden. Die Kriterien für eine solche Abweisung werden im Abschnitt 3.6 behandelt.

Es gibt zwei Lösungsansätze, mit deren Hilfe man zwei verschieden lange Prototypen vergleichen kann:

1. Dynamische Zeitverzerrung – Dynamic Timewarping
2. Markov-Modelle – (Hidden) Markov Models

In diesem System wird die Dynamische Zeitverzerrung benutzt. Dieses Verfahren wird im nächsten Abschnitt erklärt.

Die Markov-Modelle basieren auf (versteckten) Zustandgraphen. Versteckt deshalb, weil der Zustandsgraph nicht direkt modelliert wird, sondern sich mit der Zeit langsam an das zu Lernende anpaßt. Es gibt auch keine *festen* Übergänge von einem Zustand in *einen* anderen, sondern *Wahrscheinlichkeiten* für Übergänge von einem Zustand in *mehrere* andere Zustände. Es ist also ein statistisches Verfahren. Mehr Informationen zu der Benutzung von Markov-Modellen bei der Spracherkennung kann man in [ST95], [Nud] oder [Hil97] nachlesen.

3.5.1 Dynamische Zeitverzerrung

Die Dynamische Zeitverzerrung wird benutzt, um den Abstand zweier Listen unterschiedlicher Länge zu ermitteln. Dabei ist Voraussetzung, daß für die Elemente der Listen eine Metrik existiert. In unserem Fall sind die Elemente Merkmalsvektoren, für welche die euklidische Metrik benutzt wird. Damit ist die Voraussetzung erfüllt.

Das Verfahren wurde leicht modifiziert, um die Effizienz zu steigern. Genauer gesagt lag der Algorithmus in rekursiver Form vor (z. B. in [Hil97] oder [Teb95]), während für dieses System eine iterative Version implementiert wurde. Der Algorithmus ist wie folgt:

Auf einem zweidimensionalen Feld wird die eine Liste auf der unteren Achse und die andere Liste auf der linken Achse aufgetragen (dies natürlich nur gedanklich). Nun wird jedes Feld mit den Abständen der Merkmalsvektoren unten bzw. links gefüllt. Dieses Feld wird im folgenden „Abstandsfeld“ genannt. Anschließend wird derjenige Pfad durch das Feld gesucht, der unten links startet, oben rechts endet und in der Summe möglichst wenige Abstände in Kauf nehmen muß.

Die Folge der Zellen wird weiter eingeschränkt durch die Bedingung, daß von einer Zelle aus immer nur eine Zelle nach rechts, nach oben oder diagonal nach rechts oben gesprungen werden darf.

Aus dieser Bedingung ergibt sich direkt als Gültigkeitsbereich für die möglichen Punkte des Pfades ein nach rechts geneigtes Parallelogramm.

Um weiter Effizienz zu gewinnen, kann man das Parallelogramm noch beliebig schmälern. Dabei sollte man sich aber stets bewußt sein, daß man die Möglichkeiten des Pfades eventuell so weit einengt, daß die optimalen Pfade nicht mehr erreicht werden können, und das bedeutet, daß ähnliche Worte vielleicht einen großen Abstand zueinander haben werden. Es wird hier ein leicht geschmälertes Parallelogramm mit Steigungen von $\frac{1}{2}$ bzw. 2 verwendet. Die Abbildung 3.3 verdeutlicht den Sachverhalt grafisch.

Anstatt nun oben rechts anzufangen und sich rekursiv nach unten links zu hangeln, wird *spaltenweise* von links nach rechts gegangen. Dazu wird ein weiteres zweidimensionales

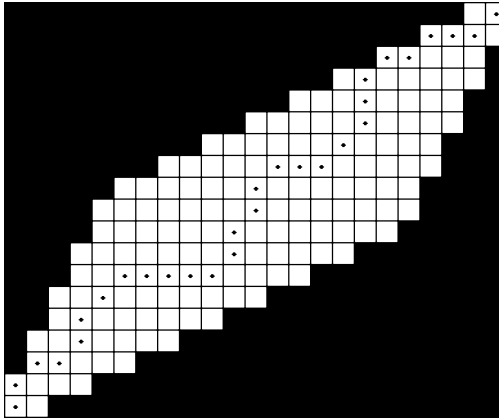


Abbildung 3.3: Dynamische Zeitverzerrung mit geschmälertem Parallelogramm (Steigung 2 bzw. 0.5). Die Punkte geben einen möglichen Pfad durch das zweidimensionale Feld an. Schwarze Felder sind nicht zugänglich.

Feld benötigt, in dem für jede Zelle die minimale Summe der Abstände gespeichert wird, die man in Kauf nehmen muß, wenn der Pfad durch *diese* Zelle gehen soll. Dieses zweidimensionale Feld wird „Summenfeld“ genannt. Das Summenfeld wird spaltenweise von links nach rechts (und in der Spalte von oben nach unten) gefüllt. Für jede Zelle in der aktuellen Spalte werden die folgenden drei Zelleneinträge verglichen:

- der Eintrag der Zelle, die sich direkt unter der aktuellen Zelle befindet,
- der Eintrag der Zelle, die direkt links von der aktuellen Zelle liegt und
- der Eintrag der Zelle, die sich direkt diagonal (links unten) befindet

Das Minimum dieser drei Werte wird ermittelt und zum Abstand, der unter der Zelle im Abstandsfeld gespeichert wurde, addiert. Dieser Wert wird nun in der aktuellen Zelle im Summenfeld gespeichert. Er beinhaltet die Summe der Abstände für den besten Pfad von der untersten linken Ecke bis zu dieser Zelle.

Ist das Summenfeld vollständig gefüllt, enthält die oberste rechte Zelle die Summe aller Abstände für den bestmöglichen Pfad. Dieser Wert wird als „Abstand“ der beiden Prototypen interpretiert.

3.6 Akzeptanzkriterien

Die Spracherkennung soll als einfache Sprachsteuerung eingesetzt werden, um einen Roboter mittels einfacher Befehls Worte zu steuern. Nun wird sich der Roboter nicht etwa in einem leeren Gebäude bewegen, in dem totale Stille herrscht. Vielmehr werden überall mehr oder weniger leise Gespräche zwischen Menschen zu hören sein. Würde die Spracherkennung einfach immer nur dasjenige Wort ermitteln, das einem „gehört“ Wort am ähnlichsten ist, würde der Roboter vielleicht Teile der Unterhaltung zweier Menschen als Befehle auffassen. Um diesem Problem zu begegnen, wurden Akzeptanzkriterien

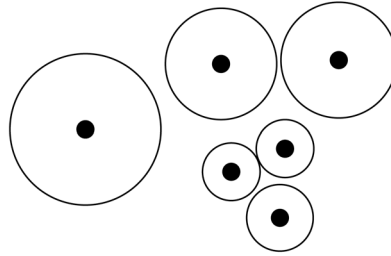


Abbildung 3.4: Sphären um Prototypen aus der Datenbank. Die Radien der Sphären sind hier jeweils halb so groß wie der Abstand zum nächsten Prototypen

eingeführt, die es ermöglichen, Worte *abzuweisen*, die keinem der Prototypen ähnlich genug sind.

Achtung! Hiermit kann natürlich nicht das Problem behoben werden, das entsteht, wenn zwei Personen sich unterhalten und dabei ein Wort fällt, welches auch tatsächlich trainiert wurde. Hierzu ein Beispiel: Angenommen es wurde, unter anderem, das Befehlswort LINKS trainiert und es käme zu folgender Unterhaltung zweier Personen in Reichweite des Mikrofons:

Person 1: Wo sind denn die Installationsdisketten?

Person 2: Im Schrank.

Person 1: Hier?

Person 2: Nein! LINKS.

Dieses Problem ist nur durch temporäres Ausschalten der Spracherkennung oder durch totale Lautverfälschung bei der Unterhaltung (z. B. durch Führen der Unterhaltung auf Pfälzisch) zu lösen.

Eines der Kriterien ergibt sich schon aus der Benutzung des Parallelogramms im Algorithmus für die Dynamische Zeitverzerrung. Dadurch scheidet ein Vergleich von zwei Prototypen bereits aus, sobald ein Prototyp länger als doppelt so lang als der andere ist. Das Kriterium greift zum Beispiel bei den beiden Wörtern ACHTUNG und LOS, d. h. der Abstand zwischen diesen beiden Wörtern wird unendlich groß sein.

Um aufgenommene Wörter abzuwehren, für die kein Prototyp trainiert wurde, reicht dieses Kriterium noch nicht aus, da es sich nur um ein zeitliches Kriterium handelt, aber nicht um ein qualitatives. Das Problem wurde wie folgt gelöst:

Nach jedem Hinzufügen eines Prototyps zur Datenbank wird eine Sphäre um jeden Prototyp in der Datenbank berechnet. Der Radius jeder Sphäre wird so gewählt, daß sie nur den eigenen Prototyp enthält. Abbildung 3.4 verdeutlicht den Sachverhalt für den zwei-dimensionalen Fall (hier sind die Merkmalsvektoren allerdings 16-dimensional). Soll nun ein Prototyp eines gehörten Wortes mit einem Prototyp aus der Datenbank verglichen werden, wird der Abstand beider Prototypen mittels Dynamischer Zeitverzerrung, wie

in Abschnitt 3.5.1 beschrieben, ermittelt. Bevor nun aber der Abstand zurückgeliefert wird, testet die Funktion, ob der Prototyp des gehörten Wortes innerhalb der Sphäre des Prototypen aus der Datenbank ist. Nur wenn der Abstand beider Prototypen kleiner als der Radius der Sphäre ist, wird der Abstand als Ergebnis zurückgeliefert. Andernfalls wird „Unendlich“ zurückgeliefert.

Hier wird der Radius der Sphäre eines Prototyps sogar nur auf die *Hälfte* des Abstands zum nächstgelegenen Prototypen gesetzt. Der sogenannte Parameter „Radiusfaktor“ ist also mit 0.5 belegt. Damit wird eine Fehlerkennung von nicht trainierten Worten noch weiter eingeschränkt. Allerdings wird dabei auch riskiert, daß eventuell sehr undeutlich gesprochene Worte nicht erkannt werden, obwohl sie trainiert wurden. Bei normalen Sprechweisen gibt es aber keinerlei Probleme. Es soll an dieser Stelle auch daran erinnert werden, daß es ja durchaus vorgesehen ist, pro trainiertes Wort mehrere Prototypen in der Datenbank zu speichern.

Falls trotzdem solche Probleme auftreten sollten, können sie aber bereits im Trainingsstadium erkannt werden und gegebenenfalls durch leichtes Tunen der Parameter eliminiert werden. Alle für die Spracherkennung wichtige Parameter können innerhalb der Trainingsapplikation eingestellt werden. Dies gilt natürlich insbesondere für den „Radiusfaktor“. Details über das Tunen der Parameter innerhalb der Trainingsapplikation sind dem Kapitel 5 zu entnehmen.

Kapitel 4

Die Hear-Schnittstelle

Die vorherigen Kapitel beschäftigten sich mit der Theorie, die der Grundlagen der Spracherkennung und die der Implementation. Ab diesem Kapitel geht es nun um die Praxis. Außerdem ist der Hear-Server ohne Portierungsaufwand nur auf **qnx**-Systemen lauffähig, da er **qnx**-spezifische Systemaufrufe benutzt. Die Trainingssuite wurde ebenfalls für die **qnx**-Plattform entwickelt. Ausschließlich der Kern der Spracherkennung läuft auch unter Unix und Linux.

Alle Applikationen, die etwas mit der Spracherkennung zu tun haben – sei es, daß sie Befehlswoorte trainieren, oder daß sie Worte erkennen – greifen auf den „Hear-Server“ zu.

4.1 Der Hear-Server

Der Hear-Server stellt alle Funktionen bereit, die nötig sind, um Befehlswoorte zu trainieren und wiederzuerkennen. Darüber hinaus bietet er auch Funktionen zur Feineinstellung der Spracherkennungsparameter. Diese Funktionen werden als „Remote Procedure Calls“ (RPS's) zur Verfügung gestellt. Dies bedeutet in Kürze folgendes: Nach dem Start des Hear-Servers registriert sich dieser beim System als Server. Möchte nun eine Applikation die Funktionen des Hear-Servers nutzen, muß sie sich zunächst mit dem Hear-Server verbinden.

Die genaue Funktionsweise des Verbindungsaufbaus oder der RPC's ist allerdings nicht von Belang, da eine Schnittstelle angeboten wird, welche die Funktionalität des Servers als „normale“ Funktionen bereitstellt und die dafür notwendigen Interaktionen mit dem Hear-Server intern regelt. Ein Client muß lediglich die Datei `hearclient.h` inkludieren und vor dem ersten Aufruf einer Hear-Funktion die Verbindung zum Hear-Server sicherstellen. Letzteres geschieht einfach durch Aufruf von `hear_init()`.

Da dadurch die Benutzung der RPC's komplett verdeckt wird, wird im folgenden nicht mehr auf Details bezüglich dieser RPC's eingegangen, sondern nur noch auf die Schnittstelle, die von `hearclient.h` bereitgestellt wird.

4.2 Allgemeines zur Schnittstelle

Allen Funktionen der Hear-Schnittstelle ist folgendes gemein:

- Die Funktionsnamen beginnen jeweils mit dem Präfix `“hear_”`.
- Alle Funktionen geben einen Pseudo-Boolean (`int`) zurück, der Aufschluß darüber gibt, ob die Abarbeitung der Funktion erfolgreich war.

Achtung! Es ist unbedingt zu beachten, daß es sich bei der Hear-Schnittstelle um eine C-Schnittstelle handelt. Das bedeutet, daß C++-Programme die Schnittstelle immer folgendermaßen inkludieren müssen:

```
extern "C" {
    #include "hearclient.h"
}
```

Es darf nicht vergessen werden, die Verbindung zum Server mittels `hear_init()` herzustellen. Die Funktionalität steht andernfalls nicht zur Verfügung!

4.3 Die Konfiguration des Servers

Die Konfiguration des Servers beinhaltet alle Parametereinstellungen der Spracherkennung. Das Einstellen dieser Parameter ist nicht unbedingt notwendig, da von mir schon Voreinstellungen getroffen wurden. Dennoch ist es sinnvoll, die Parameter an eine bestimmte Situation anzupassen – zum Beispiel an eine niedrigere Samplefrequenz, um die Effizienz noch zu steigern.

Das Einstellen eines Parameters geschieht mittels `hear_changeConfig()`, während mit `hear_askConfig()` die aktuelle Einstellung eines Parameters hinterfragt werden kann.

Die einzelnen Parameter werden im folgenden beschrieben. Dabei steht der Name des Parameters links in der umrandeten Box und der Defaultwert rechts. Zusätzlich zum Namen des Parameters wird sein Typ in C-Schreibweise mit angegeben.

<code>int sampleSpeed</code>	<code>= 11025</code>
------------------------------	----------------------

Die Samplerate mit der die gesprochene Sprache aufgenommen wird. Die Einheit des Parameters ist Hertz.

<code>int frameSize</code>	<code>= 256</code>
----------------------------	--------------------

Die Sprache wird in „Frames“ (Sprachstücke) aufgeteilt (siehe Abschnitt 3.3). Dieser Parameter enthält die Anzahl der Samplewerte, die benutzt werden, um einen solchen Frame zu bilden.

<code>int frameAdvance</code>	<code>= 128</code>
-------------------------------	--------------------

Die Anzahl der Samplewerte, die zwischen dem Beginn eines Frames und dem Beginn seines folgenden Frames liegen. Bei sich überlappenden Frames gibt dieser Wert den Versatz von zwei benachbarten Frames an. Standardmäßig ist er halb so groß wie `frameSize`, was bedeutet, das sich zwei benachbarte Frames genau zur Hälfte überlappen. Sollen sich die Frames nicht überlappen, sollten `frameAdvance` und `frameSize` den gleichen Wert beinhalten.

<code>int frameDimension</code>	<code>= 16</code>
---------------------------------	-------------------

Die Anzahl der Elemente im Merkmalsvektor. Dies ist gleichzeitig auch die Anzahl der Filterbänke. Das bedeutet, daß dieser Parameter immer zusammen mit dem Parameter `middleFreqs` geändert werden muß!

<code>int scalarMaxValue</code>	<code>= 3</code>
---------------------------------	------------------

Dieser Parameter steuert die Quantisierung der Skalare während der Berechnung eines Merkmalsvektors. Er gibt den höchsten ganzzahligen Wert an, der nach der Quantisierung erreicht werden kann. Ist der Wert beispielsweise auf 3 eingestellt, bedeutet dies, daß Merkmalsvektoren später ganzzahlige Werte zwischen 0 und 3 als Elemente enthalten.

<code>int[] middleFreqs</code>	<code>= 150 250 350 465 595 740 885 1060 1250 1500 1750 2025 2400 2850 3300 3950 4750 6250</code>
--------------------------------	---

Der Inhalt dieses Feldes bestimmt die Lage und Bandbreite der Filterbänke. Details zu den Filterbänken können dem Abschnitt 3.4.1 entnommen werden.

Das Feld enthält die Mittelfrequenzen für die einzelnen Filterbänke. Dabei dient die Mittelfrequenz des Vorgängers als linke Grenze und die Mittelfrequenz des Nachfolgers als rechte Grenze. Daher enthält das Feld zwei Werte mehr als es Filterbänke gibt, denn der erste Filter benötigt ja auch eine linke Grenze und der letzte eine rechte Grenze. Somit beginnen die eigentlichen Mittelfrequenzen erst an zweiter Stelle (Index 1) im Feld.

Es ist stets darauf zu achten, daß `frameDimension` passend gesetzt ist, also einen Wert enthält, der um zwei kleiner ist, als die Anzahl der Elemente in diesem Feld.

<code>float thresholdSilence</code>	<code>= 25.0</code>
-------------------------------------	---------------------

Ein geräteabhängiger Wert, der den Geräuschpegel angibt. Er sollte eigentlich nicht von Hand gesetzt werden. Vielmehr wird er mit dem Befehl `hear_measureSilence()` gemessen und gesetzt.

```
int silenceFrames = 20
```

Die Anzahl der Frames, die *in Serie* unterhalb des Geräuschpegels sein müssen, um das Wortende zu identifizieren. Taucht innerhalb einer Serie solcher Frames auch nur ein einziger Frame auf, dessen Lautstärkepegel oberhalb des Geräuschpegels liegt, ist die Serie unterbrochen und muß erneut gebildet werden. Die Anzahl der „leisen“ Frames wird also nicht auf der *gesamten* Sprache aufsummiert, sondern nur als Serie.

Ein zu kleiner Wert für `silenceFrames` kann dazu führen, daß ein Wortende erkannt wurde, obwohl es sich bloß um eine Innerwortpause handelte. Ein zu großer Wert hat keinen sehr negativen Einfluß auf die Erkennung, und verschlechtert die Effizienz auch nur leicht. Daher sollte der Wert im Zweifelsfall eher zu groß als zu klein ausfallen.

```
int minFrames = 10
```

Dieser Parameter bestimmt die minimale Anzahl an Frames, die ein gesprochenes Wort haben muß, damit es bearbeitet wird. Hat ein gesprochenes Wort weniger Frames, wird es als undefinierbares Geräusch interpretiert und gar nicht erst bearbeitet, geschweige denn erkannt.

```
int maxFrames = 60
```

Dieser Parameter bestimmt die maximale Anzahl an Frames, die ein gesprochenes Wort höchstens haben darf, damit es bearbeitet wird. Hat ein gesprochenes Wort mehr Frames, wird es als Aneinanderreihung *mehrerer* Wörter interpretiert und gar nicht erst bearbeitet, geschweige denn erkannt.

4.4 Die Funktionsbibliothek

```
int hear_init ()
```

Die Initialisierungsfunktion. Sie initialisiert die Schnittstelle und stellt die Verbindung zum Hear-Server her. Diese Funktion muß aufgerufen werden, bevor andere Funktionen aus dieser Bibliothek aufgerufen werden. Die Funktion gibt 1 zurück, falls die Initialisierung erfolgreich war, sonst 0.

```
int hear_changeConfig (const char* name, const char* value)
```

Mittels dieser Funktion kann der Wert eines Parameters geändert werden. Dabei wird stets sowohl der Name des Parameters (siehe vorherigen Abschnitt) als auch der neue Wert des Parameters als C-String übergeben. Der Rückgabewert gibt Aufschluß darüber, ob das Ändern des Parameters erfolgreich war oder nicht.

Beispiel: `hear_changeConfig("sampleSpeed", "22050");`

```
int hear_askConfig (const char* name, char* value)
```

Mittels dieser Funktion kann der Wert eines Parameters ausgelesen werden. Dabei wird der Name des Parameters als C-String übergeben. Außerdem wird ein C-String übergeben, der groß genug sein sollte, um den Wert des Parameters als String zu enthalten. Der Rückgabewert gibt Aufschluß darüber, ob das Ändern des Parameters erfolgreich war oder nicht.

Beispiel:

```
char valueString[10];
int value;
hear_askConfig("sampleSpeed", valueString);
value = atoi(valueString);
```

```
int hear_saveConfig (const char* filename)
```

Diese Funktion speichert alle Parametereinstellungen in einer Datei, deren Namen man der Funktion übergibt. Diese Datei ist im Textformat gespeichert und kann, falls gewollt, editiert werden. Zum Format der Textdatei siehe Funktion `hear_loadConfig()`.

```
int hear_loadConfig (const char* filename)
```

Diese Funktion lädt Parametereinstellungen aus einer Textdatei. Die Textdatei ist im Normalfall eine Datei, die mittels `hear_saveConfig()` erstellt wurde. Sie kann aber auch von Hand erstellt worden sein.

Das Format der Datei verdeutlicht die folgende Beispieldatei:

```
#####
#                                     #
#   config-file for HearServer and Hear-clients   #
#                                     #
#####

sampleSpeed      8000   # the sampling rate

frameSize        200   # nr. of samples used to construct one frame
frameAdvance     100   # nr. of samples to advance for next frame

frameDimension   14    # nr. of elements in feature vector for one frame
scalarMaxValue   3     # scalars in feature vector will be int's from 0 to scalarMaxValue

# the next one specifies the middle frequencies
# the number of elements must be:  1 + frameDimension + 1
middleFreqs      150 250 350 465 595 740 885 1060 1250 1500 1750 2025 2400 2850 3300 3950

silenceFrames    20    # nr. of frames that must be silence before speech ends
minFrames        20    # min. nr. of frames to accept sample as speech
maxFrames        50    # max. nr. of frames to accept sample as speech
```

Zeichketten, die sich (innerhalb einer Zeile) hinter einem '#' befinden, werden als Kommentare gewertet und schlichtweg ignoriert.

```
int hear_getConfigFilename (char* filename)
```

Hiermit kann die zuletzt geladene Konfigurationsdatei ermittelt werden. Falls noch keine Konfigurationsdatei geladen wurde, enthält `filename` einen leeren C-String.

```
int hear_measureSilence (double* thresholdSilence)
```

Hiermit wird der Geräuschpegel gemessen. Während der Messung, also während der Ausführung dieser Funktion, sollte der Sprecher vor dem Mikrofon nicht sprechen, da andernfalls ein zu hoher Geräuschpegel gemessen wird und damit keine Worte mehr erkannt werden können.

Diese Funktion sollte immer dann aufgerufen werden, wenn sich die örtlichen Gegebenheiten oder das Mikrofon ändern, zumindest aber ein Mal vor der ersten Benutzung dieses Systems. Der Wert dieses Parameters sollte immer vor dem Trainieren oder Erkennen von Befehlsworten gesetzt werden. Dies kann entweder durch direktes Setzen mittels `hear_changeConfig()` oder durch Laden einer Konfigurationsdatei mittels `hear_loadConfig()` geschehen.

```
int hear_getSpeech ()
```

Der Aufruf dieser Funktion blockiert den aktuellen Prozeß so lange, bis ein Wort gesprochen wurde. Das heißt, es wird auf den Anfang und auf das Ende eines Wortes gewartet. Das Wort wird noch nicht erkannt! Es wird lediglich der Prototyp des gesprochenen Wortes gebildet.

Die Funktion gibt 0 zurück, falls kein Prototyp gebildet werden konnte. Das gesprochene Wort war dann entweder zu kurz (\rightarrow `minFrames`) oder zu lang (\rightarrow `maxFrames`). Konnte der Prototyp gebildet werden wird 1 zurückgegeben.

Nach der erfolgreichen Ausführung von `hear_getSpeech()` wird man im allgemeinen versuchen, das Wort mittels `hear_recognize()` zu erkennen und/oder den Prototypen mittels `hear_store()` zur Datenbank hinzuzufügen.

Beispiel:

```
if (hear_getSpeech()) {
    // Sprache war vorhanden und ist nun Prototyp gebildet
    if (!hear_recognize(...)) {
        // Wort nicht erkannt => User nach dem Wort fragen
        . . .
        hear_store(<WORT ALS C-STRING>);
    }
}
```

Anmerkungen: Natürlich sollte immer der Rückgabewert beachtet werden (also auch bei `hear_store()`). Der Parameter von `hear_recognize` wird weiter unten behandelt.

```
int hear_store (const char* word)
```

Falls bereits ein Prototyp mit `hear_getSpeech()` gebildet wurde, wird dieser hiermit zur Datenbank hinzugefügt.

Das übergebene Wort wird dabei dem Prototypen zugeordnet. Genau dieses Wort wird beim Erkennen eines gesprochenen Wortes zurückgegeben, falls dieser Prototyp derjenige ist, der dem Prototyp des gesprochenen am nächsten ist. Mehr Details dazu bietet Abschnitt 3.5.

Der Rückgabewert gibt Aufschluß darüber, ob die Funktion erfolgreich war oder nicht.

```
int hear_recognize (RecognizeInfo* info)
```

Falls bereits ein Prototyp mit `hear_getSpeech()` gebildet wurde, kann mit dieser Funktion versucht werden, diesen Prototypen zu erkennen. Details zur Erkennung können dem Abschnitt 3.5 entnommen werden.

`info` ist ein Zeiger auf eine `RecognizeInfo`-Struktur. Diese ist in `hearclient.h` folgendermaßen deklariert.

```
typedef struct
{
    int accepted;
    char word[80];
    double distance;
    double distance2;
} RecognizeInfo;
```

`accepted` enthält einen Pseudo-Bool'schen Wert, der angibt, ob das Wort erkannt werden konnte. Dieser Wert ist unbedingt zu überprüfen! `word` ist ein C-String, der das erkannte Wort enthält. `distance` enthält den Abstand des Prototypen zum nächsten Prototypen aus der Datenbank. Der Abstand wird mit Dynamischer Zeitverzerrung (3.5.1) ermittelt. `distance2` enthält den Abstand zum zweitnächsten Prototypen. Die Differenz $|distance - distance2|$ oder der Quotient $\frac{distance2}{distance}$ sind beide ein gutes Vertrauensmaß für die Erkennung. Ist der Wert sehr klein (bei 0 bzw. bei 1), handelt es sich wahrscheinlich eher um einen Glückstreffer, als um eine korrekte Erkennung des gesprochenen Wortes.

```
int hear_saveProtos (const char* filename)
```

Diese Funktion speichert alle Prototypen, die sich zur Zeit in der Datenbank befinden, in einer Datei. Der Dateiname der Datei wird als C-String übergeben.

```
int hear_loadProtos (const char* filename, int append)
```

Diese Funktion lädt Prototypen aus einer Datei. Entweder werden diese Prototypen dann zur Datenbank hinzugefügt, oder die vorhandenen Prototypen in der Datenbank werden durch die neuen aus der Datei ersetzt. Wird der Parameter `append` auf 1 gesetzt, werden die Prototypen aus der Datei der Datenbank hinzugefügt; Wird er auf 0 gesetzt, werden zuvor alle Einträge der Datenbank gelöscht und anschließend die Prototypen aus der Datei hinzugefügt. Der Dateiname der Datei wird als C-String übergeben.

```
int hear_getProtosFilename (char* filename)
```

Hiermit kann die zuletzt geladene Prototypen-Datei ermittelt werden. Falls noch keine Prototypen-Datei geladen wurde, enthält `filename` einen leeren C-String.

```
int hear_getKnownWords (int* number, char*** words)
```

Mittels dieser Funktion kann man alle Befehlswoorte erfragen, die das System zur Zeit kennt.

`words` ist ein Zeiger auf ein Feld mit C-Strings. Dieses Feld wird von der Funktion erzeugt und gefüllt. Die Anzahl der Elemente wird in `*number` übergeben. Das Feld sollte nach Gebrauch der Informationen wieder vom Benutzer gelöscht werden. Dies wird *nicht* von der Schnittstelle erledigt!

Beispiel:

```
int number, i;
char** words;
if (hear_getKnownWords(&number, &words)) {
    . . . // Nutzen der Information in <words>
    for (i = 0; i < number; i++)
        delete [] words[i];
    delete [] words;
}
```

```
int hear_deleteWord (const char* word)
```

Diese Funktion löscht alle Prototypen aus der Datenbank, die dem übergebenen Wort zugeordnet sind.

```
int hear_deleteAllProtos ()
```

Diese Funktion löscht *alle* Prototypen aus der Datenbank.

```
int hear_saveSample (const char* filename)
```

Diese Funktion speichert die Sampledaten des gesprochenen Wortes. Dafür muß `hear_getSpeech()` vor dieser Funktion einmal aufgerufen worden sein.

Der hauptsächliche Nutzen dieser Funktion ist das Archivieren und Wiederverwenden von einmal gesprochenen Befehlsworten zum Trainieren des Systems.

```
int hear_loadSample (const char* filename)
```

Diese Funktion lädt Sampledaten des gesprochenen Wortes. Sie kann als Alternative zu `hear_getSpeech()` benutzt werden.

Der hauptsächliche Nutzen dieser Funktion ist das Archivieren und Wiederverwenden von einmal gesprochenen Befehlsworten zum Trainieren des Systems.

Kapitel 5

Trainieren der Befehle

Dem Trainieren der Befehlswoorte wurde eine eigene Applikation gewidmet. Mit dieser Applikation hat man zugleich noch die totale Kontrolle über alle Parameter der Spracherkennung.

Während des Trainings der Wörter kann man bereits feststellen, ob die Spracherkennung später funktionieren wird. Beispielsweise stellt sich heraus, ob sich die Befehlswoorte genügend unterscheiden, um erkannt zu werden oder ob die Spracherkennung sie ständig verwechselt. Manche Woorte wird das System wahrscheinlich gar nicht erkennen, weil sie viel zu kurz und viel zu undeutlich sind. Ein Beispiel dafür ist das Wort STOP. Es ist sehr kurz und enthält, abgesehen vom dem 'o', nur Zisch- und Knacklaute, sowie Innerwoortpausen. Und alles das innerhalb weniger Frames.

Stellt man fest, daß die aktuelle Konfiguration ungeeignet für das Zielvorhaben zu sein scheint, kann man mit geringem Aufwand die Parameter ändern und das Training erneut beginnen.

5.1 Die Trainingsapplikation

Vor dem Start der Trainingsapplikation muß sichergestellt sein, daß der Hear-Server bereits läuft. Dies kann beispielsweise geschehen durch `"sin | grep hearserver"` innerhalb einer Shell.

Falls der Hear-Server wider erwarten noch nicht läuft, ist er auf jeden Fall vor der Trainingsapplikation zu starten. Dies geschieht durch den einfachen Aufruf von `"hearserver"`.

Bevor zu der Erklärung der Applikation übergegangen wird, noch ein kurzer Hinweis: Auf Details der Spracherkennung wird hier nicht mehr eingegangen. Diese wurden bereits in den vorherigen beiden Kapiteln behandelt.

Nach Starten der Trainingsapplikation sollte sie sich ähnlich wie in Abbildung 5.1 zeigen.

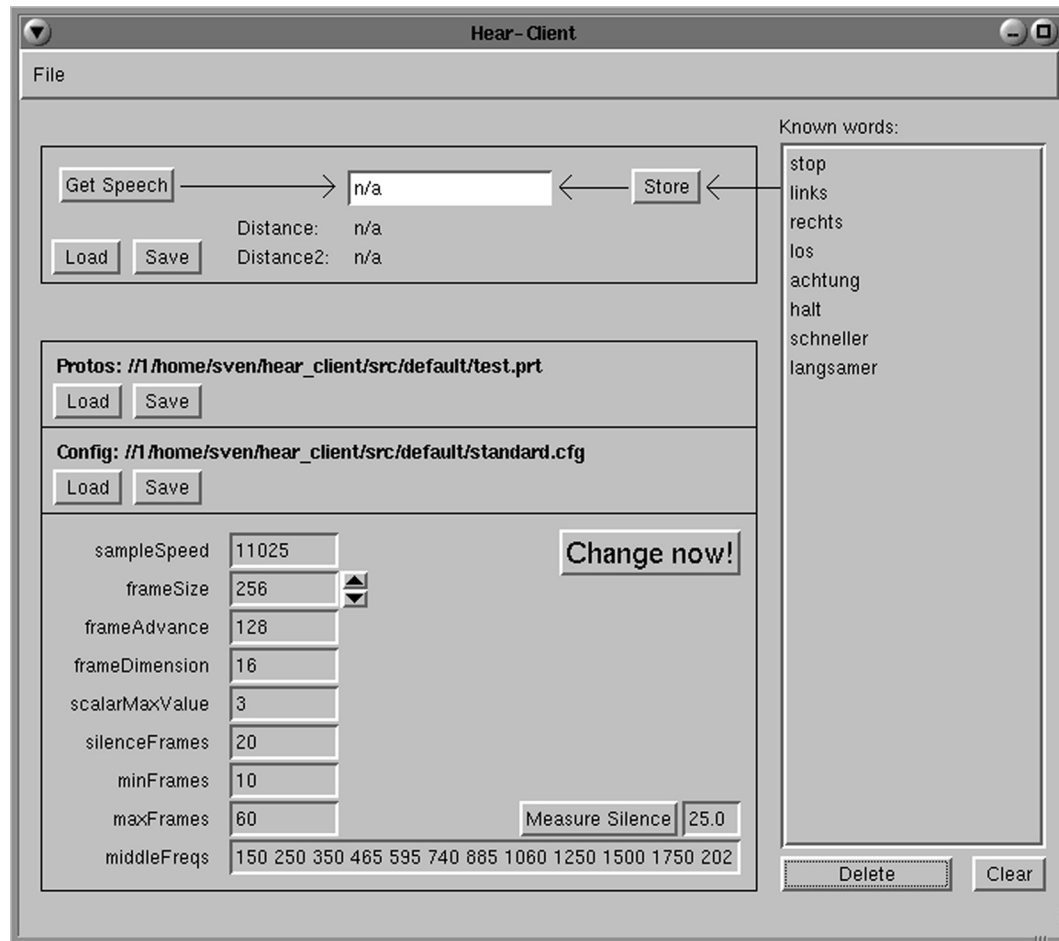


Abbildung 5.1: Die Trainingsapplikation

In der linken oberen Ecke befinden sich die Steuerelemente zur Spracheingabe und Erkennung. Nach Drücken des Buttons `Get Speech` wartet die Applikation auf die Spracheingabe eines Wortes und versucht direkt danach, dieses zu erkennen. Das heißt, intern werden nacheinander die Funktionen `hear_getSpeech()` und `hear_recognize()` aufgerufen. Während die Applikation auf die Sprache wartet, bleibt der Button gedrückt. Ist die Erkennung erfolgreich, erscheint das erkannte Wort in dem Eingabefeld rechts von dem Button. Ansonsten erscheint dort "n/a" (not available).

Wurde ein Wort erkannt, ist aber die Zuordnung falsch, so kann der Text im Eingabefeld editiert werden. Ein abschließendes `RETURN` oder ein Druck auf `Store` fügt dann den neu gewonnenen Prototypen mit der neuen Zuordnung (Eingabefeld) zur Datenbank hinzu.

Alternativ zum Editieren der Eingabezeile kann man rechts in der Liste "Known words" ein Wort auswählen. Durch Druck auf `Store` kann der neu gewonnene Prototyp mit der Zuordnung zum gewählten Wort zur Datenbank hinzugefügt werden. Beide alternativen

Verfahren rufen intern die Funktion `hear_store()` auf.

Außer dem erkannten Wort werden auch die Informationen `distance` und `distance2` über die Güte der Erkennung dieses Wortes angezeigt.

Die Buttons `Load` und `Save` beziehen sich auf das Laden und Speichern der Sampledaten.

Die Liste "Known words" enthält stets alle zur Zeit bekannten Befehlswoorte, die einem oder mehreren Prototypen zugeordnet sind. Möchte man alle Prototypen eines Wortes löschen, markiert man zunächst das entsprechende Wort in der Liste und drückt anschließend den Button `Delete` unterhalb der Liste.

Mittels `Clear` werden sämtliche Prototypen aus der Datenbank gelöscht.

Die Steuerelemente in der Mitte der Applikation ermöglichen Laden und Speichern der Prototypen sowie der Konfiguration.

Der größte Teil der Applikation besteht (unten links) aus Steuerelementen zur Einstellung der Spracherkennungsparameter. Beschreibungen zu den jeweiligen Parametern sind Abschnitt 4.3 zu entnehmen.

Durch Drücken von `Measure Silence` mißt das System den aktuellen Geräuschpegel. Während dieser Zeit sollte nicht gesprochen werden.

Achtung! Durch bloßes Verändern der Eingabefelder für die Parameter wird noch keine Änderung der Parameter vorgenommen. Erst durch Druck auf `Change now!` werden *alle* Veränderungen auf einen Schlag zum Hear-Server gesendet. Dieses Vorgehen ist notwendig, da manche Parameter voneinander abhängen. Insbesondere gilt dies für die Parameter `frameDimension` und `middleFreqs`.

Kapitel 6

Erkennen der Befehle

6.1 Einfaches Beispielprogramm zur Erkennung

```
extern "C" {
#include "hearclient.h"
}
#include <stdio.h>
#include <stdlib.h>

void DO (int result, const char* errmsg)
{
    if (!result) {
        printf("Error: %s\n", errmsg);
        exit(1);
    }
}

int main (int argc, char** argv)
{
    RecognizeInfo info;

    DO( hear_init(), "init" );
    DO( hear_loadConfig("standard.cfg"), "load config" );
    DO( hear_loadProtos("test.prt", 0), "load protos" );

    while (1) {
        if (hear_getSpeech() && hear_recognize(&info)) {
            if (info.accepted && info.word[0] != 0)
                printf("recognized: %s\n", info.word);
        }
    }

    return 0;
}
```

6.2 Einfaches Beispielprogramm zur parallelen Erkennung

Genau wie bei dem Beispielprogramm aus dem letzten Abschnitt handelt es sich auch hier um ein *sehr* einfaches Programm – praktisch nur um ein Programmgerüst. Wegen der Größe dieses Programmes wurde es auf mehreren Seiten abgedruckt.

```
#include "hearclient.h"

#include <stdio.h>
#include <sys/time.h>
#include <sys/select.h>

void child_func (int pipe_in, int pipe_out)
{
    RecognizeInfo info;
    char word[40];
    int speechAvail;
    double silence;

    struct timeval timeout;
    fd_set fdset;

    if (!hear_init())
        { printf("error initializing\n"); exit(1); }
    if (!hear_loadConfig("standard.cfg"))
        { printf("error loading config\n"); exit(1); }
    if (!hear_loadProtos("test.prt", 0))
        { printf("error loading protos\n"); exit(1); }
    if (!hear_measureSilence(&silence))
        { printf("error measuring silence\n"); exit(1); }
    printf("silence = %f\n", (float)silence);

    while (1) {
        speechAvail = hear_getSpeech();
        //printf("speechAvail = %i\n", speechAvail);
        if (speechAvail && hear_recognize(&info) &&
            info.accepted && info.word[0] != 0) {
            strcpy(word, info.word);
            printf("child: recognized %s\n", word);
            if (write(pipe_out, word, 40) < 40) {
                printf("child: error or cut off\n");
                break;
            }
        }
    }

    printf("child ready\n");
}
```

```
void daddy_func (int pipe_out, int pipe_in)
{
    int i;
    char word[40];
    int len;
    struct timeval timeout;
    fd_set fdset;
    struct timespec sleeptime;

    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = 10000000; /* 10 ms */

    while (1) {
        FD_ZERO(&fdset);
        FD_SET(pipe_in, &fdset);
        timeout.tv_sec = 0;
        timeout.tv_usec = 10;
        i = select(FD_SETSIZE, &fdset, NULL, NULL, &timeout);
        if (i > 0) {
            if (read(pipe_in, word, 40) <= 0) {
                printf("daddy: error or cut off (2)\n");
                break;
            }
            printf("child sent %s\n", word);
        } else if (i < 0) {
            printf("daddy: error or cut off (1)\n");
            break;
        } else {
            // doing something...
            nanosleep(&sleeptime, NULL);
        }
    }

    printf("daddy ready(2).\n");
}
```

```
void main ()
{
    int pid;
    int fd_pipe[2];      /* [0]: read end (child) <-- [1]: write end (daddy) */
    int fd_pipe2[2];    /* [0]: read end (daddy) <-- [1]: write end (child) */

    if (pipe(fd_pipe) != 0) {
        printf("Could not open a new pipe (1)!\n");
        exit(1);
    }

    if (pipe(fd_pipe2) != 0) {
        printf("Could not open a new pipe (2)!\n");
        exit(1);
    }

    pid = fork();

    if (pid != 0) {
        /* daddy */
        close(fd_pipe[0]);
        close(fd_pipe2[1]);
        daddy_func(fd_pipe[1], fd_pipe2[0]);
    } else {
        /* child */
        close(fd_pipe[1]);
        close(fd_pipe2[0]);
        child_func(fd_pipe[0], fd_pipe2[1]);
    }
}
```

Kapitel 7

Fazit

Nach anfänglichen Schwierigkeiten, die richtigen Filterbänke und einen Algorithmus zum Vergleichen zweier Prototypen (Dynamische Zeitverzerrung) zu finden, stellten sich die Erfolge relativ schnell ein.

Es hat sich gezeigt, daß die Einzelworterkennung *bei gleichem Sprecher* sehr zuverlässig funktioniert. Darüber hinaus ist sogar teilweise eine sprecherunabhängige Erkennung möglich, wenngleich dieses System nicht dafür ausgelegt ist. Sicherheitshalber sollten dennoch alle Sprecher, die dieses System nutzen wollen, ihre eigenen Sprachproben in die Datenbank aufnehmen. Dann stellt die Erkennung aller Befehls Worte für alle Sprecher kein Problem mehr dar.

Einige Worte lassen sich nur schwer oder gar nicht trainieren, zum Beispiel das Wort STOP. Dieses Wort besteht fast ausschließlich aus Zisch- und Knacklauten sowie Innerwortpausen und ist dabei noch extrem kurz. Dies ist eine Mischung, die ein Training dieses Wortes mit dem hier vorgestellten System nahezu unmöglich macht. Man sollte daher stattdessen zum Wort HALT übergehen. Sicherlich existiert für jedes schlecht zu trainierende Wort ein Synonym, welches sich besser trainieren läßt.

Außerdem sollte darauf geachtet werden, daß sich die Befehls Worte gut unterscheiden. Es ist zwar möglich, beispielsweise sowohl LANGSAM als auch LANGSAMER zu trainieren. Hierbei handelt es sich sogar um ein Beispiel von zwei ähnlichen Worten, die das System trotzdem gut unterscheiden kann, aber die Worte SCHNELL und SCHNELLER werden schon nicht mehr so gut unterschieden. Auch hierbei gilt es also, notfalls auf Synonyme auszuweichen, so daß sich wirklich alle Worte paarweise gut unterscheiden. Ob sich Worte gut unterscheiden, läßt sich schnell mit der Trainingsapplikation (siehe Abschnitt 5.1) nachprüfen.

Nach Betrachtung der Fakten ist die Spracherkennung zur Robotersteuerung geeignet, falls folgende Voraussetzungen erfüllt sind:

- Die Anzahl der Befehls Worte wird nicht zu groß gewählt.
- Die Worte werden so gewählt, daß sich alle paarweise gut unterscheiden.
- Die Anzahl der Personen bleibt klein.
- Der Geräuschpegel benachbarter, sich unterhaltender Personen ist nicht zu hoch.

Literaturverzeichnis

- [Hil97] Hermann Hild. *Buchstabiererkennung mit Neuronalen Netzen in Auskunftssystemen*. Dissertation. Shaker Verlag, May 1997.
<http://www.is.cs.cmu.edu/papers/speech/pdh-thesis/thesis-hermann.html>.
- [Nud] Matthew S. Ryan & Graham R. Nudd. *Dynamic Character Recognition using Hidden Markov Models*. University of Warwick, Dep. of Computer Science. INF 199 382-244.
- [ST95] E. G. Schukart-Talamazzini. *Automatische Spracherkennung – Grundlagen, statistische Modelle und effiziente Algorithmen*. Vieweg, 1995. ELT 976/036.
- [Teb95] Joe Tebelskis. *Speech Recognition using Neural Networks*. PhD thesis, School of Computer Science, Carney Mellon University, Pittsburgh, Pennsylvania, May 1995.
- [Wit82] I. H. Witten. *Principles of Computer Speech*. Computers and People Series. Academic Press, London, 1982.