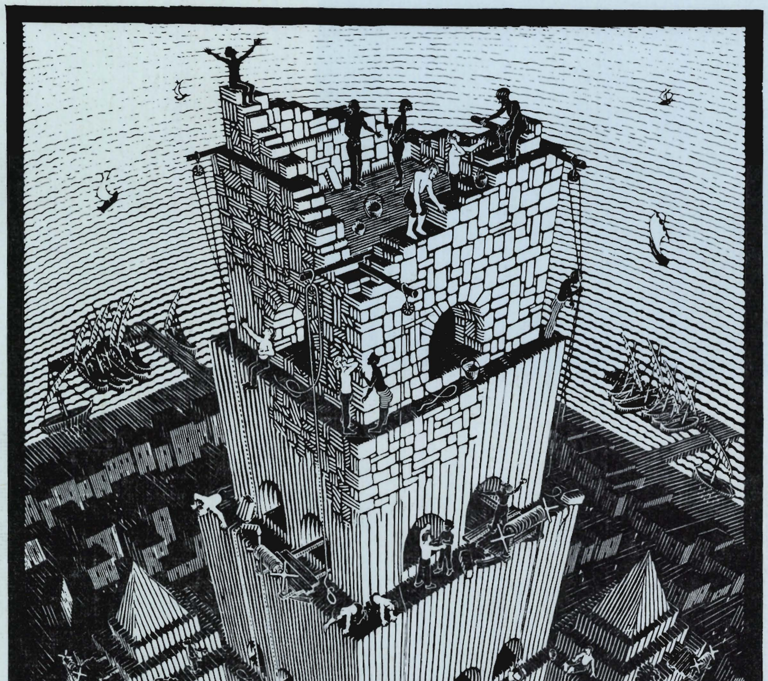
Speech recognition by recursive stochastic modelling

DOI:
10.6100/IR372913

Document status and date:
Published: 01/01/1992

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

SPEECH RECOGNITION BY RECURSIVE STOCHASTIC MODELLING



J.J. NIJTMANS

SPEECH RECOGNITION BY RECURSIVE STOCHASTIC MODELLING

At the cover:

Tower of Babel, by M.C. Escher (woodcut, 1928)

SPEECH RECOGNITION BY RECURSIVE STOCHASTIC MODELLING

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op woensdag 6 mei 1992 om 16.00 uur

door

JOANNES JOSEPHUS NIJTMANS
Geboren te Oisterwijk

Dit proefschrift is goedgekeurd
door de promotoren

prof. dr. ir. W.M.G. van Bokhoven
en
prof. dr. ir. A.C.P.M. Backx

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Nijtmans, Joannes Josephus

Speech recognition by recursive stochastic modelling /
Joannes Josephus Nijtmans. - [S.l. : s.n.]. - Ill., fig., tab.
Proefschrift Eindhoven. - Met lit. opg., reg.- Met
samenvatting in het Nederlands.
ISBN 90-9005038-8
NUGI 832
Trefw.: spraakherkenning (computertechniek).

to my parents

SUMMARY

The study of the application of algorithms for automatic speech recognition, as described in this thesis, consists of two parts. The first part is concerned with finding suitable parameters to describe a speech signal. Log area ratio parameters were chosen. Based on these parameters, a segmentation algorithm for speech is developed.

The second part discusses the recognition process. A new method is developed based on a new model: the Recursive Markov Model (RMM). This method is an extension of the existing Hidden Markov Model (HMM). It has some important advantages:

- Shorter computations. By using a flexible form of state pruning, various states can be temporary removed during the training and recognition process.
- Hierarchical modelling. All levels (like the syntax- , phoneme-level) share the same structure.
- State sharing. For instance syllables that are used in various different words need only be defined once. This sharing is possible at all levels, and constitutes a considerable improvement, especially when extensive vocabularies are used.

Two new algorithms are developed for RMM-based training and recognition. Based on training sequences, the Recursive Forward Backward Training algorithm adjusts all parameters. Supervised training is possible. Like the corresponding Forward Backward algorithm as used in HMM, the RMM-algorithm yields a local maximum of the likelihood function. The Recursive Viterbi algorithm is developed for speech recognition; by applying this algorithm speech can be segmented and recognized simultaneously. The Recursive Viterbi algorithm is therefore more effective in speech segmentation than is the segmentation algorithm developed in the first part of this thesis.

The speech analysis algorithm is implemented on a TMS320C25 Digital Signal Processor (DSP). The log area ratio parameters can be calculated in real-time. Because the memory of the used DSP-system was too small, the remaining algorithms are not implemented on this DSP. The segmentation algorithm developed in the first part is implemented using

MATLAB on a PC. This algorithm locates segment border locations very accurately; the problem is that too many borders are missed and too many extra borders inserted. For correction of these mistakes more information about the speech signal is needed, which is not available.

The Recursive Forward Backward Training algorithm and the Recursive Viterbi algorithm are implemented using ANSI-C on an APOLLO DN3000 workstation. The software is hardware independent, so it can be run by any system using ANSI-C and having sufficient memory.

Two languages are used to test the algorithms. First the syntax of the numbers 1 to 999999 was converted to an RMM-structure. The Recursive Viterbi algorithm was found to recognize sequences whose parameters were disturbed by random Gaussian noise, using the syntax constraints available. The calculation times needed for recognition were short (10 sec .. 3 min.), dependent on the desired accuracy. The training algorithm was tested and found to work very well.

Another language, the Speech Controlled Robot-language (SCR), is converted to an RMM. Tests show that this grammar is very suitable for RMM speech recognition.

SAMENVATTING

Het onderzoek aan algoritmen voor automatische spraakherkenning, zoals beschreven in dit proefschift, bestaat uit twee gedeelten. Het eerste deel houdt zich bezig met het beschrijven van een spraaksignaal door middel van een geschikte set parameters. Een keuze is gemaakt voor de log area ratio parameters. Er is een segmentatie-algoritme ontworpen gebaseerd op deze parameters.

Het tweede gedeelte beschijft het herkenningsproces. Er is een nieuwe methode ontwikkeld, gebaseerd op een nieuw model: het Recursive Markov Model (RMM). Deze methode is een uitbreiding van het bestaande Hidden Markov Model (HMM), met de volgende voordelen:

- Vermindering van rekentijd. Onwaarschijnlijke toestanden kunnen tijdelijk verwijderd worden als de kans laag genoeg is.
- Hiërarchische modellen worden ondersteund, waarin zowel hoge als lage niveaus (syntax / phonemen) dezelfde structuur hebben.
- Hergebruik van modellen is ondersteund. Bijvoorbeeld lettergrepen die in meerdere woorden voorkomen hoeven slechts één keer gedefinieerd te worden. Dit is mogelijk op alle niveaus, en leidt vooral bij een grote woordenschat tot een aanzienlijke geheugenbesparing.

Twee nieuwe algoritmen zijn ontwikkeld voor training en herkenning op basis van het RMM. Het Recursive Forward Backward algoritme is in staat alle parameters te trainen enkel op basis van trainingszinnen. Ook 'supervised' training, waarbij de identiteit van elk woord gegeven wordt, is mogelijk. In dit proefschift wordt een bewijs afgeleid, dat dit algoritme altijd convergeert naar een lokaal maximum van de gedefinieerde waarschijnlijkheidsfunctie, zoals ook het overeenkomstige HMM-algoritme doet. Het Recursive Viterbi algoritme is de basis voor de herkenning. Met dit algoritme kan tegelijkertijd met de herkenning een segmentatie uitgevoerd worden, ook rekening houdend met informatie over hogere niveaus. Daarom bleek dit algoritme beter te zijn dan die in het eerste deel.

Het algoritme voor spraak-analyse is geïmplementeerd op een TMS320C25 digitale signaalprocessor (DSP). De log area ratio parameters kunnen in real-time berekend worden met behulp van de geschreven software. Omdat het geheugen van het DSP-systeem te klein was, zijn de segmentering en herkenning niet hierop geïmplementeerd. Het segmentatie-algoritme, geschreven in MATLAB op een PC, is in staat zeer nauwkeurig segmentgrensposities te bepalen; alleen is het probleem dat te veel grenzen overgeslagen worden en ook te veel toegevoegd. Om deze fouten te corrigeren is meer informatie over het spraaksignaal nodig, wat niet beschikbaar is.

Het Recursive Forward Backward Training algoritme en het Recursive Viterbi algoritme zijn geïmplementeerd in ANSI-C op een APOLLO DN3000 workstation. De software is hardware-onafhankelijk, dus het kan gebruikt worden op ieder systeem met ANSI-C en genoeg geheugen.

Twee abstracte talen zijn gebruikt om de algoritmes te testen. Allereerst is de syntax van de nummers 1 tot 999999 (Engels) geconverteerd naar een RMM-struktuur. Het Recursive Viterbi algoritme bleek sequenties, waarin de parameters verstoord waren door Gaussisch verdeelde ruis, zeer nauwkeurig te kunnen herkennen, gebruik makend van de beschikbare syntax-informatie. De rekentijden benodigd voor deze herkenning waren kort (10 sec .. 3 min.), afhankelijk van de vereiste nauwkeurigheid. Ook de training is getest, en bleek naar verwachting zeer goed te werken.

Een andere abstracte taal, de Speech Controlled Robot-taal (SCR) is omgezet naar een RMM. De gedane tests laten zien dat deze taal zeer geschikt is om gebruikt te worden in een spraakherkenningssysteem volgens het RMM-principe.

CONTENTS

LIST OF USED SYMBOLS

DSP	Digital Signal Processor
DTW	Dynamic Time Warping
FFT	Fast Fourier Transform
FSM	Finite State Machine
HMM	Hidden Markov Model
LPC	Linear Predictive Coding
LSP	Line Spectral Pair
RMM	Recursive Markov Model
SCR	Speech Controlled Robot
TM	Template Matching
t	Time index
T	Length of time interval
s_t $(\tilde{s}_t)$	Signal sample, (predicted signal sample)
e_t	Error sample
ϕ_{ij}	Correlation coefficient
R_i	Autocorrelation coefficient
$a_i^{(p)}$	Filter coefficient (p = order)
$b_i^{(p)}$	'Split' filter coefficient (p = order)
$n_i^{(p)}, d_i^{(p)}$	'Schur' variables
$q_i^{(p)}$	'Split Schur' variables
k_i	Reflection coefficient
β_i	'Split reflection' coefficient
g_i	Log area ratio coefficient
$d(i,t)$	Distance between symbols U_i and Y_t.
$D(i,t)$	Cumulative distance between $U_1..U_i$ and $Y_1..Y_t$, along best path.
$G(i)$	Centre of Gravity function (see Section 3.4)
A	Matrix containing transition probabilities a_{ij}, a_{Ij} and a_{iF}.
a_{ij}, $(\hat{a}_{ij})$	(Estimated) Transition probability from state S_i to S_j.
a_{Ij}	Transition probability to state S_j, after S is initiated.
a_{iF}	Probability that state S has finished, after S_i has finished.
b_{im}	Probability that state S_i produces symbol m. (HMM)

In the following symbols, '·' stands for any index:

$P^f_\cdot(S,t)$	Forward probability of state S at time t.
$P^b_\cdot(S,t)$	Backward probability of state S at time t.
$P^v_\cdot(S,t)$	Viterbi probability of state S at time t. (=probability of best path)
$P^\cdot(S,t)$	Probability that state S is active at time t. (except for backward probabilities, see Section 5.2)
$P^\cdot_I(S,t)$	Probability that state S is initiated at time t.
$P^\cdot_F(S,t)$	Probability that state S has finished at time t.
$\tilde{P}^\cdot_\cdot(S,t)$	Probability, scaled by its parent state on time t.
$\hat{P}^\cdot_\cdot(S,t)$	Probability, scaled by its parent state at the previous iteration (see Section 6.2)
$P_E(S,\bar{u})$	Probability that symbol $\bar{u}$ is produced, given state S.
Y_t	Discrete output symbol at time t
$\bar{y}_t$	Continuous output symbol (vector) at time t
τ_{ij}	Expected number of transitions a_{ij} taken during training
υ_{im}	Expected number of occurrences of symbol m at state S_i. (HMM)
w_t	Probability that state S is active at time t
τ	Expected number of times that state S is active, see (5.24)
$\overline{m}$	Weighted sum of observation vectors, see (5.27)
V	Weighted sum of observation vector correlations, see (5.27)
$\bar{\mu}$, $(\bar{\mu}_i)$	mean vector (for state S_i)
Σ, (Σ_i)	Autocorrelation matrix (for state S_i)
H(S)	Entropy of state S
L(S)	Average duration of state S
u_i	Relative probability of state S_i to be initiated, with respect to parent state S

1. INTRODUCTION

One of the most popular imaginations of human beings are machines that can act and think like humans. Many Science Fiction films and books show fully operational robots or speaking computers, assisting humans in their everyday work. Apart from the question whether this would be desirable or not, it is highly questionable whether such machines could actually be built. However, in the near future there are many useful applications for speech processing combined with artificial intelligence.

Nowadays, many machines are operated by keyboard and display. For some people this operation entails severe restrictions. Speech would be a much more natural means of operating machines. It would be a lot simpler if money could be drawn from a bank by voice, instead of typing the required amount and the PIN-code. Bank transactions could be done by telephone, allowing the identity of the speaker to be checked simultaneously. For blind people, being much more dependent on speech, a voice controlled robot could become a valuable help for some specific tasks. All kinds of apparatus would be more readily accessible to much more people.

The history of speech recognition starts in the late 1960s, when available computer power made it feasible. The first computer systems used a recognition methodology known as dynamic time warping. The restrictions in the first systems were:

- Only isolated words were recognized.
- Only a single person could be understood.
- Only a limited vocabulary (20-50 words) was available.

The first commercial speech recognition systems appeared in the early 1970. The VIP 100 developed by Threshold Technology Inc. even won a US National award in 1972. However, because of the limited computer power available in those days, systems could hardly be improved without ending up with unacceptable processing times.

In 1971 a project was started (ARPA-SUR) to study the use of upper-level information, like syntax, for increasing the realm and the quality of a speech recognizer. When the project was finished, in 1976, several systems had been built. The most successful one, the HARPY system [28], required so much computer power that it could not be used in practice. The main conclusion was that acoustic-phonetic modelling was still too primitive, because it was not able to use upper-level information.

After the classical paper of Baum in 1972 [10], the Hidden Markov Model (also used in HARPY) became very popular. Human speech, however, is so complex that the basic HMM is not powerful enough to handle aspects like:

- Connected speech. When words are pronounced without a pause, the pronunciation of one word can influence the pronunciation of the words following and preceding it.
- Extensive vocabulary. Each word (or other chosen unit) requires a separate model, so the memory use grows proportional to the size of the vocabulary.
- Duration modelling. The exponential duration of the basic HMM is in conflict with the physical duration of real speech.
- Acoustic-Phonetic preprocessing. As the basic HMM is discrete while speech is continuous, some preprocessing is necessary.

Several solutions to the above problems have been found, resulting in systems like Tangora [4] and SPHINX [30]. These systems use specialized hardware to execute their complex algorithms in real time. These methods use some basic unit (like word, syllable, diphone). Each unit has a separate HMM as its model, which has internal states representing the speech frames. Further improvements have been realized on two scores:

- The HMM state model. Usually successive speech frames are highly correlated. Proper segmentation or more accurate duration modelling could result in calculation reductions or improve the quality of the succeeding HMM recognizer.
- The Language model. Imposing constraints requiring the sentences to be valid English, also reduces the number of words to be searched.

In 1988, the Circuit Design group (EEB) of the University of Technology in Eindhoven started a research project on speech recognition with the idea that classical signal-processing systems, constructed with dedicated hardware and circuit/system design methods could yield an improved system for speech recognition. The group had some past experience in Linear Predictive Coding speech analysis (LPC). Among others, the Schur algorithm, for calculating the reflection coefficients, has been implemented on a TMS 32010 processor [45]. These reflection coefficients show characteristics useful in a speech recognition system. This idea lead to the current investigation of speech recognition algorithms, which forms the basis for this thesis. However, no specific use is made of the reflection coefficients any more. A more general approach is taken, that allows many parameter sets, including LPC-parameters.

The research resulted in a new Recursive Markov Model (RMM), which can be seen as a generalization of the Hidden Markov Model (HMM). The algorithms for training and recognition of HMMs are extended to be used with the RMM. These extended algorithms, using additional features of the RMM, do not suffer from some of the drawbacks of HMM-algorithms.

Speech analysis has preceded the recognition. The analysis algorithm extracts a set of parameters from the speech signal, thus capturing its most important features. Many parameter sets are feasible. Good parameter sets have a smaller information content than the original speech signal, while yielding adequate description for training and recognition. LPC-parameters are good, but there are alternatives.

The feature parameters are needed both in the training and the recognition algorithm. They represent the speech signal. Apart from these parameters, the RMM contains a set of model parameters. For these parameters a training process is needed. Using training sequences, a Maximum Likelihood estimate for the RMM parameters can be found. After training the full model, a recognition algorithm compares the analyzed speech signals with the trained RMM. This algorithm results in a sequence of RMM states that fits the speech signal best. This state sequence constitutes the desired recognition result.

This research is treated in a number of chapters that can be summarized as follows:

Chapter 2 considers the possibilities of efficient implementation of the analysis algorithms. The Schur algorithm is implemented on a Digital Signal Processor (DSP), of type TMS 320C25. The LPC-parameters, and several other parameter sets suitable for use in a speech recognition system are compared. Especially an integrated derivation of the Levinson and Schur algorithms and its "split" counterparts are given. The used method stresses, more then others, the strong relation between the various parameter sets. This relation can be expressed in matrix transformations only.

Some other components of a traditional speech recognition system, like vector quantization and segmentation, are discussed in Chapter 3. The described techniques are similar to the ones used in the first systems to appear on the market around 1970. The various components give a good indication of the complexity of the speech recognition problem. Despite of the restricted model, the number of calculations that have to be executed is enormous. Therefore it is necessary to look for algorithms that save calculations wherever possible, otherwise no practical system can be realized.

In Chapter 4 the Dynamic Time Warping method (DTW) and the Hidden Markov Model (HMM) are discussed in greater detail. Especially, the relation between DTW and the Viterbi algorithm is explained. Also extensions like 'duration modelling' and the use of continuous symbols are discussed.

A new extension to HMM is derived in Chapter 5. In the Recursive Markov Model (RMM) two already mentioned deviations of the basic HMM approach are integrated. 'duration modelling' and 'language processing' can now be implemented simultaneously.

The principles of Scaling and State pruning are introduced in Chapter 6. With the help of these principles the practical problems that arise in Chapter 5 can be solved.

Chapter 7 describes the implementation of the Schur algorithm, the Recursive Forward Backward algorithm and the Recursive Viterbi algorithm. Several comparative tests are executed with these algorithms.

Finally, Chapter 8 gives the result of several tests of the implemented algorithms and considers relative merits of the methods of speech recognition or other fields. Suggestions are also given for a follow-up.

2. THE SPEECH ANALYSIS PROBLEM

The initial step in speech recognition consists of speech analysis. The incoming speech is divided into frames, which are represented by a set of parameters. The aim of this representation is to describe the speech signal with less bits, without loosing the information necessary for successful recognition. One important consideration is the choice of the parameter set to be used. The best known sets are the LPC-parameters and its variants. These parameters are all derived from the Yule-Walker equation. Well known algorithms for calculating the LPC-parameters are the Levinson and Schur recursion algorithms. Recently, the more efficient "split" algorithms were discovered.

In this chapter we consider four algorithms: the Levinson, Schur, Split Levinson and Split Schur algorithm. These algorithms can be derived along the same lines, the only difference being the set of variables. A matrix transformation can map each parameter set onto the others.

Each algorithm is evaluated for implementation on a TMS320C25 DSP. This processor is specially designed for fast matrix operations. We will show that on the TMS320C25 the "Split" algorithms have no advantage. The main reason is the extra division operations which are needed to calculate the reflection coefficients. If the number of coefficients is low, these divisions take a large percentage of time. Therefore, only the Schur algorithm is implemented.

2.1 The Speech Production Model

Before speech can be recognized, it must be converted into an electrical signal. After sampling and quantization, further digital processing is possible. The sequence of quantized speech samples can be seen as a stochastic signal, which can be represented by a suitable set of parameters.

An often used characterization procedure originates from the following speech production model, which describes how the speech signal is produced physically. The human speech production system consists of the vocal chords, followed by a flexible tube (the vocal tract) which ends at the lips. The sound wave originating from the mouth can be seen as a function of the vibration of the vocal chords, the shape of the vocal tract and the radiation at the lips.

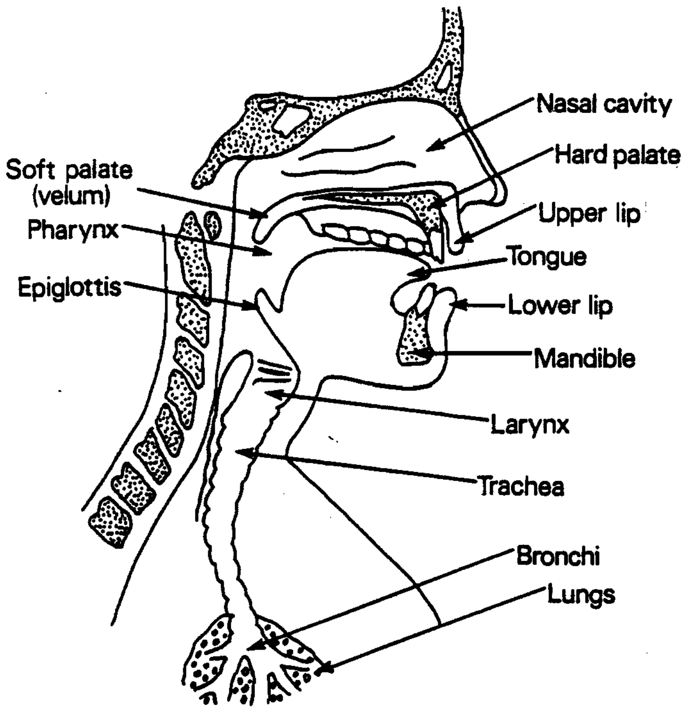


Fig. 2.1: The human vocal tract

For vowels (e.g. 'a','o') the vocal chords produce a periodic sound wave, that is filtered by the vocal tract. For fricatives ('s','f') a noise excitation is formed further forward in the vocal tract. The speech production can also be modelled by an excitation source - producing a periodic pulse train or noise - followed by a filter. Some systems also allow a combination of these two. For instance 'z' is periodic, but contains a lot of noise generated inside the mouth.

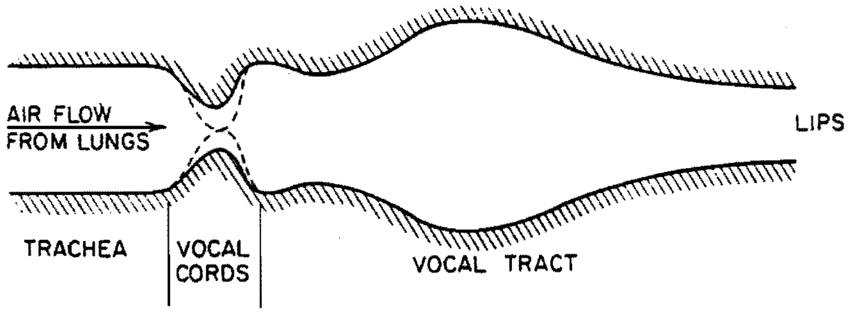


Fig. 2.2: The vocal tract model

The spectrum of the excitation is generally not white. However, any periodic wave form can be produced by filtering a periodic pulse train using the appropriate filter. In the same way any coloured noise can be produced by filtering white noise. Without any restriction we can combine this filter with the vocal tract filter.

Another simplification arises when we restrict the filter to the all-pole type. This is equivalent to supposing that the filter consists of a single tube, neglecting airflow through the nose. In this way difficulties arise with producing nasal sounds like 'm' and 'n'. However, most sounds are produced with the soft palate (see Fig. 2.1) closed, so that the nasal cavity does not contribute to the final sound emitting the mouth.

Usually the speech data will be available in sampled form. Sampling frequencies normally used range from 6.67 to 20 kHz. This means that for the filter a discrete model can be used, equivalent to an acoustic tube consisting of multiple fixed length sections, each having its own diameter. The transfer function of such a filter can be modelled by an all-pole filter.

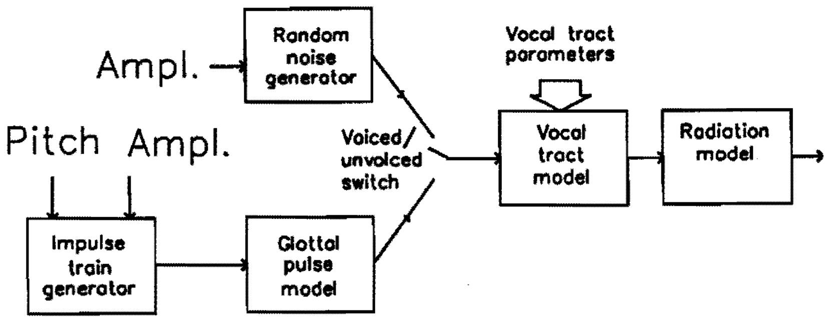


Fig. 2.3: The simplified vocal tract model

With these simplifications we arrive at the speech production model of Fig. 2.3. Having determined the voiced/unvoiced bit, the pitch, the amplitude and the width of each tube section, all information needed about the character of the speech signal is available.

2.2 Various Parameter Sets for Speech Characterization

The above speech production model is common in literature. For the various parts (excitation, filtering) suitable parameter sets can be developed, which are as good a description of the speech signal as are its quantized samples. Specific use can be made of the fact that speech is either noise-like or periodic.

If the excitation is noise-like, it can be described as a white noise generator followed by a filter. The noise generator only has to be specified with a single parameter, its amplitude, which is slowly varying in time. The filter can be combined with the vocal tract filter. If the excitation is periodic, the noise generator is replaced by a pulse source, fully specified with its amplitude and its period. Hence, apart from the filter, only three additional parameters are needed:

- Voiced/Unvoiced bit, specifying the type of excitation.
- Amplitude of the excitation source.
- The Period of the excitation source, in case of voiced sounds. This Period is known as the pitch.

For speech recognition these parameters are not so interesting. The amplitude and the pitch of the speech signals have no influence on the identity of spoken words. At first sight the voiced/unvoiced bit seems more important, but in fact it is not. If in the speech production the pulse excitation is replaced by white noise, the effect is similar to whispering. Without any pitch, we are still able to understand speech. Therefore our main concern will be the vocal tract filter parameters.

The transfer function of the vocal tract filter can be specified in many ways. One possibility is to suppose that the filter is of the all-pole type, and to use the coefficients of this filter, as suggested in Fig. 2.3. This results in the so-called LPC-parameters (Linear Predictive Coding). This name derives from an alternative way of viewing these parameters, in which each sample s_t is supposed to be predicted by a linear combination of past samples. This assumption leads to a prediction error filter, which can be shown to be the inverse of the vocal tract filter. This is worked out in more detail in Section 2.3. Many alternative parameter sets can be derived from the LPC-parameters.

Another way of specifying the vocal tract filter, is splitting the signal in separate frequency bands. The energy in each band can be used for specification of the speech signal, because the human ear is insensitive to phase-information contained in the speech signal. If the frequency bands are chosen equi-distant, then the spectrum can be calculated by the Fast Fourier Transform (FFT). A choice has to be made about the number of frequency bands and the number of bits used in each band.

A typical example of a system using such an approach is the Ensigma SRS-1 [12] specifying a sampling rate of 8 kHz, using a standard telephone codec IC delivering 8 bits/sample (such as the TCM 29C18). The bit rate of the input signal is 64 kbit/s. This signal is filtered by a bank of ten fourth-order bessel filters, whose centre frequencies are spaced uniform on a logarithmic scale between 300 and 3200 Hz. The power estimate of each frequency band is calculated each 20 ms in logarithmic form with 16 bits. The bit rate after this filter bank is 8 kbit/s. Without this reduction it would not be possible to perform speech recognition in real time using a single DSP type TMS320C17, even with the limited library of only 12 words.

In this thesis a choice has been made to use LPC-derived parameters. They have the advantage that the operations involved are mainly matrix operations, that can be implemented very efficiently on a DSP. How this is done, will be shown in the remaining part of this chapter.

2.3 The LPC method, the Yule-Walker equation

The Linear Predictive Coding (LPC) theory assumes that a speech signal can be produced by a signal source (white noise or periodic delta pulses) followed by an all-pole filter. Comparison with the human speech production shows that this is a logical assumption, since the vocal chords behave like a signal source and the vocal tract behaves like a filter. If the vocal tract behaves like a single tube without energy loss in the sections, this can be represented by using an all-pole filter.

Let us try to predict sample s_t from the speech signal, using previous samples $s_{t-p}..s_{t-1}$. We suppose that the first sample is available at time t=1, and that before this time all sample values are zero. The prediction of future samples can be done applying a transversal filter with coefficients $-a_j$ (j=1..n). The minus sign is used here, to prevent minus signs to appear in formula (2.2):

$$\tilde{s}_t = \sum_{i=1}^{n} -a_i \cdot s_{t-i} = -\sum_{i=1}^{n} a_i \cdot s_{t-i} \qquad (2.1)$$
$$\text{where } s_t = 0 \text{ for } t<1$$

The prediction error e_t can be written as :

$$e_t = s_t - \tilde{s}_t =$$

$$s_t - \sum_{i=1}^{n} -a_i \cdot s_{t-i} = \sum_{i=0}^{n} a_i \cdot s_{t-i} \tag{2.2}$$

with: $a_0 = 1$.

This can be seen as a filter, the **prediction error filter**. Its inverse, which is an all-pole filter, can be used to restore the speech signal from the prediction error:

$$s_t = \tilde{s}_t + e_t = e_t + \sum_{i=1}^{n} -a_i \cdot s_{t-i} \tag{2.3}$$

In order to estimate the filter coefficients a_i, the power E of the error signal e_t is minimized during a given sample interval t=1..T. This energy is defined as:

$$E = \sum_{t=1}^{T} e_t^2 \tag{2.4}$$

The second derivative of E with respect to any coefficient a_i can be calculated as:

$$\frac{\partial^2 E}{\partial a_i^2} = 2 \cdot \sum_{t=1}^{T} s_{t-i}^2 \qquad \{i = 1..n\} \tag{2.5}$$

Because this second derivative is always positive, if there exists a point where the first derivative $\partial E/\partial a_i = 0$, this must be a unique minimum. This minimum can be determined by zeroing the derivatives of E with respect to all coefficients a_i (i=1..n).

$$\frac{\partial}{\partial a_i} \sum_{t=1}^{T} e_t^2 = 0 \Rightarrow$$

$$\frac{\partial}{\partial a_i} \sum_{t=1}^{T} \left(s_t + \sum_{j=1}^{n} a_j \cdot s_{t-j} \right)^2 = 0 \Rightarrow$$

$$\sum_{t=1}^{T} 2 \cdot \left(s_t + \sum_{j=1}^{n} a_j \cdot s_{t-j} \right) \cdot s_{t-i} = 0 \Rightarrow$$

$$\sum_{j=1}^{n} a_j \cdot \sum_{t=1}^{T} s_{t-j} \cdot s_{t-i} = - \sum_{t=1}^{T} s_t \cdot s_{t-i} \tag{2.5}$$

If we define the following correlation coefficients ϕ_{ij} as:

$$\phi_{ij} = \sum_{t=1}^{T} s_{t-j} \cdot s_{t-i} \qquad \{i,j = 0..n\} \tag{2.6}$$

Then we can re-write (2.5) as:

$$\sum_{j=1}^{n} a_j \cdot \phi_{ij} = -\phi_{i0} \qquad \{i = 1..n\} \tag{2.7}$$

The coefficients ϕ_{ij} can be calculated in about $O(n \cdot T)$ multiplications and additions. The n equations with n unknown coefficients (2.7) can be solved in $O(n^3)$ multiplications and additions. The calculation of the correlation coefficients ϕ_{ij} costs significantly more time than the solving of the set of equations if $T \gg n^2$, which is usually the case.

A significant calculation reduction can be achieved if an additional assumption is made. If the sample interval considered is large enough, then ϕ_{ij} becomes almost the same as $\phi_{i-1,j-1}$. We could as well replace all ϕ_{ij} by autocorrelation coefficients R_{i-j}, where:

$$R_i = \sum_{t=i+1}^{T} s_t \cdot s_{t-i} \qquad \{i = 0..n\} \tag{2.8}$$

Another way to view this simplification is extending the sample interval from $t=-\infty..\infty$, pre-multiplying the samples by a window function which is zero outside the region $t=1..T$. The number of coefficients to be computed is reduced from $\frac{1}{2}(n^2+n)$ to n. Equation (2.7) can also be written as a Toeplitz matrix equation:

$$\begin{pmatrix} R_0 & \cdots & R_{n-1} \\ \vdots & \ddots & \vdots \\ R_{n-1} & \cdots & R_0 \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = - \begin{pmatrix} R_1 \\ \vdots \\ R_n \end{pmatrix} \tag{2.9}$$

This equation, the **Yule-Walker** equation, forms the basis of the **auto-correlation** method. Because this matrix equation has a special shape (Toeplitz), a significant reduction in calculations is possible from $O(n^3)$ to only $O(n^2)$. Several related algorithms are derived in the following sections.

2.4 The Levinson recursion

The Levinson approach to determine the LPC-coefficients a_i consists of an iterative procedure, replacing equation (2.9) with a sequence of solutions of lower order Yule-Walker equations, ranging from orders 1 to n. Instead of solving the n^th^-order Yule-Walker equation directly, we start with the first order. Then we have simply:

$$R_0 \cdot a_1^{(1)} = -R_1 \quad \Rightarrow \quad a_1^{(1)} = -R_1/R_0 \tag{2.10}$$

The additional index of the LPC-coefficients denotes the order of the currently solved equation. If it is possible to describe the p^th^ order solution of the Yule-Walker equation in terms of the p-1^th^ order, we can reach the desired n^th^-order solution by iterating this procedure over every order, starting with p=1.

Consider the following p^th^-order problem, which consists of the Yule-Walker equation. In the first row, an extra variable $d_0^{(p)}$ is added, through its defining equation. The advantage of this formulation is that all autocorrelation coefficients only appear on the left side in a single matrix.

$$\begin{pmatrix} R_0 & R_1 & \cdots & R_p \\ R_1 & R_0 & \cdots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \cdots & R_0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ a_1^{(p)} \\ \vdots \\ a_p^{(p)} \end{pmatrix} = \begin{pmatrix} d_0^{(p)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.11}$$

We will derive $a_i^{(p)}$ in a recursive way, whereby we let p increase from 1 to n. R_0 to R_n are supposed to be specified constants. The auxiliary variables $n_0^{(p-1)}$, $d_0^{(p)}$ and k_p are defined as follows:

$$n_0^{(p-1)} = R_p + \sum_{j=1}^{p-1} a_j^{(p-1)} R_{p-j} \tag{2.12}$$

$$d_0^{(p)} = R_0 + \sum_{j=1}^{p} a_j^{(p)} R_j \tag{2.13}$$

$$k_p = -n_0^{(p-1)}/d_0^{(p-1)} \tag{2.14}$$

Equation (2.11) will be solved by supposing that the solution of the (p-1)^th^-order system has already been determined:

$$\begin{pmatrix} R_0 & R_1 & \cdots & R_{p-1} \\ R_1 & R_0 & \cdots & R_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ R_{p-1} & R_{p-2} & \cdots & R_0 \end{pmatrix} \begin{pmatrix} 1 \\ a_1^{(p-1)} \\ \vdots \\ a_{p-1}^{(p-1)} \end{pmatrix} = \begin{pmatrix} d_0^{(p-1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.15}$$

To this purpose, the solved equation (2.15) is extended to: (using the definition of $n_0^{(p-1)}$and the symmetry of R)

$$\begin{pmatrix} R_0 & R_1 & \dots & R_{p-1} & R_p \\ R_1 & R_0 & \dots & R_{p-2} & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R_{p-1} & R_{p-2} & \dots & R_0 & R_1 \\ R_p & R_{p-1} & \dots & R_1 & R_0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a_1^{(p-1)} & a_{p-1}^{(p-1)} \\ \vdots & \vdots \\ a_{p-1}^{(p-1)} & a_1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} d_0^{(p-1)} & n_0^{(p-1)} \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ n_0^{(p-1)} & d_0^{(p-1)} \end{pmatrix} \tag{2.16}$$

Multiplication the left and right side matrices by $\begin{pmatrix} 1 \\ k_p \end{pmatrix}$ gives:

$$\begin{pmatrix} R_0 & R_1 & \dots & R_{p-1} & R_p \\ R_1 & R_0 & \dots & R_{p-2} & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R_{p-1} & R_{p-2} & \dots & R_0 & R_1 \\ R_p & R_{p-1} & \dots & R_1 & R_0 \end{pmatrix} . \begin{pmatrix} 1 \\ a_1^{(p-1)}+k_p a_{p-1}^{(p-1)} \\ \vdots \\ a_{p-1}^{(p-1)}+k_p a_1^{(p-1)} \\ k_p \end{pmatrix} = \begin{pmatrix} d_0^{(p-1)}+k_p n_0^{(p-1)} \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{2.17}$$

This equation is similar to equation (2.11). This relation indicates how the coefficients have to be updated:

$$\left.\begin{aligned} n_0^{(p-1)} &= \sum_{j=1}^{p-1} a_j^{(p)} R_{p-j} + R_p \\ k_p &= - n_0^{(p-1)}/d_0^{(p-1)} \\ \begin{pmatrix} a_1^{(p)} \\ \vdots \\ a_{p-1}^{(p)} \\ a_p^{(p)} \end{pmatrix} &= \begin{pmatrix} a_1^{(p-1)} + k_p\ a_{p-1}^{(p-1)} \\ \vdots \\ a_p^{(p-1)} + k_p\ a_1^{(p-1)} \\ k_p \end{pmatrix} \\ d_0^{(p)} &= d_0^{(p-1)} + k_p\ n_0^{(p-1)} \end{aligned}\right\} \text{ for } p = 1 \dots n \tag{2.18}$$

What remains is the initialization of all variables, which follows immediately from (2.11) for p=0:

$$d_0^{(0)} = R_0$$

The total number of multiplications and additions is $\sum_{p=1}^{n}(2p-1) = n^2$.
The number of divisions is n.

A side-product of these iterations is the set of reflection coefficients k_p, that can also be used as frame coefficients. After the Schur recursion is derived we will see that the use of the reflection coefficients is especially advantageous when the processing must be done by a fixed point DSP.

2.5 The Schur recursion

The Schur-algorithm can be derived using the same notation. The auxiliary variables $n_i^{(p)}$, $d_i^{(p)}$ and k_p are defined as follows.

$$n_i^{(p)} = R_{p+i+1} + \sum_{j=1}^{p} a_j^{(p)} R_{p+i+1-j} \tag{2.19}$$

$$d_i^{(p)} = R_0 + \sum_{j=1}^{p} a_j^{(p)} R_{i+j} \tag{2.20}$$

$$k_p = - n_0^{(p-1)}/d_0^{(p-1)} \tag{2.21}$$

In the following explanation will be shown, how these variables can be used in a recursive updating algorithm without using $\{a_i \mid i=1..p\}$. The first step is to extend equation (2.11) to include the additional variables $n_i^{(p)}$ and $d_i^{(p)}$. By using matrix notation the relation with the Levinson recursion is made clear.

$$\begin{pmatrix} R_0 & R_1 & \cdots & R_p \\ R_1 & R_0 & \cdots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \cdots & R_0 \\ \hline R_{p+1} & R_p & \cdots & R_1 \\ \vdots & \vdots & & \vdots \\ R_n & R_{n-1} & \cdots & R_{n-p} \end{pmatrix} \begin{pmatrix} 1 & a_p^{(p)} \\ a_1^{(p)} & \vdots \\ \vdots & a_1^{(p)} \\ a_p^{(p)} & 1 \end{pmatrix} = \begin{pmatrix} d_0^{(p)} & 0 \\ 0 & \vdots \\ \vdots & 0 \\ 0 & d_0^{(p)} \\ \hline n_0^{(p)} & d_1^{(p)} \\ \vdots & \vdots \\ n_{n-p-1}^{(p)} & d_{n-p}^{(p)} \end{pmatrix} \tag{2.22}$$

Again, assume first that the solution of the (p-1)th order system has already been determined:

$$\left(\begin{array}{cccc} R_0 & R_1 & & R_{p-1} \\ R_1 & R_0 & \dots & R_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ R_{p-1} & R_{p-2} & \dots & R_0 \\ \hline R_p & R_{p-1} & \dots & R_1 \\ \vdots & \vdots & & \vdots \\ R_n & R_{n-1} & \dots & R_{n-p-1} \end{array}\right) \begin{pmatrix} 1 & a^{(p-1)}_{p-1} \\ a^{(p-1)}_1 & \vdots \\ \vdots & a^{(p-1)}_1 \\ a^{(p-1)}_{p-1} & 1 \end{pmatrix} = \left(\begin{array}{cc} d^{(p-1)}_0 & 0 \\ 0 & \vdots \\ \vdots & 0 \\ 0 & d^{(p-1)}_0 \\ \hline n^{(p-1)}_0 & d^{(p-1)}_1 \\ \vdots & \vdots \\ n^{(p-1)}_{n-p-2} & d^{(p-1)}_{n-p-1} \end{array}\right) \tag{2.23}$$

Equation (2.23) can be rearranged in the same form as equation (2.22). A column is added to the R-matrix and the second column of the two other matrices are shifted down. This is allowed because the R-matrix is Toeplitz.

$$\left(\begin{array}{cccc} R_0 & R_1 & \dots & R_p \\ R_1 & R_0 & \dots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \dots & R_0 \\ \hline R_{p+1} & R_p & \dots & R_1 \\ \vdots & \vdots & & \vdots \\ R_n & R_{n-1} & \dots & R_{n-p} \end{array}\right) \begin{pmatrix} 1 & 0 \\ a^{(p-1)}_1 & a^{(p-1)}_{p-1} \\ \vdots & \vdots \\ a^{(p-1)}_{p-1} & a^{(p-1)}_1 \\ 0 & 1 \end{pmatrix} = \left(\begin{array}{cc} d^{(p-1)}_0 & n^{(p-1)}_0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ n^{(p-1)}_0 & d^{(p-1)}_0 \\ \hline n^{(p-1)}_1 & d^{(p-1)}_1 \\ \vdots & \vdots \\ n^{(p-1)}_{n-p} & d^{(p-1)}_{n-p} \end{array}\right) \tag{2.24}$$

After post-multiplying both sides by $\begin{pmatrix} 1 & k_p \\ k_p & 1 \end{pmatrix}$, the form becomes identical to equation (2.22). So the solution of the (p-1)th order equation can be updated to the pth order solution by:

$$k_p = - n^{(p-1)}_0 / d^{(p-1)}_0 \tag{2.25}$$

$$\begin{pmatrix} 0 & d^{(p)}_0 \\ n^{(p)}_0 & d^{(p)}_1 \\ \vdots & \vdots \\ n^{(p)}_{n-p-2} & d^{(p)}_{n-p-1} \\ n^{(p)}_{n-p-1} & d^{(p)}_{n-p} \end{pmatrix} = \begin{pmatrix} n^{(p-1)}_0 & d^{(p-1)}_0 \\ n^{(p-1)}_1 & d^{(p-1)}_1 \\ \vdots & \vdots \\ n^{(p-1)}_{n-p-1} & d^{(p-1)}_{n-p-1} \\ n^{(p-1)}_{n-p} & d^{(p-1)}_{n-p} \end{pmatrix} \begin{pmatrix} 1 & k_p \\ k_p & 1 \end{pmatrix} \tag{2.26}$$

The coefficient $d^{(p)}_{n-p}$ does not need to be calculated because its value is not used any more in the algorithm. The initialization of the algorithm can be derived directly from (2.24) at p=1:

$$\begin{pmatrix} n^{(0)}_0 & d^{(0)}_0 \\ \vdots & \vdots \\ n^{(0)}_{n-1} & d^{(0)}_{n-1} \end{pmatrix} = \begin{pmatrix} R_1 & R_0 \\ \vdots & \vdots \\ R_n & R_{n-1} \end{pmatrix} \tag{2.27}$$

The updating costs n divisions and $\sum_{p=1}^{n-1} 2(n-p) = n^2 - n$ multiplications and additions. The difference between the Schur and the Levinson algorithm is that the LPC-coefficients a_i are no longer calculated. These can still be derived from the reflection coefficients k_p by repeating the update of $a^{(p)}_i$ from formula (2.18), but this costs an additional $\sum_{p=1}^{n} (p-1) = n\cdot(n-1)/2$ multiplications and additions.

At this point in the analysis we have formulated three alternative representations of a speech frame in terms of various coefficients, that in principle could all be used for speech recognition:

- Autocorrelation coefficients $R_0..R_n$. In order to make them independent from the signal energy, we could divide them all by R_0.
- LPC-coefficients $a_1..a_n$. These are best calculated by the Levinson algorithm.
- Reflection coefficients. These are best calculated by the Schur algorithm.

The use of reflection coefficients, compared to the other alternatives, has several advantages, such as:

- They allow for an easy switch between orders, without recalculation of lower order coefficients.
- All variables used in the Schur algorithm can be proven to be limited in value, which is a considerable advantage when a fixed point DSP is used. If the prediction error filter has all zeros within the unit circle, we can prove that:
  $|n^{(p)}_i| < R_0$; $|d^{(p)}_i| < R_0$; $|k_p| < 1$;
  If at the start of the Schur algorithm, all autocorrelation coefficients are normalized in such a way that $R_0=1$, the maximum range of the processor is used with (in theory) no possibility of overflow.

Equation (2.22) shows a very useful relation between the 'Levinson' variables $a_i^{(p)}$ and the 'Schur' variables $n_i^{(p)}$ and $d_i^{(p)}$. The nth order autocorrelation matrix can be seen as a transformation between the Levinson and the Schur domain. The multiplications by the vector $\begin{pmatrix}1\\k_p\end{pmatrix}$ (Levinson) and the matrix $\begin{pmatrix}1 & k_p\\k_p & 1\end{pmatrix}$ (Schur) serve the same purpose in the two algorithms.

2.6 Split recursion algorithms

The Levinson and Schur algorithms are using different sets of variables that have a strong relationship. If the a_i-coefficients were used for calculating $n_0^{(p)}$and $d_0^{(p)}$, the outcome was the Levinson algorithm. If $n_i^{(p)}$ and $d_i^{(p)}$ were used, the outcome was the Schur-algorithm. Recently a method has been developed to reduce the number of multiplications even further. It results in the so-called split algorithms. As before, a split-Levinson or a split-Schur algorithm can be derived. The term 'split' originates from the fact that the variables are split in a symmetric and an anti symmetric part. Only one of these is used in a three-term recursion, instead of using both in a two-term recursion. The advantage is that the number of multiplications can be reduced in this way. One possible form of the split Schur algorithm will be derived below, from the Levinson and Schur algorithms.

First two new sets of variables are defined, and then the new update-formulas will be derived accordingly:

$$b_i^{(p)} = a_i^{(p-1)} + a_{p-i}^{(p-1)} \qquad i = 1 \,..\, p-1 \qquad (2.28)$$

$$q_i^{(p)} = n_i^{(p-1)} + d_i^{(p-1)} \qquad i = 0 \,..\, n-p \qquad (2.29)$$

One should note that $b_i^{(p)} = b_{p-i}^{(p)}$, so in fact we only have to store half of the b-variables. Similarly, instead of $n_i^{(p)}$ and $d_i^{(p)}$ we only have a single vector $q_i^{(p)}$. Reconsidering equation (2.24), we see that multiplying both sides by $\begin{pmatrix}1\\1\end{pmatrix}$ results in a new equation containing the new variables $b_i^{(p)}$ and $q_i^{(p)}$:

$$\begin{pmatrix} R_0 & R_1 & \dots & R_p \\ R_1 & R_0 & \dots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \dots & R_0 \\ \hline R_{p+1} & R_p & \dots & R_1 \\ \vdots & \vdots & & \vdots \\ R_n & R_{n-1} & \dots & R_{n-p} \end{pmatrix} \begin{pmatrix} 1 \\ b_1^{(p)} \\ \vdots \\ b_{p-1}^{(p)} \\ 1 \end{pmatrix} = \begin{pmatrix} q_0^{(p)} \\ 0 \\ \vdots \\ 0 \\ q_0^{(p)} \\ \hline q_1^{(p)} \\ \vdots \\ q_{n-p}^{(p)} \end{pmatrix} \tag{2.30}$$

The same can be applied to the two previous order solutions, combined in one matrix equation:

$$\begin{pmatrix} R_0 & R_1 & \dots & R_p \\ R_1 & R_0 & \dots & R_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_p & R_{p-1} & \dots & R_0 \\ \hline R_{p+1} & R_p & \dots & R_1 \\ \vdots & \vdots & & \vdots \\ R_n & R_{n-1} & \dots & R_{n-p} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ b_1^{(p-1)} & 1 & 1 \\ \vdots & b_1^{(p-1)} & b_1^{(p-2)} \\ b_{p-2}^{(p-1)} & \vdots & b_{p-3}^{(p-2)} \\ 1 & b_{p-2}^{(p-1)} & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} q_0^{(p-1)} & q_1^{(p-1)} & q_1^{(p-2)} \\ 0 & q_0^{(p-1)} & q_0^{(p-2)} \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ q_0^{(p-1)} & 0 & q_0^{(p-2)} \\ q_1^{(p-1)} & q_0^{(p-1)} & q_1^{(p-2)} \\ \hline q_2^{(p-1)} & q_1^{(p-1)} & q_2^{(p-2)} \\ \vdots & \vdots & \vdots \\ q_{n-p+1}^{(p-1)} & q_{n-p}^{(p-1)} & q_{n-p+1}^{(p-2)} \end{pmatrix} \tag{2.31}$$

Now it is not so difficult to see how to combine these three columns in order to get a new equation similar to equation (2.30). First a new auxiliary variable β_p is defined:

$$\beta_p = q_0^{(p-1)}/q_0^{(p-2)}; \tag{2.32}$$

Exactly the same form as (2.30) will be attained if we multiply the left and right side of (2.31) by the vector $\begin{pmatrix} 1 \\ 1 \\ -\beta_p \end{pmatrix}$.

So $q_i^{(p)}$ can be updated by (Schur version):

$$q_i^{(p)} = q_i^{(p-1)} + q_{i+1}^{(p-1)} - \beta_p \, q_{i+1}^{(p-2)} \qquad \{i=0..n-p\} \tag{2.33}$$

Alternatively (Levinson version) the coefficients $b_i^{(p)}$ could be updated with:

$$b_1^{(p)} = b_{p-1}^{(p)} = b_1^{(p-1)} + 1 - \beta_p \tag{2.34}$$

$$b_i^{(p)} = b_{p-i}^{(p)} = b_i^{(p-1)} + b_{i-1}^{(p-1)} - \beta_p \cdot b_{i-1}^{(p-2)} \tag{2.35}$$

while $q_0^{(p)}$ can be calculated by (compare with (2.30)):

$$q_0^{(p)} = R_0 + \sum_{i=1}^{p-1} b_i^{(p)} \cdot R_i + R_p \tag{2.36}$$

Only the initialization still has to be considered. Equation (2.31) for p=2 shows:

$$\left(\begin{array}{ccc} R_0 & R_1 & R_2 \\ R_1 & R_0 & R_1 \\ R_2 & R_1 & R_0 \\ \hline R_3 & R_2 & R_1 \\ \vdots & \vdots & \vdots \\ R_n & R_{n-1} & R_{n-2} \end{array}\right) \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 2 \\ 0 & 1 & 0 \end{pmatrix} = \left(\begin{array}{ccc} q_0^{(1)} & q_1^{(1)} & q_1^{(0)} \\ q_0^{(1)} & q_0^{(1)} & 2q_0^{(0)} \\ q_1^{(1)} & q_0^{(1)} & q_1^{(0)} \\ \hline q_2^{(1)} & q_1^{(1)} & q_2^{(0)} \\ \vdots & \vdots & \vdots \\ q_{n-1}^{(1)} & q_{n-2}^{(1)} & q_{n-1}^{(0)} \end{array}\right) \tag{2.37}$$

So the initialization becomes:

$q_0^{(0)} := R_0;$

$q_i^{(0)} := 2 \cdot R_i; \qquad i = 1 \ldots n-1;$

$q_i^{(1)} := R_i + R_{i+1}; \quad i = 0 \ldots n-1;$

Now β_{p+1} and $q_i^{(p)}$ can be calculated for all p and i, but if we are only interested in the reflection coefficients k_p these still have to be calculated. It is possible to express the reflection coefficients k_p in the 'split reflection coefficients' β_p.

note that:

$$\begin{aligned} d_0^{(p-1)} &= d_0^{(p-2)} + k_{p-1} \cdot n_0^{(p-2)} = d_0^{(p-2)} - k_{p-1}^2 \cdot d_0^{(p-2)} \\ &= (1 - k_{p-1}^2) \cdot d_0^{(p-2)} \end{aligned} \tag{2.38}$$

thus:

$$\begin{aligned} \beta_{p+1} &= \frac{q_0^{(p)}}{q_0^{(p-1)}} = \frac{n_0^{(p-1)} + d_0^{(p-1)}}{n_0^{(p-2)} + d_0^{(p-2)}} = \frac{(1-k_p) \cdot d_0^{(p-1)}}{(1-k_{p-1}) \cdot d_0^{(p-2)}} \\ &= \frac{(1-k_p)}{(1-k_{p-1})} \cdot (1 - k_{p-1}^2) = (1-k_p) \cdot (1+k_{p-1}) \end{aligned} \tag{2.39}$$

$$\Rightarrow \ k_p = 1 - \frac{\beta_{p+1}}{(1+k_{p-1})} \tag{2.40}$$

Because k_p, $n_i^{(p)}$ and $d_i^{(p)}$ are bounded in value, we can show that β_p and $q_i^{(p)}$ must be bounded in value too:

$$0 < \beta_p < 4$$

$$|q_i^{(p)}| = |n_i^{(p-1)} + d_i^{(p-1)}| < 2 \cdot R_0$$

The final number of additions, multiplications and divisions are:

Split Schur:	n^2 additions
	$\sum_{p=2}^{n} n-p+1 = n(n-1)/2$ multiplications
	n-1 divisions
calculation of k_p:	2n additions
	n divisions

Indeed, as compared with the Levinson and Schur algorithms the number of multiplications is halved while the number of additions remains the same. If, in addition, the reflection coefficients k_p are needed, this costs an extra 2n additions and n divisions. If n is small or if a system is used in which divisions are very time consuming, the Schur algorithm may be faster.

It may be a better alternative to directly use the β_p-coefficients $\{p=2..n\}$ as frame parameters, together with the first reflection coefficient k_1. The purpose of considering these alternative parameter sets is to determine the best set for a small speech recognition system using the DSP (the TMS320C25). Only after the recognition system is completed a valid comparison can be made, to decide which set will result in the highest score within an acceptable calculation time.

Again relation (2.30) shows the duality between the 'Split Levinson' variables $b_i^{(p)}$ and the 'Split Schur' variables $q_i^{(p)}$. The nth-order autocorrelation matrix can be seen as a transformation between the 'Split Levinson' and the 'Split Schur' domain.

2.7 Implementation of the Schur algorithm

Before describing any implementations let us consider the relative merits of a fixed-point DSP. Some of the remarks only apply for the TMS320C25, others are of a more general character.

Signal processing algorithms usually require many additions and multiplications. These are optimized to use only a single instruction cycle, by an on-chip hardware multiplier. Thus, an addition is not cheaper than a multiplication, as is the case in normal microprocessors.

Because DSP's do not contain a hardware division unit, divisions are much less efficient than multiplications. On the TMS320C25 a division costs at least 16 cycles. This makes the "Split" algorithms less attractive if the reflection coefficients need to be calculated, as an extra division is required for each order.

Because the TMS320C25 is fixed point, all variables must be scaled to use the full range of the processor. For the "Schur" and "Split Schur" variables, the maximum range is known, which can be successfully applied in the implementation of the Schur algorithms.

Because of the above arguments, the Schur algorithm was chosen for implementation on the TMS320C25. A library of various functions has been developed. The A/D-converter is read at a sample rate of 10 kHz. Each 12 ms a new window of 240 samples (50% overlap) is taken. These windows are multiplied by a Hamming window. Then, 11 autocorrelation coefficients are calculated. These are used in the Schur algorithm as described in Section 2.5. All parameters mentioned, such as the sampling rate, can be easily changed in the program. Only, the maximum window length is 256 samples, because of the limited on-chip memory.

With the above parameters, calculation of the reflection coefficients requires about 10000 cycles (= 1 ms) for each frame. The majority of them (about 3000 cycles = 0.3 ms) is used for the calculation of the autocorrelation coefficients. The processor is only used for 8% of the time, which means that real-time speech analysis is possible on a DSP without any problem. Even more tasks could be easily assigned to the same processor, if desired.

2.8 Other parameter sets

Several other parameter sets are possible, however, at an extra expense of time. An important set consists of the log area ratio parameters. These are defined as:

$$g_i = \text{bllog}(k_i) = \log\left(\frac{1 + k_i}{1 - k_i}\right)$$

where the coefficients k_i are the reflection coefficients. The inverse transformation is:

$$k_i = \text{blexp}(g_i) = \frac{1 + \exp(g_i)}{1 - \exp(g_i)}$$

These coefficients have been shown by several authors to be very useful in quantization, because of its relatively flat spectral sensitivity (Gray and Markel [18]). Calculation of the log function is very time consuming, but considering the total transformation from k_i to g_i, a good approximation can be made. If the function is divided in 9 linear segments, as is shown in Fig. 2.4 and Table 2.1, a very simple approximation function results. It can be computed in only 3.3 µs on the TMS320C25. Because the slope in each segment is a power of 2, the function can be implemented using additions, conditional jumps and shifts only. Further specification of the realization of this function can be found in Chapter 7.

$k_i = \text{blexp}(g_i)$		$g_i = \text{bllog}(k_i)$		slope
-1		-6	($-\infty$)	64
-0.96875	(-31/32)	-4	(-4.1431)	16
-0.90625	(-29/32)	-3	(-3.0123)	8
-0.8125	(-13/16)	-2.25	(-2.2687)	4
-0.5	(-1/2)	-1	(-1.0986)	2
0.5	(1/2)	1	(1.0986)	4
0.8125	(13/16)	2.25	(2.2687)	8
0.90625	(29/32)	3	(3.0123)	16
0.96875	(31/32)	4	(4.1431)	64
1		6	(∞)	

Table 2.1: Piecewise linear approximation of the bllog() function

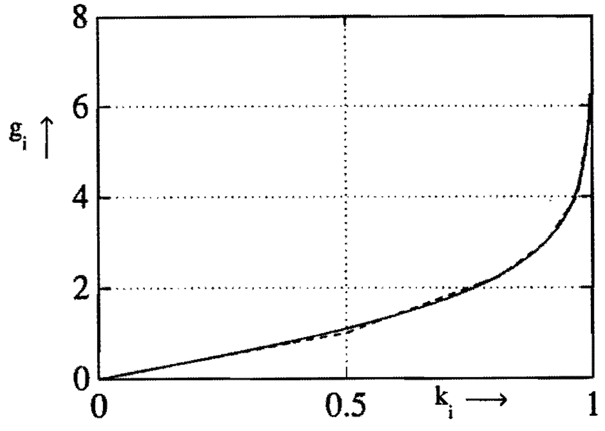


Fig. 2.4: Piecewise linear approximation of the bllog() function

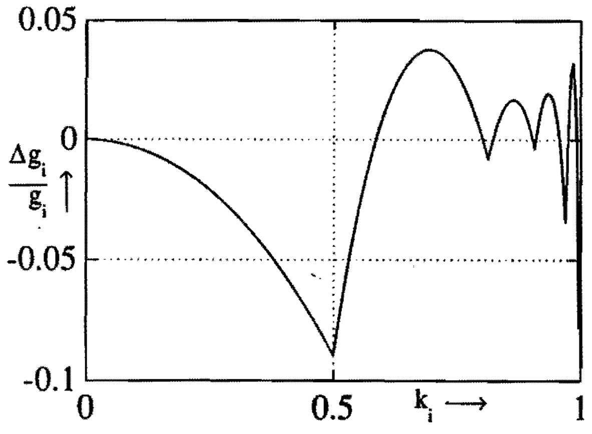


Fig. 2.5: Relative error on bllog() approximation

Fig. 2.5 shows that the maximum error is 9.0 %, at k_i=0.5. At k_i=0.9965 the error again reaches 9.0 % and increases continuously while k_i is further approaching 1. In practice this will be no problem because, if the reflection coefficients are approaching ±1, the zeros of the prediction error filter approach the unit circle. This indicates that the synthesis filter (the inverse of the prediction error filter) is approaching instability. Therefore it is better to restrict the range of the log area ratio parameters to such an extent that small errors in the reflection coefficients have not much influence on the calculated log area ratio parameters.

Other parameter sets, like the Line Spectral Pair (LSP) and the Cepstrum parameters, are not discussed, because they require significantly more calculations than the sets mentioned in this chapter. The LSP-parameters are closely related to the zeros of the predictor polynomial. These zeros only can be calculated using an iterative procedure because no direct method is known for polynomial orders higher than 3. The Cepstrum parameters involve an additional recursion starting with the LPC parameters (compare Rabiner and Schafer [52] page 442), but they have a nice interpretation.

In speech not only the frame parameters can contain information about the spoken words. Much information is usually contained in the transitions between stationary regions. Several speech recognition systems try to use this fact by adding first (or even higher) derivatives of other parameters to the frame vector. The general approach is called a Linear Predictive HMM, which is discussed in Kenny et al. [24]. Because the aim of this thesis does not include carrying out an exhaustive search for possible parameter sets, this possibility is not further considered.

Even though very good results applying the Cepstrum parameters have been reported, other, less expensive, parameter sets will be used. Currently, the set most promising for application on a fixed point DSP like the TMS320C25 seems to be the log area ratio parameter set. However, the ideal set will probably be a combination of various type parameters, which still has to be determined.

3. SOME COMPONENTS OF A SPEECH RECOGNITION SYSTEM

General early day speech recognition systems consist of four parts: analysis, vector quantization, segmentation and dynamic time warping (DTW). The analysis has already been treated in the previous chapter. In this chapter the global structure of a general speech recognition system is analyzed. Special attention is paid to the use of vector quantization and segmentation.

3.1 Outline of a speech recognition system

Speech recognition takes several steps. The speech signal has to be transformed into a representation that is suitable for further digital processing. A library has to be created that contains a description of all words in the recognition system. This library has to be trained by the speakers that will later use the system. A matching algorithm will compare the incoming speech with the contents of the library.

The earliest speech recognition systems had a very simple structure. They employed vector quantization and isolated word recognition. In addition, the systems had a small library and were trained by only one person. These restrictions were necessary to make speech recognition at all possible.

Presenting a speech signal to a small scale digital computer poses several problems. A sample frequency of at least 6.67 kHz is necessary to support telephone quality speech. A processor, handling this sampling rate, has hardly time left for additional processing. This problem can be solved by special hardware, that transforms the speech signal in a limited number of slowly varying parameters, as described in Chapter 2.

Vector quantization

LPC-analysis yields for every frame a set of coefficients, often called the feature vector. If the feature vector contains N coefficients, any vector can be assigned a single point in a N-dimensional space. A large collection of these feature vectors yields a nonuniform density in this N-dimensional space, which is shown in Fig. 3.1 (page 33). A vector quantizer can be used to exploit this fact. The parameter space is divided in a fixed number of segments, each of them is assigned a label. In regions containing many vectors the segments are smaller. Each segment has its own centre vector, which is used as a representation for all vectors within the segment.

Isolated words

In continuous speech, words are often connected and it is impossible to locate the word boundaries from the speech signal alone. To avoid this problem, one could insert a silent period between words. Now each word can be compared with previously stored words. Such systems will be called "isolated", while unrestricted systems will be called "connected". In literature occasionally the terms "discrete utterance" versus "continuous" are used, but this might cause confusion because HMM's also have a "continuous" and a "discrete" version. The Dynamic Time Warping (DTW) method, also known as Template Matching (TM), is basically an isolated recognizer. It can, however, be adapted to continuous speech.

Limited Vocabulary

Another choice in developing speech recognizers has to do with library size. A larger library will result in a larger overlap between words because many words share common syllables. The recognition system can make use of this fact by selecting a shorter recognition unit such as a syllable, di-syllable, demi-syllable, phoneme, di-phone, tri-phone and so on. The smaller the unit the smaller the accuracy. Loss of accuracy can be compensated for by higher levels of processing. Note that the recognizer is expected to recognize the spoken words and not the units.

Single person

The library is highly simplified if only a single person needs to be recognized. The training process is simplified, because only one or several examples of each word are needed. Because different people can have very different pronunciations of the same words, all these possibilities need to be present in the library.

A simple recognizer thus performs:

- Analysis
- Vector quantization
- Segmentation
- Dynamic Time Warping

In addition a training algorithm is needed to make up the library.

Vector quantization and Segmentation will be discussed in the next sections, while Dynamic Time Warping is explained in greater detail in Chapter 4.

3.2 Vector quantization.

After the analysis, vectors of parameters are available for each time frame. These can be quantized into a fixed set of feature vectors $\bar{\mu}_m$ (m=1..M) by a vector quantizer. The feature vector closest to the incoming parameter vector is selected as a representation of the original vector. The representation set $\bar{\mu}_m$ (m=1..M) has to be computed beforehand.

If the number of segments is chosen large enough, the quantization error can be made arbitrary small. The number of bits needed to describe the frame vector is reduced to ${}^2\log(M)$, where M is the number of segments. Because the English language contains about 40 phonemes, that must be assigned a minimum of one label each, M must be at least 40. In practical systems the vector quantizer has a size ranging from 64 to 1024 (6 to 10 bits/frame). This is much less than the size of any parameter vector, for which 6 coefficients of 6 bits each per frame appears to be the absolute minimum.

A "distance measure" between incoming speech vectors and representation vectors is needed. Often the squared Euclidean distance,

$$d(m,t) = (\bar{\mu}_m - \bar{y}_t)^* \cdot (\bar{\mu}_m - \bar{y}_t) \qquad (3.1)$$

is used.

If the parameters have a different range but are still supposed to be uncorrelated, this can be extended to

$$d(m,t) = \sum_{l=1}^{L} \left(\frac{\mu_{ml} - y_{tl}}{\sigma_l} \right)^2 \tag{3.2}$$

which is a special case of the Mahalanobis distance. Here σ_l is the expected variance of parameter y_{tl}. For each frame $\bar{y}_t$ the distance d(m,t) to each of the representation vectors $\bar{\mu}_m$ (m=1..M) is calculated. The index m of vector $\bar{\mu}_m$ with minimum distance is input to the remainder of the recognition algorithm.

An algorithm to calculate a suitable set of representation vectors $\bar{\mu}_m$ is the k-means clustering algorithm. First a large set of candidate vectors are created using sample vectors from various input samples. Two vectors from the set which have the maximum distance are taken as the initial centres of a two-element vector quantizer. Using these centres the frame vectors are divided into two groups. Then the two centres are replaced by the respective centres-of-gravity. Another re-assignment of each vector is made and again the new centre of each group is calculated. This process is repeated until it converges and no further changes take place.

After convergence of the process, each group is split in two by again selecting two candidate vectors with maximum distance. The process is iterated until the required number of representation vectors is obtained. Note that M will be a power of 2. Fine tuning by hand is possible, removing templates containing not enough vectors or splitting templates with too many vectors. After each change this iteration must be started again until no further changes take place in the vector library.

Implementation

The vector quantization and k-means clustering algorithms are implemented using a TMS320C25 emulator (SWDS, produced by Texas Instruments), a conversion program written in TURBO PASCAL, and MATLAB (a mathematical laboratory toolbox). First, the log area ratio parameters of the dutch sentence "De bal vloog over de schutting" are calculated in real time using an assembler program on the TMS320C25. The speech signal is sampled with a 10 kHz sampling rate. Each 12 ms a group of 240 samples (50% overlap) is multiplied by a Hamming window. For each frame 10 reflection coefficients are calculated and converted to the log area ratio parameters. The result is stored in memory. The SWDS-emulator, which is used for debugging the program, is able to store the parameters to disk. This disk file is converted to MATLAB format by a specially written conversion program. In the end, all 200 vectors are available as a 200x10 matrix in MATLAB, ready for further processing.

The k-means clustering algorithm is implemented in MATLAB. Because MATLAB is a general toolbox - which is not optimized for storage efficiency - the algorithm was very slow. The clustering lasted several minutes, depending on the number of vectors used.

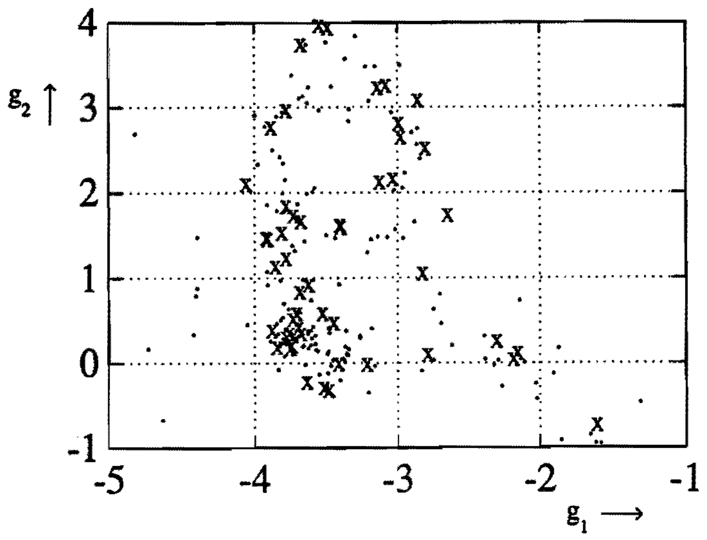


Fig. 3.1: result of the k-means clustering algorithm

3.3 Isolated word segmentation

In order to be able to segment the speech, a unit has to be defined. One possibility is to choose linguistic units like words, syllables, demi-syllables, phonemes etc. This kind of segments has the disadvantage that the segment borders can not be located using the speech signal only. Additional assumptions are needed. Isolated words separated by silent periods [22] are often used. One could also use diphones as segmentation units, as is done by van Hemert [19].

The simplest algorithms assume that speech consists of isolated words, separated by periods of silence. A speech recognition system based on this assumption requires a very cooperative speaker which distinctly pronounces each separate word. The input signal is squared (calculating the power), low pass filtered and compared with some threshold. If the power falls below the threshold a silent period is declared. The problems with such an approach are:

- sensitivity to background noise
- glottal stops might be confused with silent periods
- short sounds (like coughs) are considered as words

A way to overcome these possible problems is to use a Finite State machine (FSM) as in [22]. This FSM has 4 states (Fig. 3.4), each representing a distinguished speech state as:

- Silence
- Increasing level
- Decreasing level
- Word

The energy level E is compared to 3 fixed levels:

$E_{threshold}$: Maximum level for transition to state **Silence**
E_{noise}: Minimum level for transition from state **Silence**
E_{high}: Minimum level required for state **Word**

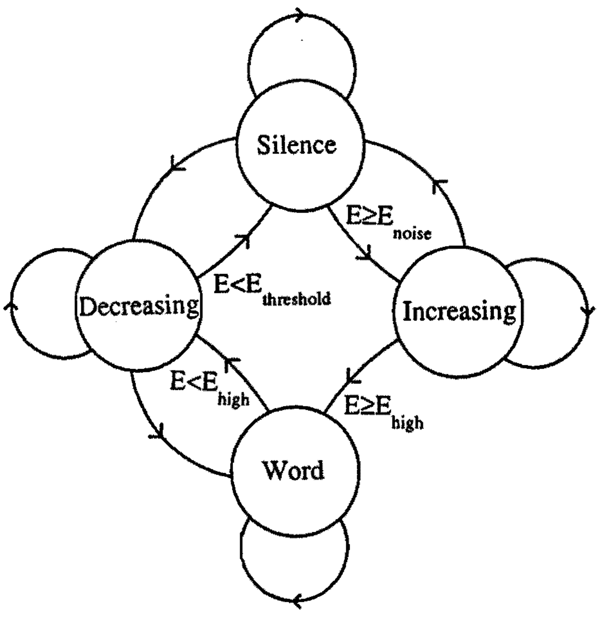


Fig. 3.4: Finite State Machine for word detection

Initially the FSM is in the **Silence** state. As long as the level is below E_{noise}, the FSM remains here. As soon as $E \geq E_{noise}$ a transition is made to the **Increasing** state. The moment of transition is recorded.

The FSM remains in the **Increasing** state, as long as $E_{noise} \leq E < E_{high}$. If the energy level is lower, then the segments up to this point are considered belonging to a silent period, and the process starts again. If the energy level becomes higher than E_{high}, a transition is made to state **Word**. The moment of transition is marked as the beginning of a new word.

If the energy level decreases below E_{high} the **Decreasing** state is entered, indicating the possible ending of the word. Further decrease below $E_{threshold}$, results in a transition back to the **Silence** state. A final decision as to a silent period is made after a certain elapse of time, for otherwise a glottal stop could be confused with a silent period.

This mechanism has been implemented on a small TMS32010 system, as described in [22]. It works well, but it is only useful for isolated word recognition. When connected words belonging to a large vocabulary are to be recognized, a more sophisticated segmentation in smaller units is necessary.

3.4 Segmentation of speech in phoneme-like units

Next we consider segmentation in phoneme-like segments. Speech consists of stationary segments and transitions. During stationary segments the parameters are varying slowly. They can be calculated very accurately, and succeeding frames have a lot of correlation. On the other hand, during transitions the parameters are rapidly varying. The correlation between frames is low, but parameter values will not be accurate. We will describe an algorithm that is able to use these changes in parameter variations for determining local boundaries in the speech signal. This algorithm is implemented and evaluated.

We start out by defining a correlation measure between frames. This measure will have a value of 1 if the frames are identical, and it will approach 0 as frames become more distinct. A useful function is:

$$C_{ij} = \exp\left(-\frac{1}{2} \sum_{l=1}^{L} \left(\frac{y_{jl} - y_{il}}{\sigma_l} \right)^2 \right) \tag{3.3}$$

where i and j are frame indices and y_{il} (l=1..L) are the log area ratio coefficients (or any other parameter set).

The summation term in (3.3) is again the Mahalanobis distance, where σ_l is a weight factor (the variance of the parameter). This can be explained as follows. Suppose that the parameters of frame j are formed by adding Gaussian noise with variance σ_l^2 to the parameters y_{il}. Then the probability density of y_{jl} and $\bar{y}_j$ is:

$$P(y_{jl}) = \frac{1}{\sigma_l \cdot \sqrt{2\pi}} \exp\left(-\frac{1}{2} \left(\frac{y_{jl} - y_{il}}{\sigma_l} \right)^2 \right) \tag{3.4}$$

$$P(\bar{y}_j) = \prod_{l=1}^{L} P(y_{jl}) = \prod_{l=1}^{L} \frac{1}{\sigma_l \cdot \sqrt{2\pi}} \cdot \exp\left(-\frac{1}{2} \sum_{l=1}^{L} \left(\frac{y_{jl} - y_{il}}{\sigma_l} \right)^2 \right) \tag{3.5}$$

This is, apart from a constant scaling factor, equal to C_{ij}. The function C_{ij}, with i fixed and j varying, will have a maximum at j = i.

Let us define a 'centre of gravity' function G(i) that is positive if the centre of gravity occurs right from the maximum and that is negative otherwise:

$$G(i) = \sum_{j=i-N}^{i+N} (j-i).C_{ij} \tag{3.6}$$

If the centre of gravity is left from the maximum (G(i) < 0), then frame i has more correlation with previous frames than with future frames. So previous frames are more stable, and future frames are more unstable. Zero-crossings of the function G(i) indicate the locations of stability extremes, both maxima and minima. If G(i) has a negative zero-crossing, then frame i is the centre of a region where the stability is (locally) maximal. In a positive zero-crossing the stability is minimal, which constitutes an indication of a transition between two more stable regions.

Note that still a value N must be defined. When, for j farther from i, C_{ij} is increasing again, these frames should not be included in G(i). They are likely to belong to another region having about the same parameters.

In this way the incoming speech can be segmented. The borders of the segments are located by searching for the positive zero-crossings of G(i). The centres are the negative zero-crossings of G(i).

For each segment the vector quantizer will deliver a label, the duration and the correlation C_{sv}, for which s is the central frame of the segment and v is the chosen library vector. It is also possible to deliver more candidates with probabilities. During input the incoming frame vectors are compared with the library vectors. The resulting label(s) can be input to a DTW, HMM or RMM recognizer (as described in Chapter 4, 5 and 6).

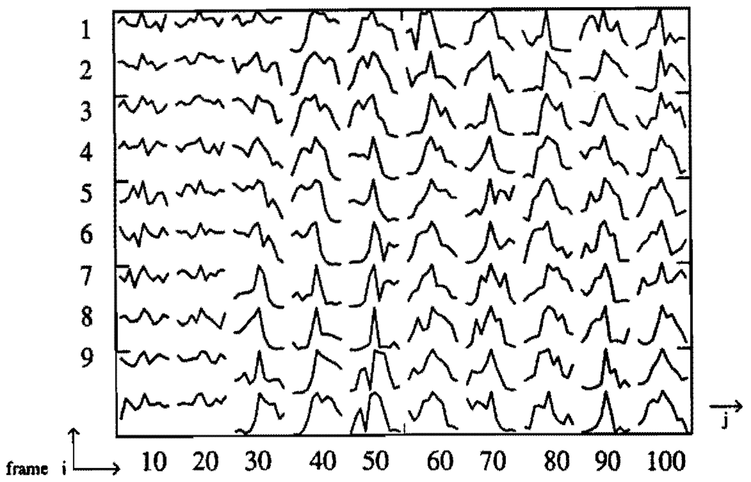


Fig. 3.5: Correlation function C_{ij} applied to the sentence:
"De bal vloog"

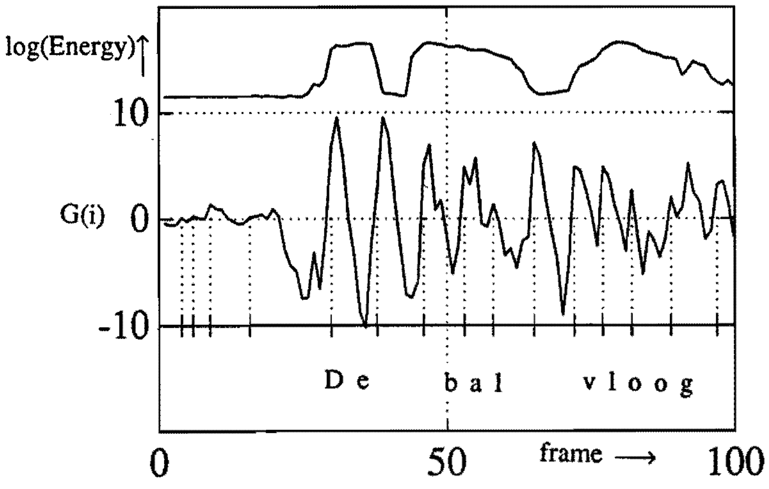


Fig. 3.6: Centre of Gravity function G(i) applied to the sentence:
"De bal vloog"

A test was performed to see how this segmentation works, and whether it could be used for reduction of the number of frames. Also, an optimal value of N had to be determined. The algorithm described above is implemented in MATLAB. A choice of N=10 produced the best results. For the same sentence as used in the vector quantizer test, the correlation function C_{ij} $(i-N<j<i+N)$ has partly been drawn in Fig. 3.5 while the centre of gravity function G(i) has been plotted in Fig. 3.6.

The centre of gravity function G(i) has very clear zero crossings. This confirms the results of van Hemert [19], who uses this algorithm for a different purpose. The borders between rather stationary segments can be estimated very accurately. However, a big disadvantage is that insertions and deletions of segment borders are very likely to occur. This can be explained as follows:

During long steady sounds (like vowels) the centre of gravity function G(i) will be around zero. Small fluctuations in the parameters will cause additional zero crossings, that will be interpreted as extra segment borders. This causes extra undesired insertions. This effect especially arises during silence periods, like in frames 1-20 in Fig. 3.6. Increasing N will reduce this effect, but increases the probability of undesired deletions.

Very fast changes, like "vl" in "vloog" (frame 70-80), could easily be missed. Of course N could be decreased, but this will increase the number of undesired insertions. A definite solution to this problem is only possible when upper level information is available. If it is known beforehand which phonemes occur in the sentence, another algorithm can be used comparing the sentence with the given phonemes [19]. This algorithm makes no selection mistakes, because the phonemes are already given, but the estimation of the segment border positions will be much less accurate. Matching of the segment borders as found by these two algorithms yields a new algorithm, combining the advantages of both previous algorithms.

In our case upper level information about the phoneme content of the sentence is only available **after** the speech recognition is complete, while the segmentation is used **before** the recognition. In Chapter 5 it will be shown that it is possible to construct a speech recognition system without pre-segmentation the incoming speech. Segmentation can only be done during the recognition process, using upper level information about the structure of the spoken sentence.

4. THE SPEECH RECOGNITION PROBLEM

For a good understanding of the current state of speech recognition, it is useful to compare various methods that are presently used. Despite their differences, Dynamic Time Warping (DTW) and the Hidden Markov Model method (HMM) have a number of features in common. This chapter will show the strong and weak points of each method, and discuss the differences.

The mathematical notation used in this chapter is different from notations used in other publications. This is necessary to develop a common framework for comparison. Besides, the notation introduced in this chapter will also be used in Chapter 5, in which a more general method of recognition is developed.

4.1 Dynamic Time Warping

The first method that was used for isolated word recognition is Dynamic Time Warping (DTW), also known as Template Matching (TM). This approach starts with the segmentation of the incoming speech into words, using silences as a separator between words. Each word has a discrete label or a set of parameters (for instance LPC-parameters) for each discrete instance of time. Such a label or set of parameters describing the momentary symbol is called a **frame**. A word library is present which contains a template of each word using the same labels or parameters.

With these assumptions the recognition problem can be re-formulated as follows. A procedure is needed by which a 'distance' can be calculated that gives an indication of the measure of correspondence between an incoming word and each word template in the library. If the incoming word and the template are of equal duration, each corresponding frame of the two words can be compared using a local distance measure. The total distance could be defined as the sum of all local distances found.

Usually, the two words will not be of equal duration because words are rarely spoken at the same rate. For various words the duration even might vary a lot. The method of Dynamic Time Warping handles this by allowing each frame t of the incoming speech (t=1..T) to be matched with another frame i of the template (i=1..N). If the library word contains frames U_i and the incoming speech supplies frames Y_t, a function d(i,t) is defined to describe the 'distance' between these.

If U_i and Y_t are discrete labels, a table containing the distance of each pair of U_i and Y_t is used. If U_i and Y_t are N-dimensional vectors (represented by $\bar{u}_i$ and $\bar{y}_t$) then we could use the euclidean distance measure:

$$d(i,t) = (\bar{u}_i - \bar{y}_t)^* \cdot (\bar{u}_i - \bar{y}_t) \qquad (4.1)$$

Other functions are possible as long as these are valid distance functions, i.e. at least satisfy the following conditions:

- $d(i,t) = 0$ if $U_i = Y_t$
- $d(i,t) > 0$ if $U_i \neq Y_t$

Suppose we have defined such a function d(i,t). If the two words differ in time scale, we have to develop the path in such a way that the cumulative sum of distances reaches its maximum, or:

$$D(i,t) = \sum_{\substack{x,y=1,1 \\ \text{along the} \\ \text{best path}}}^{i,t} d(x,y) \qquad (4.2)$$

Fig. 4.1. shows a representation of this procedure. Along the horizontal axis are the incoming speech frames, along the vertical axis are the template frames. A path needs to be found with a minimum cumulative distance from the lower left to the upper right corner.

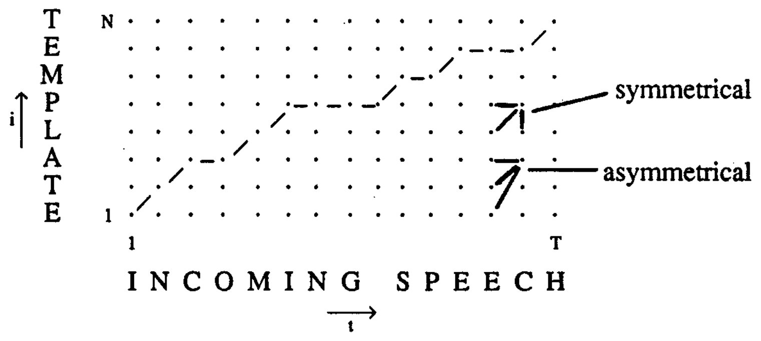


Fig. 4.1. Illustration of the Dynamic Time Warping (DTW) approach

Two versions of the algorithm exist, a **symmetric** and an **asymmetric** one. In the **symmetric** version any point (i,t) has three predecessors (i-1,t) (i-1,t-1) and (i,t-1). Exchanging the incoming speech and the template speech has no influence on the DTW-algorithm. In the **asymmetric** version, each point (i,t) also has three predecessors, but instead of (i-1,t) the point (i-2,t-1) is selected.

Let D(i,t) be the sum of all path distances up to point (i,t), where each point has only three predecessors as indicated above. Then we can express D(i,t) in a recursive relation. The symmetric version becomes:

$$D(i,t) = \min\left(\begin{array}{l} D(i\ ,t-1) + \ \ d(i,t), \\ D(i-1,t-1) + 2\cdot d(i,t), \\ D(i-1,t\ ) + \ \ d(i,t) \end{array} \right) \qquad (4.3)$$

For a diagonal step the local distance is counted double, because a diagonal step is comparable to a horizontal and a vertical step together. This assures that all possible paths have the same number of distances included in the cumulative distance.

The asymmetric version becomes:

$$D(i,t) = \min\Big(D(i,t-1),\ D(i-1,t-1),\ D(i-2,t-1) \Big) + d(i,t) \qquad (4.4)$$

The initialization in case of the asymmetric version starts with:

$$D(1,1) = d(1,1); \qquad D(i,1) = 0 \text{ for } i=2..N; \tag{4.5}$$

And in case of the symmetric version:

$$D(1,1) = d(1,1); \quad D(i,1) = D(i-1,1) + d(i,1) \text{ for } i=2..N \tag{4.6}$$

In the symmetric algorithm, the sum contains N+T local distances, one for each horizontal and vertical step. In the asymmetric algorithm only horizontal steps are counted, resulting in only T terms.

In this approach it is assumed that all possible paths from (1,1) to (N,T) are equally likely. In practice, diagonal steps are more likely to correspond with the input speech than horizontal and vertical steps. The algorithms can be extended by introducing extra penalties for each different step. From now on, only the asymmetrical algorithm will be used, but for the symmetrical algorithm a similar derivation can be developed:

Initialization: $$D(1,1) = d(1,1); \qquad D(i,1) = 0 \text{ for } i=2..N; \tag{4.5}$$

Updating: $$D(i,t) = \min\left(\begin{array}{l} D(i\ \ ,t-1) + hp, \\ D(i-1,t-1) + dp, \\ D(i-2,t-1) + vp \end{array} \right) + d(i,t) \tag{4.7}$$

hp = horizontal penalty
dp = diagonal penalty
vp = vertical penalty

A major class of dynamic time warping methods belongs to this group. In case penalties are selected, as in the last example, their values have to be determined experimental. No method is known for training these penalties. Training of words from the word library is easily done. Each word that has to be recognized by the recognition system has first to be pronounced once.

The advantage of simple training, is a disadvantage at the same time. If a word is trained by only one pronunciation, any error in pronouncing it affects the performance of the recognition system. Therefore, it is advantageous to provide multiple pronunciations. Using a larger library entails longer searches.

It is clear that the quality of the speech recognition system can be improved by adding more parameters. But these additional parameters must be set to the correct value. Improperly trained parameters might impair the recognition process. The number of parameters should be proportional to the amount of available training data.

4.2 The Hidden Markov Model

The Dynamic Time Warping method is reported to work very well in small speech recognition systems. Still, if recognition systems with a large vocabulary or speaker independent systems are required, the disadvantages of DTW become apparent. If the number of words or the number of pronunciations for each word increases, a longer training session becomes necessary. Time and storage required for recognition grow in proportion to the vocabulary. It should be possible to model a spoken word from multiple training sequences, using a kind of average of the various pronunciations. This makes the training more difficult, but the recognition will require less time and memory.

A theory has been developed, mainly thanks to Baum [10], which presents a probabilistic interpretation of the dynamic time warping method. If instead of using **distances**, the algorithms are re-formulated using **probabilities**, a different method can be derived for recognition: the Forward algorithm. First, the Hidden Markov Model (HMM) is introduced. Secondly, the relation with the dynamic time warping algorithm is explained.

Suppose we have a state machine, as in Fig. 4.2., having states $S_1..S_N$. At discrete time instance t=1 this machine performs a transition to one of the states S_i. At time t=2 a transition is made to another state S_j. This process continues until at time t=T+1 a transition is performed to node F. The machine produces symbols $Y_1..Y_T$.

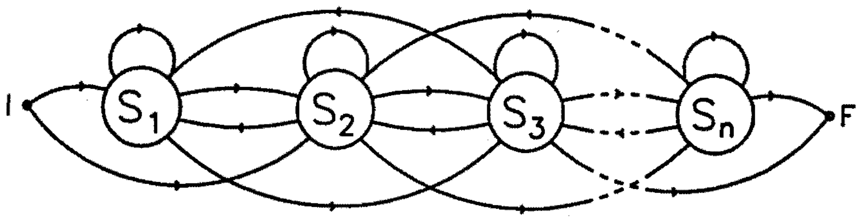


Fig. 4.2: An example of a Hidden Markov Model

Such a machine exists for each word in the library. Supposing we have a model W and a sequence $Y = Y_1..Y_T$, we will derive a method of calculating the probability of word W producing sequence Y. Repeating this for all words, the word with the highest probability is selected as the recognized word. According to Bayes' theorem we can write:

$$P(W|Y) = P(Y|W) \cdot P(W) \,/\, P(Y) \tag{4.8}$$

where: P(W|Y) = probability that W is the correct word, given sequence Y
P(Y|W) = probability that W produces sequence Y
P(W) = a priori probability for word W
P(Y) = probability that any word produces sequence Y

Here P(Y) is independent of the word, so it has no effect on the word selection. P(W) is the a priori word probability. If all words are equally likely, P(W) is a constant that as well could be removed without influencing the selection algorithm. The selection then only consists of calculating P(Y|W) for each word and selecting the word with maximum probability. If the word probabilities are not equally likely, we need to calculate P(W)·P(Y|W) in order to find the word with maximum a posteriori probability.

Associated with each transition is a parameter a_{ij}, indicating the probability that state S_j will be active at any time t+1, given that state S_i is active on time t. This transition probability is assumed to be time independent.

The training algorithm uses the same model. All parameters a_{ij} will be trained to maximize the above probability P(W|Y).

4.3 Recognition using the HMM: The Forward algorithm.

For calculation of the probabilities P(Y|W) for each word, a recursive algorithm exists, the Forward algorithm. Some a priori probabilities need to be trained first before they can be used in the recognition. For now we suppose this training has already been done. We define:

$\mathbf{a_{Ij}}$	= a priori probability of making a transition to state S_j coming from node I.
$\mathbf{a_{ij}}$	= a priori probability of making a transition to state S_j after leaving state S_i.
$\mathbf{a_{iF}}$	= a priori probability of making a transition to node F after leaving state S_i.

The symbol a_{ij} is not only used as a parameter associated with a transition from state S_i to S_j, but also as the transition itself. This allows us to write about transition a_{ij} instead of the transition from state S_i to S_j, which is much easier.

There two basic alternative formulations of the forward algorithm. The difference is, that we can assume that the output is associated with a transition or with a state. If the output is associated with the transitions, two further possibilities exist, because the symbol can be produced before or after the transition takes place:

Version 1a:

First, let us assume that all transitions a_{ij} and a_{iF} have a probability function p(i,U) describing the probability that symbol U is produced. We see that this function is only dependent on the state the transition is coming from, and is independent from j. The transitions

a_{Ij} have no output symbol. Then the initial probability that the state machine is in a particular state S_j, assumed that at time t=1 the first transition a_{Ij} has been taken, can be written as:

$$P^f_I(S_j,1) = a_{Ij} \tag{4.9}$$

where $P^f_I(S_j,t)$ is equivalent to $\alpha(j,t)$ in other publications (like Baum [10]) and is defined as:

$$P^f_I(S_j,t) = P(Y_1..Y_{t-1},S_j \text{ active at time } t)$$

Or, in other words: $P^f_I(S_j,t)$ represents the probability that the HMM is in state S_j at time t, and sequence $Y_1..Y_{t-1}$ is produced, assumed that at time t=1 the first transition occurred. State S_j can only be initiated after any state S_i has finished, thereby producing symbol Y_t and performing transition a_{ij}. The state probabilities for times t=2..T can also be calculated applying the following recursive relation:

$$P^f_I(S_j,t) = \sum_{i=1}^{N} P^f_I(S_i,t-1) \cdot p(i,Y_t) \cdot a_{ij} \tag{4.10}$$

The probability that, finally, at time t=T a transition a_{iF} is performed, thereby producing the final output symbol Y_T, is:

$$P(Y|W) = \sum_{i=1}^{N} P^f_I(S_i,T) \cdot p(i,Y_T) \cdot a_{iF} \tag{4.11}$$

This is what we need to know, as was stated in the beginning of this section. This algorithm is known as the forward algorithm, because its probabilities are calculated in a forward recursive way.

Version 1b:

An alternative formulation is possible if we assume that each output symbol connected with transition a_{ij} is only dependent on j. Also we assume that transition a_{Ij} produces an output symbol and a_{iF} does not. Then we can write:

$$P^f_F(S_j,2) = a_{Ij} \cdot p(j,Y_1) \tag{4.12}$$

where $P^f_F(S_j,t)$ is defined as:

$$P^f_F(S_j,t) = P(Y_1..Y_{t-1},S_j \text{ active at time } t-1)$$

Or, in other words: $P^f_F(S_j,t)$ represents the probability that the HMM is in state S_j at time t-1, and sequence $Y_1..Y_{t-1}$ is produced, assuming that at time t=1 the first transition occurred. The state probabilities for times t=2..T+1 can be calculated employing the following recursive relation:

$$P^f_F(S_j,t) = \left(\sum_{i=1}^{N} P^f_F(S_i,t-1) \cdot a_{ij} \right) \cdot p(j,Y_{t-1}) \tag{4.13}$$

The probability that, finally, at time t=T a transition a_{iF} occurs, is:

$$P(Y|W) = \sum_{i=1}^{N} P^f_F(S_i,T+1) \cdot a_{iF} \tag{4.14}$$

Again we have developed a recursive relation that calculates the probability that the HMM produces sequence $Y_1..Y_T$. In order to illustrate the relation between these two versions, a third interpretation is described in which each symbol is associated with a state.

Version 2:

In this case, we make a distinction between two possible definitions of the state probabilities, **before or after** a symbol is produced. Each time a symbol is produced, time increases with one unit. Therefore, any state S_i that is initiated at time t will produce a symbol Y_t according to probability function $p(i,Y_t)$ and perform a transition to the next state at time t+1.

We define:

$$P^f_I(S_i,t) = P(Y_1..Y_{t-1},S_i \text{ initiated at time } t)$$
$$P^f_F(S_i,t) = P(Y_1..Y_{t-1},S_i \text{ finished at time } t)$$

It will be no surprise that these definitions are equivalent to the two versions of the state probabilities. Also, if state S_i is active at time t it will finish the state at time t+1, which explains the use of t-1 instead of t in the previous definition of $P^f_F(S_i,t)$. Using both variables in one recursive algorithm, the forward algorithm can be written as:

The probability of each state before any symbol is emitted, $P_I^f(S_i,t)$, is already described in equation (4.9). For any other time t=2..T, the probability of initiating any state S_j consists of the probabilities of leaving each state S_i and performing transition a_{ij}.

$$P_I^f(S_j,t) = \sum_{i=1}^{N} P_F^f(S_i,t) \cdot a_{ij} \tag{4.15}$$

For any other time t=1..T-1 the probability of each state after a symbol has been emitted, can be written as:

$$P_F^f(S_i,t+1) = P_I^f(S_i,t) \cdot p(i,Y_1) \tag{4.16}$$

Finally, we already derived in equation (4.14) how to calculate P(Y|W).

Summarized, the forward algorithm can be executed as:

1) Calculate $P_I^f(S_i,1)$ using formula (4.9)
2) Calculate $P_F^f(S_i,2)$ using formula (4.16)
3) Calculate $P_I^f(S_i,2)$ using formula (4.15)
4) repeat step 2) and 3) for t=3..T
5) Calculate $P_F^f(S_i,T+1)$ with formula (4.16)
6) Use formula (4.14) to calculate P(Y|W)

If we substitute equation (4.9) into (4.16) for t=1 and (4.15) in (4.16) for t>1 then we get relations (4.12) and (4.13). This shows that the algorithm where each output symbol is assigned to each state is mathematically fully equivalent to the algorithm where each symbol is associated with a transition and only depends on the destination state.

If the symbols are discrete, the probabilities of each state S_i producing a symbol $Y_t = m$ is stored in a matrix b_{im}. For this case we can replace:

$$p(i,m) = b_{im} \tag{4.17}$$

4.4 The Viterbi algorithm

Instead of adding the probabilities of all possible paths, an alternative approach is developed, only calculating the probability of the best path. This is called the Viterbi algorithm. The probability of the best path can be calculated by replacing all additions in (4.9)..(4.11) by the maximum operator.

The Viterbi probabilities are defined by:

$P^V_I(S_i,t) = P(Y_1..Y_{t-1},\text{best path initiating } S_i \text{ at time } t)$

$P^V_F(S_i,t) = P(Y_1..Y_{t-1},\text{best path finishing } S_i \text{ at time } t)$

This definition is similar to the forward probabilities. The difference with the forward probabilities is that instead of the sum of all path probabilities, only the probability of the best path is used. The Viterbi algorithm can be written in three versions, corresponding with the three versions of the Forward algorithm:

Version 1a) Only using $P^V_I(S_i,t)$: symbol associated with index i of a_{ij}

Version 1b) Only using $P^V_F(S_i,t)$: symbol associated with index j of a_{ij}

Version 2) Using both of them: symbol associated with state S_i.

If we start with, for example, version 1b), replacing all additions in the corresponding Forward algorithm by the maximum operator, the Viterbi algorithm becomes:

initialization: $$P^V_F(S_j,1) = a_{Ij} \cdot p(j,Y_1) \tag{4.18}$$

for t=2..T: $$P^V_F(S_j,t) = \max_i \left(P^V_F(S_i,t-1) \cdot a_{ij} \right) \cdot p(j,Y_t) \tag{4.19}$$

finally: $$P^V_F(Y_1..Y_T|W) = \max_i \left(P^V_F(S_i,T) \cdot a_{iF} \right) \tag{4.20}$$

$P^V(Y_1..Y_T|W)$ is the probability of the best path in word W that produces sequence $Y_1..Y_T$.

Now we are able to see the correspondence between the Viterbi algorithm and the asymmetrical Dynamic Time Warping algorithm. All we have to do is to transform all probabilities of the Viterbi algorithm into their logarithms. We define:

$$D(i,t) = -\log P_F^V(S_i,t) \qquad (4.21)$$
$$d(i,t) = -\log p(i,Y_t)$$

With these variables we can re-formulate the Viterbi algorithm as:

initialization: $$D(j,1) = -\log(a_{Ij}) + d(j,1) \qquad (4.22)$$

for t=2..T: $$D(j,t) = \min_i \left(D(i,t-1) - \log a_{ij} + d(j,t) \right) \qquad (4.23)$$

finally: $$\text{Distance} = \min_i \left(D(i,T) - \log a_{iF} \right) \qquad (4.24)$$

The above equations are a generalization of the asymmetrical DTW-algorithm where each state S_i corresponds to one template frame U_i. We see that the HMM model is much more general than the DTW method, in the sense that:

- Each state need not to correspond with one template frame.
- More than three transitions are allowed, with the corresponding penalties described by coefficients ($-\log a_{ij}$).

There is a price to be paid for in the HMM method for this improvement. Because the number of parameters has increased, the training becomes more complex. In the DTW method, the parameters were fixed and for the penalties it was sufficidnt to fill in some heuristically reasonable values. For HMM, a separate training algorithm has to be developed for training. Such an algorithm is known as the Forward-Backward algorithm, and it makes use of the probabilistic interpretation as introduced in this chapter.

4.5 Training of the HMM-data: The Forward-Backward algorithm

The training of all parameters in the HMM can be done as follows. Each parameter a_{ij} denotes the relative frequency of making a transition from state S_i to state S_j. If several training sequences are supplied, we could try to estimate how many of these transitions occur in the training sequences. Based on these estimates, a new value for a_{ij} can be calculated by dividing the number of times each transition occurs in the training sequence by the total number of expected transitions leaving from the same point. The same can be done for b_{im} in the case of discrete parameters. We introduce the following definitions:

τ_{ij}=	expected number of times that transition a_{ij} is made, given sequence $Y_1..Y_T$. Equivalent for τ_{Ij} and τ_{iF}.
υ_{im}=	expected number of times that state S_i emits symbol Y_t=m, given sequence $Y_1..Y_T$.
w_{it}=	probability that at time t state S_i produces symbol Y_t, given sequence $Y_1..Y_T$.

Each of the variables τ_{ij}, υ_{im} and w_{it} can be expressed in forward and backward probabilities. These backward probabilities are not yet defined, but they are symmetrical to the forward probabilities:

$$P^b_I(S_i,t) = P(Y_t..Y_T|S_i \text{ initiated at time } t)$$

$$P^b_F(S_i,t) = P(Y_t..Y_T|S_i \text{ finished at time } t)$$

They can be calculated by the backward algorithm that is symmetrical to the forward algorithm. The probability that at time T+1 the final node F is reached, given that state S_i is finished at that time, is described by the coefficients a_{iF}:

$$P^b_F(S_i,T+1) = a_{iF} \tag{4.25}$$

For any other time t=T..1, the probability that at time T+1 node F is reached starting at state S_i at time t, consists of making transition a_{ij} to all states S_j at time t and creating the remaining symbols:

$$P^b_F(S_i,t) = \sum_{j=1}^{N} a_{ij} \cdot P^b_I(S_j,t) \tag{4.26}$$

For any other time t=T-1..1 the probability of each state after a symbol Y_t has been emitted, can be written as:

$$P_I^b(S_i,t) = p(i,Y_t) \cdot P_F^b(S_i,t+1) \tag{4.27}$$

Finally, by the same reasoning we can easily see that:

$$P(Y|W) = \sum_{j=1}^{N} a_{Ij} \cdot P_I^b(S_j,T+1) \tag{4.28}$$

Summarized, the backward algorithm can be executed as:

1) Calculate $P_F^b(S_j,T+1)$ using formula (4.25)
2) Calculate $P_I^b(S_j,T)$ using formula (4.27)
3) Calculate $P_F^b(S_j,T)$ using formula (4.26)
4) repeat step 2) and 3) for t=T-1..1
5) Calculate $P_I^b(S_j,1)$ using formula (4.27)
6) Use formula (4.28) to calculate P(Y|W)

The probability of executing transition a_{ij} at time t is equal to the probability of emitting symbols $Y_1..Y_{t-1}$, finishing state S_i at time t, followed by a transition a_{ij} and followed by the probability of emitting the remaining output symbols. The extra assumption that $Y_1..Y_T$ is produced by this model is accounted for by dividing by $P(Y_1..Y_T|W)$. Adding this for all t=2..T (at t=1 a transition a_{Ij} is made) results in the expected number of times transition a_{ij} is made:

$$\tau_{ij} = \sum_{t=2}^{T} \frac{P_F^f(S_i,t) \cdot a_{ij} \cdot P_I^b(S_j,t)}{P(Y_1..Y_T|W)} \tag{4.29}$$

The same reasoning holds for τ_{Ij} and τ_{iF}, with the difference that transition a_{Ij} can only be performed at time t=1 and transition a_{iF} on time t=T+1. Therefore:

$$\tau_{Ij} = \frac{a_{Ij} \cdot P_I^b(S_j,1)}{P(Y_1..Y_T|W)} \;;\; \tau_{iF} = \frac{P_F^f(S_i,T+1) \cdot a_{iF}}{P(Y_1..Y_T|W)} \;; \tag{4.30}$$

The probability that state S_i is active at time t is equal to the probability of emitting symbols $Y_1..Y_{t-1}$ while S_i is initiated at time t, followed by the production of symbol Y_t, followed by the production of the remaining symbols $Y_{t+1}..Y_T$. Again we assume that sequence $Y_1..Y_T$ is produced by word W:

$$w_{it} = \frac{P_I^f(S_i, t) \cdot p(i, Y_t) \cdot P_F^b(S_i, t+1)}{P(Y_1..Y_T \mid W)} \tag{4.31}$$

If the symbols are discrete we can calculate the expected number of occurrences of each output symbol (having index m) at each state, by summing w_{it} for all t where symbol Y_t=m.

$$\upsilon_{im} = \sum_{t=1}^{T} w_{it} \Big|_{Y_t=m} \tag{4.32}$$

Hence, we can write as re-estimation formulas:

$$\hat{a}_{Ij} = \frac{\tau_{Ij}}{\sum_k \tau_{Ik}}; \quad \hat{a}_{ij} = \frac{\tau_{ij}}{\sum_k \tau_{ik} + \tau_{iF}}; \quad \hat{a}_{iF} = \frac{\tau_{iF}}{\sum_k \tau_{ik} + \tau_{iF}}; \tag{4.33}$$

When discrete symbols are used, we write:

$$\hat{b}_{im} = \frac{\upsilon_{im}}{\sum_k \upsilon_{ik}} \tag{4.34}$$

4.6 Continuous Parameter Distributions

If the symbols are continuous vectors, we can replace function $p(i,Y_t)$ with any other function. For instance if vector $\bar{u}$ produced by state S_i is supposed to be a random gaussian vector with mean $\bar{\mu}_i$ and covariance matrix Σ_i, we have instead:

$$p(i,\bar{u}) = \frac{|\Sigma_i|^{-1/2}}{(2\pi)^{N/2}} \cdot \exp\left(- \frac{(\bar{u}-\bar{\mu}_i)^* . \Sigma_i^{-1} . (\bar{u}-\bar{\mu}_i)}{2} \right) \tag{4.35}$$

For re-estimation of Σ_i and $\bar{\mu}_i$, remember that w_{it} refers to the probability that state S_i is producing a symbol at time t. If w_{it} is near 1, then state S_i is very likely at time t, while state S_i is very unlikely if w_{it} is near 0. In the re-estimation formulas for the mean vector $\bar{\mu}_i$ and covariance matrix Σ_i, w_{it} is used as a weight factor, in such a manner that likely states have a higher influence than unlikely states.

$$\hat{\mu}_i = \frac{\sum_t w_{it} \cdot \bar{y}_t}{\sum_t w_{it}} \tag{4.36}$$

$$\hat{\Sigma}_i = \frac{\sum_t w_{it} \cdot (\bar{y}_t - \hat{\mu}_i)^* \cdot (\bar{y}_t - \hat{\mu}_i)}{\sum_t w_{it}} = \frac{\sum_t w_{it} \cdot \bar{y}_t^* \cdot \bar{y}_t}{\sum_t w_{it}} - \hat{\mu}_i^* \cdot \hat{\mu}_i \tag{4.37}$$

These estimation formulas are proven by Liporace [39] to lead to the Maximum Likelihood solution.

4.7 Variable Duration Modelling

The probability to stay in some state S_i for d time periods depends on transition a_{ii}. Defining $P_d(S_i,d)$ as this probability, we can write:

$$P_d(S_i,d) = \begin{cases} 0 & \text{if } d \leq 0 \\ (1-a_{ii}) \cdot a_{ii}^{d-1} & \text{if } d>0 \end{cases} \tag{4.38}$$

We can easily check that:

$$\sum_{d=1}^{\infty} P_d(S_i,d) = (1-a_{ii}) \cdot \sum_{d=1}^{\infty} a_{ii}^{d-1} = \frac{(1-a_{ii})}{(1-a_{ii})} = 1 \tag{4.39}$$

The above formula shows that if state S_i is entered at time t, it is guaranteed to be finished at some time t+d with d>0. Figures representing this distribution using several values of a_{ii} are drawn in Fig 4.1.

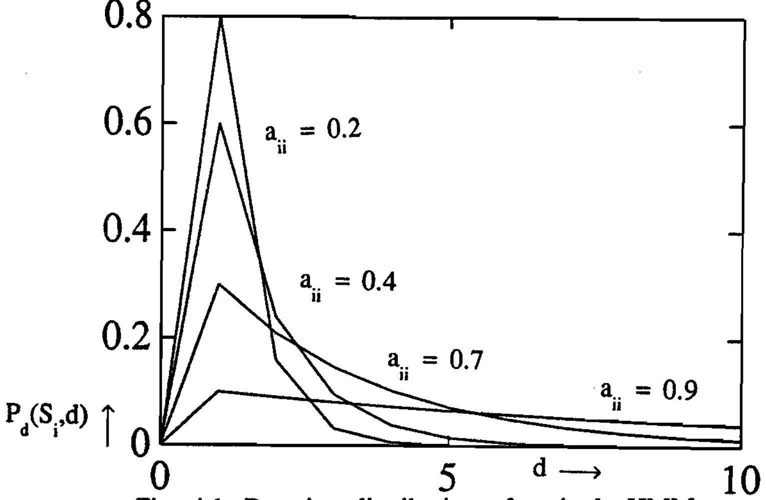


Fig. 4.1: Duration distribution of a single HMM-state

The expected duration of state S_i, or the expected number of frames the HMM will remain in state S_i, can be shown to be:

$$E(d) = \sum_{\delta=1}^{\infty} P_d(S_i,\delta)\cdot\delta = (1-a_{ii})\cdot \sum_{\delta=1}^{\infty} a_{ii}^{\delta-1}\cdot\delta = \frac{1}{(1-a_{ii})} \tag{4.40}$$

The expected duration of state S_i is not the duration with the highest probability. We can see in Fig. 4.1 that the most likely duration is 1. This is not what we would expect in practical situations. In practical speech each phoneme has some most likely duration. Durations shorter or longer than the most likely value should have a lower probability. Therefore different approaches are possible. One way is introducing alternative distributions like:

- Gamma distribution, see Levinson [37]:

  $$P_d(S_i,d) = \frac{\eta_i^{\nu_i}}{\Gamma(\nu_i)} \cdot d^{\nu_i-1}\, e^{-\eta_i\cdot d}$$

  mean value ν_i/η_i, variance ν_i/η_i^2
- Poisson distribution, see Russell [53]:

  $$P_d(S_i,d) = \frac{\lambda^d}{d\,!} \cdot e^{-\lambda}$$

  mean value λ, variance λ
- Binomial distribution, see Chang [13]:

  $P_d(S_i,d) = \binom{n}{d}\cdot p_i^d\cdot(1-p_i)^{n-d}$

  mean value $p_i\cdot n$, variance $p_i\cdot(1-p_i)\cdot n$

The easiest way to implement this is by replacing a single state S_i by the following structure of sub-states S_{id} (d=1..T), as shown in Fig. 4.2. The values of function $P_d(S_i,d)$ are the transition probabilities to each sub-state S_{id}.

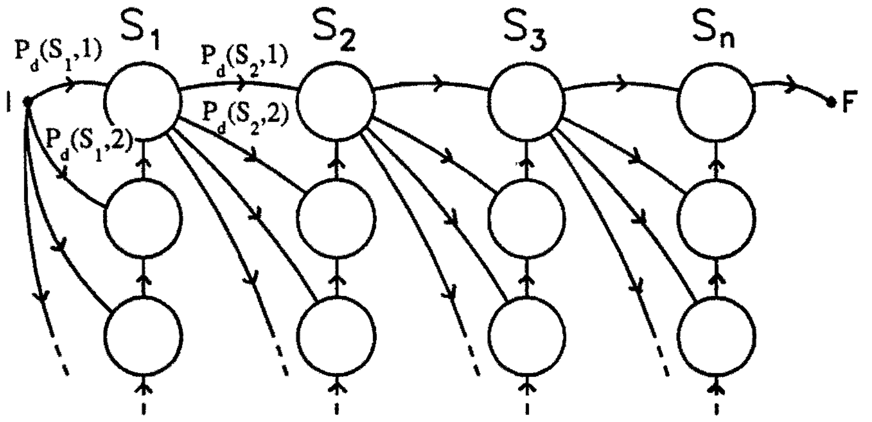


Fig. 4.2: Including state duration in HMM

For each sub-state we can write the following relations:

$$P^f_I(S_{id},t) = P^f_F(S_{id+1},t) + P_d(S_i,d) \cdot P^f_I(S_i,t) \tag{4.41}$$

$$P^f_F(S_{id},t+1) = P^f_I(S_{id},t) \cdot p(i,Y_t) \tag{4.42}$$

$$P^f_F(S_i,t) = P^f_F(S_{i1},t) \tag{4.43}$$

For the backward variables, we write:

$$P^b_I(S_{id},t) = P^b_F(S_{id},t+1) \cdot p(i,Y_t) \tag{4.44}$$

$$P^b_I(S_i,t) = \sum_{d=1}^{T} P^b_I(S_{id},t) \cdot P_d(S_i,d) \tag{4.45}$$

$$P^b_F(S_{id},t) = P^b_I(S_{id-1},t) \tag{4.46}$$

For each state more probabilities have to be stored (maximal T). Instead of O(N·T) the revised algorithm will be proportional to O(N·T^2). This can be reduced by introducing a minimum and a maximum duration for each state. The training of the parameters of the Poisson (λ_i), Gamma (ν_i,η_i) or Binomial (p_i) distribution can be performed by maximizing P(Y|W) with given sequences Y. This has been worked out in the already mentioned publications.

The three possibilities can be ranked in the following order:

Gamma:	Most complicated (2 parameters), and has therefore less restrictions.
Poisson:	Possesses only one parameter, and is therefore easier to calculate.
Binomial:	Possesses also only one parameter and, in addition, only n sub-states are needed.

Why the Binomial distribution is the easiest to calculate can be illustrated in an alternative way. Let us suppose we replace state S_i with multiple sub-states S_{id} as in Fig.4.4. The number of possible paths of duration d will be exactly $\binom{n}{d}$ while d times a transition p_i is made and n-d times transition $(1-p_i)$. This representation delivers exactly the binomial distribution, while resulting in very straightforward relations between all probabilities.

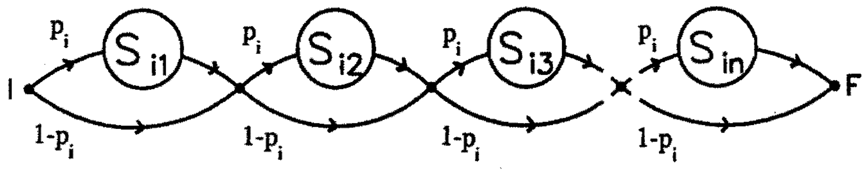


Fig. 4.4: representation of Binomial distribution in an HMM

The forward relations following this sub-division become:

$$P_I^f(S_{i1},t) = P_I^f(S_i,t) \cdot p_i \tag{4.47}$$

$$P_I^f(S_{id},t) = P_F^f(S_{id-1},t) \cdot p_i \tag{4.48}$$

$$P_F^f(S_{id},t+1) = P_I^f(S_{id},t) \cdot p(i,Y_t) + (1-p_i) \cdot P_I^f(S_{id-1},t) \tag{4.49}$$

The backward relations become:

$$P_F^b(S_{in},t) = P_F^b(S_i,t) \tag{4.50}$$

$$P_I^b(S_{id},t) = p(i,Y_t) \cdot P_F^b(S_{id},t+1) \tag{4.51}$$

$$P_F^b(S_{id},t) = p_i \cdot P_I^b(S_{id+1},t) + (1-p_i) \cdot P_F^f(S_{id+1},t) \tag{4.52}$$

The training of the coefficients p_i can be done by estimating the number of times any transition p_i and $1\text{-}p_i$ is made:

$$\tau_{p_i} = \sum_{t=1}^{T} \sum_{d=1}^{n} \frac{P_F^f(S_{id-1}, t) \cdot p_i \cdot P_I^b(S_{id}, t)}{P(Y_1..Y_T|W)} \tag{4.53}$$

$$\tau_{1-p_i} = \sum_{t=1}^{T} \sum_{d=1}^{n} \frac{P_F^f(S_{id-1}, t) \cdot (1-p_i) \cdot P_F^b(S_{id}, t)}{P(Y_1..Y_T|W)} \tag{4.54}$$

Hence, a new estimate $\hat{p}_i$ of p_i can be estimated by:

$$\hat{p}_i = \frac{\tau_{p_i}}{\tau_{p_i} + \tau_{1-p_i}} \tag{4.55}$$

Re-estimation formulas for the parameters of the Gamma and the Poisson distribution can be determined by maximizing $P(Y_1..Y_T|W)$, setting its derivatives in respect to all parameters to be trained to zero.

4.8 Conclusion

The models described in this chapter share some very important properties:

- Each word (or other chose unit) has its own model.
- Some 'score' is defined to judge the correspondence between the incoming speech and the model.
- This score can be calculated using a time-recursive procedure.
- Maximum score (or minimum, in the case of DTW) indicates the most suitable word, which is selected as the recognized word.
- The more parameters in the model, the more training data are needed for estimation of these parameters.

HMM, in principle, works the same as DTW, only here the number of parameters has increased and the 'score' metric has been improved (Forward instead of Viterbi). Making the model more complex, by introducing continuous symbols or different duration models, results in an increase of the number of parameters. However, the estimation of these additional parameters can still be determined using Bayes' rule, maximizing $P(Y_1..Y_T|W)$.

Many further extensions of the basic HMM are possible, further increasing the number of parameters to be estimated. The problems of these methods that are still unsolved can be grouped as (See Mariani [42]):

A. Speech has no separator. In fluent speech there is no silence between words.
B. Models are usually context dependent, which is difficult to include in HMM.
C. Variability due to different speakers in different speaking model (singing, shouting etc.).
D. The fundamental phoneme properties (what makes an 'a' an 'a'?) are still unknown. The only solution is comparing the speech with many examples (or a large model), which costs many calculations.
E. Speech is redundant. It contains more information than only the identity of the spoken words.
F. Speech has multiple levels of information, that are difficult to represent in a recognition system.

Point A, E and F are the ones that need special attention. These handle about the information inside the acoustic signal. The ideal speech recognition system should not use silences, but higher level information, in order to be able to distinguish various words.

Point B, C and D are problems that need further study by linguistic experts. Too little is known about the speech production rules used by humans. If these rules could be formulated, they could in principle be included in the recognition model.

5. THE RECURSIVE MARKOV MODEL

In this chapter the Recursive Markov Model (RMM) is defined, which is an extension of the Hidden Markov Model (HMM). Some examples show given how this model can form the basis of a speech recognition system. One of the powerful techniques that can be used within the RMM, is **State Sharing.** *An example demonstrates that state sharing makes use of the shared speech elements from everyday language, like common syllables used in many words, common phonemes used in many syllables etc.*

A new algorithm, which is an extension to the Forward Backward algorithm, is derived to train all parameters in the RMM. . In appendix A it is proved that the model parameters calculated by this algorithm always converge to a local maximum of the likelihood function.

As an extension to the Viterbi algorithm, the Recursive Viterbi algorithm is derived, which performs the recognition.

For every state machine, statistical properties are defined that give an indication of the complexity of the state machine. A new recursive algorithm is derived to calculate the entropy and state duration of all states in an RMM.

5.1 Definition of the Recursive Markov Model

In the preceding chapters we have seen how the HMM is implemented in current speech recognition systems. Many extensions to HMM have been proposed, such as for example duration and language modelling. These extensions deviate more and more from the original model. For all extensions additional parameters have been introduced and more training data was needed. Computation time for recognition and training increases as well.

For a practical speech recognition system, the number of parameters will have to be reduced without simplifying the model. This can be done by sharing common parameters. For example, if a library contains the words 'four', 'fourteen' and 'fourty', then these words have the first syllable in common. However, HMM does not contain a technique to combine the parameters of this syllable. In contrast with that, an extension to HMM is introduced in this chapter, which makes such a sharing approach possible. It turns out that the well-known Forward Backward and Viterbi algorithms, as used in HMM training and recognition, can be adapted to the new model.

A HMM consists of a collection of states and a set of transition probabilities. At every frame time the model will change state in accordance with these transition probabilities. Also at every frame time an output symbol is produced, according to another probability law defined on the states.

The extension of the HMM as presented in this chapter will be called RMM, which differs from HMM in the states that are allowed to produce more than one symbol. This means that a transition does not have to occur every frame time. Elementary states are introduced to be compatible with HMM-states.

Definition of RMM:

- A RMM consists of a collection of states and transitions. Each transition is associated with a parameter, describing its probability of occurance.
- These states can be elementary or non-elementary. Elementary states are similar to the states in HMM.
- Elementary states are active during one frame time period, and produce a single symbol.
- Non-elementary states are active during one or more frame time periods, and produce one symbol for each active frame time.

Both elementary and non-elementary states need to be further specified.

Elementary states:

- A function $P_E(S,U)$ is specified to describe the probability (or probability density) that an elementary state S produces symbol U. This symbol may be discrete or continuous. Any multivariate random function for $P_E(S,U)$ is allowed.

Non-elementary states:

- A non-elementary state S consists of a set of child states and a set of transition probabilities stored in a matrix A.
- The elements of A only depend on the assigned state S, and not on time t.
- If state S is initiated, a transition is made to one of its child states, according to the associated transition probability from matrix A.
- After a child state has finished to produce symbols, either a transition is made to another child state or state S itself has finished. These probabilities are also stored in matrix A.

If a transition is made from state S_i to S_j at time t, we say that state S_i has finished and S_j is initiated at time t. If S_i is initiated at time $t=t_0$ and has finished at time $t=t_1$, then we can state that S_i is active in the interval $[t_0,t_1>$ (t_0 is included but not t_1). In the RMM, transitions do not take time. The production of an output symbol by an elementary state however takes one time unit.

The Variable Duration Modelling, as described in Section 4.6, is an extension to HMM which allows more symbols to be produced for each state. This is equivalent to a replacement of a state with multiple sub-states, as shown in Fig. 4.2. In RMM, non-elementary states are introduced which can be employed for the same purpose in a more versatile way. Therefore, this Variable Duration Modelling is already contained in the RMM.

An example of an RMM with three states is drawn in Fig. 5.1:

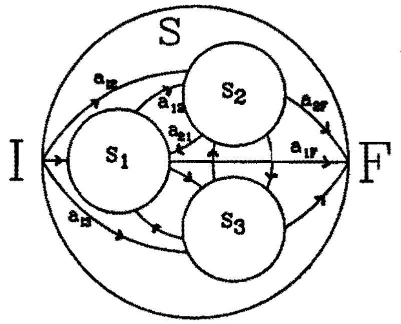


Fig. 5.1: Example RMM with 3 states.

For all states that produce more than one symbol, a further specification is needed. Such a non-elementary state is supposed to be a Markov chain of sub-states. In the same way, these sub-states must be further specified if they also produce more than a single symbol. This recursive process ends in elementary states that produce only a single symbol. The name Recursive Markov Model (RMM) refers to this recursive way of splitting states into sub-states. The whole RMM can be considered as a single Root state, containing all other states.

Let us now consider some state S and its sub-states $S_1..S_N$. State S itself may be a sub-state of some other state, and S_i may be split further into smaller sub-states. Such an organization looks like a tree structure, which describes the relation between the states. On top will be a single Root state, representing the whole RMM. On the bottom we find all elementary states as leaves of the tree, as is shown in Fig. 5.2.

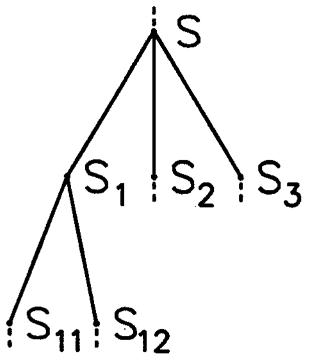


Fig. 5.2: Tree structure of an RMM

Attached to State S with sub-states $S_1..S_N$ is a matrix A which contains all state transition probabilities, which are assumed to be independent of time t:

$$A = \begin{pmatrix} a_{I1} & & a_{IN} & 0 \\ a_{11} & .. & a_{1N} & a_{1F} \\ \vdots & \ddots & \vdots & \vdots \\ a_{N1} & .. & a_{NN} & a_{1F} \end{pmatrix} \tag{5.1}$$

a_{Ij} = P(S_j is initiated at time t | S is initiated at t)
a_{ij} = P(S_j is initiated at time t | S_i has finished at t)
a_{iF} = P(S has finished at time t | S_i has finished at t)

The first row of the matrix contains the probabilities for transitions to one of $S_1..S_N$, after state S is initiated. No other transitions are possible. The other rows contain the probabilities of initiating S_j or finishing S, after S_i is finished. One and only one of these transitions can be made at each time t, independent from t. This puts some restrictions on the values of a_{Ij}, a_{ij} and a_{iF}:

$$\sum_{j=1}^{N} a_{Ij} = 1; \qquad \sum_{j=1}^{N} a_{ij} + a_{iF} = 1; \qquad \{i=1..N\} \tag{5.2}$$

Each non-elementary state S has such a matrix A attached to it. For elementary states another approach is needed. For these a function $P_E(S,U)$ has been defined on page 64 as the probability of producing symbol U:

$$P_E(S,U) = P(\text{symbol } U \text{ emitted} \mid \text{state } S)$$

If the output symbols are discrete, each elementary state S has an attached symbol Y(S). Thus the function $P_E(S,U)$ can be specified as:

$$P_E(S,U) = \begin{cases} 1 & \text{if } U=Y(S) \\ 0 & \text{otherwise} \end{cases}$$

For the following sections no assumption will be made about the function $P_E(S,U)$, except that it must be a valid probability (density) function. Summed for all possible output symbols (or integrated for continuous symbols) the result must yield 1. For now we only assume that $P_E(S,U)$ is known for each elementary state S and can be evaluated. This assumption will be a sufficient condition to derive and prove the correctness of the Recursive Forward-Backward and the Recursive Viterbi algorithm. In Section 5.4 the training of the elementary states will be considered.

An important feature of RMM is the possibility of **State Sharing**. This allows for more complex model descriptions without increasing the number of parameters to be trained and it is equivalent to tied transition probabilities in HMM. With RMM the recursive model enables the description of much more complex relations between states, which recursion can easily be used in the training and recognition algorithms. This will be illustrated by the following example.

Suppose, an RMM has been constructed for modelling the pronunciation of all numbers between 1 and 999999. The Root state (1-999999), representing all possible numbers, can be divided in 4 sub-states. The first and the last of these, representing the numbers 1-999, are identical. The structures of the Root state (1-999999) and state (1-99) are shown in Fig. 5.3.

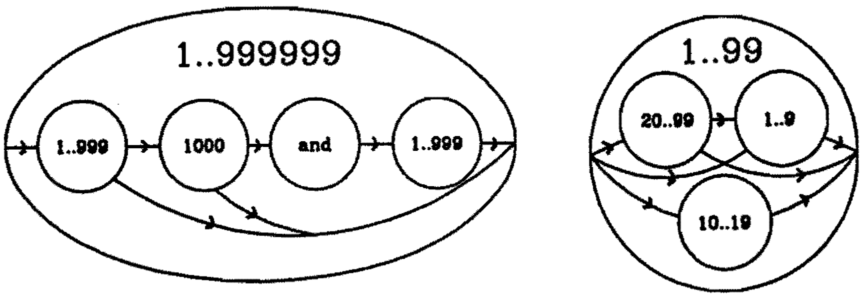


Fig. 5.3: An example RMM: numbers 1-999999

Each of the states in this model has its own matrix A. However, because the first and the last child state of (1-999999) are identical, it makes sense to form one matrix A which is shared between these two states. In a high-level language like C or Pascal this sharing is easily implemented by using pointers. If state (1-999) is decomposed further into smaller states, new identical states will arise. For instance, there may eventually exist states named 'four' in eight different places. These represent the eight different functions of 'four', which are all used in the words 'fourtyfourthousendandfourtyfour' and 'fourhundredfourteenthousendandfourhundredfourteen'. Because these states not only share their matrices A but also the full lower level state structure, this principle is called **State Sharing**. All algorithms derived in this thesis operating on an RMM fully support State Sharing.

In a tree like in Fig. 5.2 each child state only has one parent. If State Sharing is allowed, states could have more than one parent state. The tree structure changes into a directed graph, as is shown on page 149. The full model of this example can be found in appendix C, page 143-149. In the graph, eight different paths to 'four' exist, which indicates the eight different functions that this word can have.

5.2 The Recursive Forward Backward algorithm

The Recursive Forward-Backward algorithm is an iterative training procedure which calculates new matrices $\hat{A}$ from old estimates A, in such a manner that the probability that the RMM would produce the given training sentences will be increased. This algorithm is guaranteed to reach a local maximum of the likelihood function. This property has already been proven by Baum [10] for the HMM and extended by Liporace [39] for continuous output distributions with ellipsoidal symmetry. In Appendix A we will prove that this also holds for the re-estimation of matrices A and functions $P_E(S,U)$ by the methods derived in this chapter, even with multiple training sequences and State Sharing.

As each state S has its own matrix A, we should always write a^S_{ij} instead of a_{ij}. However, because all relations derived are only local to a single state S, no misunderstanding can occur about the state which is meant. Therefore, index S will be removed and assumed implicitly.

The Recursive Forward Backward algorithm will now be derived in the next sections.

5.2.1 The Forward recursion

Let $Y_1..Y_T$ be an observed sequence of symbols, and P(...) represent the probability (or probability density) of the description between parenthesis. The following notation will be introduced:

$$\boxed{\begin{aligned} P^f_I(S,t) &= P(Y_1..Y_{t-1},\ \text{S is initiated at time t}) \\ P^f_F(S,t) &= P(Y_1..Y_{t-1},\ \text{S has finished at time t}) \\ P^f(S,t) &= P(Y_1..Y_t\ ,\ \text{S is active at time t}) \end{aligned}} \tag{5.3}$$

In words: $P^f_I(S,t)$ represents the probability that state S produces the sequence $Y_1..Y_{t-1}$ and state S is initiated at time t.

These probabilities are called the forward probabilities, because they can be calculated in a forward recursive way. The interrelations among the probabilities are derived for the various states as follows:

a) Initiation of **elementary state** S at time t will automatically mean producing a symbol Y_t, being active at time t and finishing at time t+1, hence for **elementary states:**

$$P_F^f(S,t+1) = P^f(S,t) = P_I^f(S,t) \cdot P_E(S,Y_t) \qquad (5.4)$$

b) **Non-elementary states.** For these states we have to consider three possibilities:

1) Any sub-state S_j of a **non-elementary** state S can only be <u>initiated</u> if S is initiated and a transition to S_j is made, or any state S_i has finished and a transition to S_j is made. These transitions are independant, hence:

$$P_I^f(S_j,t) = a_{Ij} \cdot P_I^f(S,t) + \sum_i a_{ij} \cdot P_F^f(S_i,t) \qquad (5.5)$$

2) A **non-elementary** state S can only have <u>finished</u> when any of its sub-states S_i has finished and a transition is made that finishes state S, yielding:

$$P_F^f(S,t+1) = \sum_i a_{iF} \cdot P_F^f(S_i,t+1) \qquad (5.6)$$

3) Finally, **non-elementary** state S can only be <u>active</u> if one of its sub-states is active:

$$P^f(S,t) = \sum_i P^f(S_i,t) \qquad (5.7)$$

If S is an **elementary state,** $P_F^f(S,t+1)$ and $P^f(S,t)$ can be calculated directly using (5.4). Otherwise, the problem is recursively divided in smaller problems by the following algorithm:

Recursive Forward Algorithm (for **non-elementary** states)

```
Given: $P_I^f(S,t)$, $P_F^f(S_j,t)$ for every sub-state $S_j$
calculate: $P^f(S,t)$ and $P_F^f(S,t+1)$
--------------------------------------------------------------
  1. for every child $S_j$:
       $P_I^f(S_j,t) = a_{Ij} \cdot P_I^f(S,t) + \sum_i a_{ij} \cdot P_F^f(S_i,t)$        (5.5)
       calculate $P^f(S_j,t)$, $P_F^f(S_j,t+1)$ (Recursive)
  2. finally:  $P^f(S,t) = \sum_i P^f(S_i,t)$                                              (5.6)
               $P_F^f(S,t+1) = \sum_i a_{iF} \cdot P_F^f(S_i,t+1)$                         (5.7)
```

This algorithm contains two recursions. One of these is the time recursion, similar to HMM. It means that for the calculation of the probabilities at time t all probabilities on time t-1 must have been determined. The other recursion is called the state recursion, which works up and down in the hierarchy. Formula (5.5) expresses the initial probability of S_j in the initial probability of its parent state S. The formulas (5.6) and (5.7) work the opposite way. They express the state probability and the final probability of state S, in the state probabilities and the final probabilities of all the sub-states S_i. Assembling these formulas together in the given order, assures that all probabilities are known when they are needed.

To initialize all variables, we assume that the Root state is initiated at time t=1. This means that:

$$P^f_I(\text{Root},t)=\begin{cases}1: & t=1\\ 0: & t\neq 1\end{cases};\quad P^f(S,0)=0;\quad P^f_F(S,1)=0;\quad \text{for all states } S \tag{5.8}$$

Starting at t=1, the Recursive Forward Algorithm can be used to calculate all initial and state probabilities at t=1 and all final probabilities at t=2. This can be repeated for t=1..T. So, all forward probabilities can be determined.

5.2.2 The Backward recursion

The backward probabilities $P^b_.(..)$ are defined as:

$$\boxed{\begin{aligned} P^b_I(S,t) &= P(Y_t..Y_T \mid S \text{ is initiated at time } t)\\ P^b_F(S,t) &= P(Y_t..Y_T \mid S \text{ has finished at time } t)\\ P^b(S,t) &= \begin{cases} P^b_I(S,t) & \text{if } S \text{ is an elementary state}\\ \sum_j P^b(S_j,t) & \text{otherwise}\end{cases}\end{aligned}} \tag{5.9}$$

These probabilities are called 'backward', because they can be calculated in a backward recursive way. The difference in definition with (5.3) is necessary to end up with a symmetrical algorithm. $P^b(S,t)$ cannot be interpreted as a probability, because it may become >1 during the Recursive Backward algorithm. However, as described in chapter 6, this quantity will prove to be very useful later on.

Using the same steps as in the forward recursion, we obtain the following relations, which are equivalent to (5.4)..(5.7). Again we subdivide into elementary and non-elementary states:

a) An **elementary state** S, initialized at time t, will produce a symbol Y_t and must finish at time t+1:

$$P_I^b(S,t) = P^b(S,t) = P_E(S,Y_t) \cdot P_F^b(S,t+1) \qquad (5.10)$$

b) **Non-elementary states.** Again three situations must be considered:

1) A **non-elementary** state S, after having <u>finished</u> a sub-state S_i, either must make a transition to finish S or initialize another sub-state S_j:

$$P_F^b(S_i,t+1) = a_{iF} \cdot P_F^b(S,t+1) + \sum_j a_{ij} \cdot P_I^b(S_j,t) \qquad (5.11)$$

2) After **non-elementary** state S is <u>initialized</u> a transition will be made to some sub-state S_j, which will be initialized too.

$$P_I^b(S,t) = \sum_j a_{Ij} \cdot P_I^b(S_j,t) \qquad (5.12)$$

3) From the definition (5.9) we conclude that for **non-elementary** state S:

$$P^b(S,t) = \sum_j P^b(S_j,t) \qquad (5.13)$$

If S is an **elementary** state, $P^b(S,t)$ and $P_I^b(S,t)$ can be calculated in a direct way using (5.10). Otherwise the problem is divided recursively into smaller problems by the Recursive Backward Algorithm:

Recursive Backward Algorithm (for **non-elementary** states)

```
Given: $P_F^b(S,t+1)$, $P_I^b(S_i,t+1)$ for every sub-state $S_i$
calculate: $P^b(S,t)$ and $P_I^b(S,t)$
--------------------------------------------------------------
 1. for every child $S_i$:
      $P_F^b(S_i,t+1) = a_{iF} \cdot P_F^b(S,t+1) + \sum_j a_{ij} \cdot P_I^b(S_j,t+1)$
      calculate $P^b(S_i,t)$, $P_I^b(S_i,t+1)$ (Recursive)
 2. finally:   $P^b(S,t) = \sum_j P^b(S_j,t)$
               $P_I^b(S,t) = \sum_j a_{Ij} \cdot P_F^b(S_j,t)$
```

As an initialization, the RMM is assumed to finish the Root state at time T+1:

$$P^b_F(\text{Root},t)=\begin{cases}1:t=T+1\\0:t\neq T+1\end{cases};\; P^b(S,T+1)=0;\; P^b_I(S,T)=0;\text{ for all states S} \qquad (5.14)$$

5.2.3 The training of the model

With the two algorithms from the previous sections, all forward and backward probabilities can be calculated. The actual goal is to train the model parameters A of all non-elementary states: given initial estimates A, we want to calculate better estimates $\hat{A}$ for all states. These estimates $\hat{A}$ can be approximated from frequency distributions for the transition probabilities calculated with the Recursive Forward and Recursive Backward algorithms. This approximation can be continued, until it converges to a solution that maximizes $P^f_F(S,T+1)$. For simplicity, in this chapter the relations are derived in a heuristic way. In appendix A is proved that the parameters in the model converge to a (local) maximum of the likelihood function $P^f_F(S,T+1)$.

Let τ_{ij} be the expectation value for the number of times the transition a_{ij} is made given the sequence $Y_1..Y_T$. Equivalently τ_{Ij} and τ_{iF} are defined. Like the case for the coefficients a_{ij}, each state S also has its own τ_{Ij}, τ_{ij} and τ_{iF}. For reasons of simplicity, we do not write τ^S_{ij}, to indicate which state is meant. Since all relations derived are only valid locally for some state S, we implicitly assume that this state is meant. We can determine the τ_{Ij}, τ_{ij} and τ_{iF} from the calculated forward and backward probabilities. Performing transition a_{ij} at time t is equivalent to first finishing some state S_i producing $Y_1..Y_t$, next making transition a_{ij}, and finally producing $Y_{t+1}..Y_T$. However, the expectation value for τ_{ij} is determined by the sum of the contributed probabilities for all t divided by the probability of all possible paths:

$$\boxed{\begin{aligned} \tau_{Ij} &= \frac{\sum_t P_I^f(S,t)\cdot a_{Ij}\cdot P_I^b(S_j,t)}{P_F^f(\mathrm{Root},T+1)} \\ \tau_{ij} &= \frac{\sum_t P_F^f(S_i,t)\cdot a_{ij}\cdot P_I^b(S_j,t)}{P_F^f(\mathrm{Root},T+1)} \\ \tau_{iF} &= \frac{\sum_t P_F^f(S_i,t)\cdot a_{iF}\cdot P_F^b(S,t)}{P_F^f(\mathrm{Root},T+1)} \end{aligned}} \qquad (5.15)$$

The coefficients a_{ij} are re-estimated by dividing the expected number of transitions from state S_i to S_j by the total expected number of transitions finishing S_i:

$$\boxed{\hat{a}_{Ij} = \frac{\tau_{Ij}}{\sum_k \tau_{Ik}}; \quad \hat{a}_{ij} = \frac{\tau_{ij}}{\sum_k \tau_{ik} + \tau_{iF}}; \quad \hat{a}_{iF} = \frac{\tau_{iF}}{\sum_k \tau_{ik} + \tau_{iF}}} \qquad (5.16)$$

The update of τ_{ij}, τ_{Ij} and τ_{iF} can be combined with the backward algorithm. We have to make sure that only quantities are used that have been calculated before. It turns out that τ_{ij} can be updated before step 1 of the Recursive Backward algorithm is executed, and τ_{Ij} and τ_{iF} can be calculated between step 1 and step 2. Adding these two steps to the Recursive Backward algorithm results in the Recursive Backward Training Algorithm. The advantage of this set-up is that all calculated quantities that are not needed any more for the recursion neither are needed for the τ-update any longer. Hence, these quantities do not have to be stored, which conserves memory.

Let τ_{ij}^t denote the righthand side of (5.15) when the summation is performed from time t..T, instead of time 1..T. The algorithm becomes:

Recursive Backward Training Algorithm (for non-elementary state S)

```
Given: $P^b_F(S,t+1)$, $P^b_I(S_i,t+1)$ for every sub-state $S_i$
calculate: $P^b(S,t)$ and $P^b_I(S,t)$
------------------------------------------------------------
1. update $\tau_{ij}$ for all i,j:
       $\tau^{t+1}_{ij} = \tau^{t+2}_{ij} + \frac{P^f_F(S_i,t+1)\cdot a_{ij}\cdot P^b_I(S_j,t+1)}{P^f_F(\mathrm{Root},T+1)}$
2. for every child $S_i$:
       $P^b_F(S_i,t+1) = a_{iF}\cdot P^b_F(S,t+1) + \sum_j a_{ij}\cdot P^b_I(S_j,t+1)$
       calculate $P^b(S_i,t)$, $P^b_I(S_i,t)$ (Recursive)
3. update $\tau_{Ij}$ and $\tau_{iF}$:
       $\tau^{t+1}_{Ij} = \tau^{t+2}_{Ij} + \frac{P^f_I(S,t)\cdot a_{Ij}\cdot P^b_I(S_j,t)}{P^f_F(\mathrm{Root},T+1)}$
       $\tau^{t}_{iF} = \tau^{t+1}_{iF} + \frac{P^f_F(S_i,t+1)\cdot a_{iF}\cdot P^b_F(S,t+1)}{P^f_F(\mathrm{Root},T+1)}$
4. finally:   $P^b(S,t) = \sum_j P^b(S_j,t)$
              $P^b_I(S,t) = \sum_j a_{Ij}\cdot P^b_F(S_j,t)$
```

The above way of processing has another advantage. If state sharing is used, the transition probabilities of two or more states are forced to have the same value. The usual way of processing this is to use the sum of the expectations for each copy of a_{ij} for the update. In our algorithm this is automatically performed if we share the variables τ_{ij} between the shared states, just like matrix A is shared between the same states. During the backward algorithm all contributions from the different states will be added together automatically.

If the coefficients have to be trained by using more training sequences, each training sequence delivers its own values for τ_{ij}. The total expected number of transitions in all training sequences is the sum of the expected number of transition from each training sequence. This sum can also be obtained by performing the Forward Backward Training Algorithm for the second to the last training sequence without initializing τ_{ij}, τ_{Ij} and τ_{iF} again. The procedure for multiple training sequences then becomes:

1. initialize $\tau_{ij} = \tau_{Ij} = \tau_{iF} = 0$ for all states.
2. perform the Recursive Forward algorithm for the first training sequence, for t=1..T.
3. perform the Recursive Backward Training algorithm to update τ_{ij}, τ_{Ij} and τ_{iF}, for t=T..1.
4. repeat step 2 and 3 for all other training sequences, without re-initializing τ_{ij}.
5. update $\hat{A}$ for all states, using (5.16).

This forgoing algorithm will train all coefficients in the RMM. State Sharing and multiple training sequences are handled by sharing variables between states. This heavily reduces memory requirements. In the algorithm only local variables are used, except for one value: $P_F^f(\text{Root},T+1)$.

Some additional improvement can be obtained by storing τ_{ij}/a_{ij} instead of τ_{ij}, and correspondingly for τ_{Ij} and τ_{iF}. Because a_{ij} is a constant factor in τ_{ij}, as can be seen in (5.15), the multiplication by a_{ij} can be done after all training sequences are processed.

If training is done with a single training sequence only, the division by $P_F^f(\text{Root},T+1)$ has no influence on the training and need not be performed. However, the training is normally performed with many training sequences. Removing this division in the Recursive Backward Training algorithm has the same effect as weighting the influence of each training sequence in the training. Sequences with a higher probability will have a higher influence on the training than sequences with a lower probability. This effect is not desirable, because sequences with a low probability indicate that the RMM is not yet trained sufficiently.

These sequences should rather be given a high weight in the training. Sequence length differences have the same adverse effect. Longer training sequences will automatically have a much lower probability, but they contain much more data. In Appendix A is shown that the factor $P^f_F(\mathrm{Root},T+1)$ is essential for the algorithm to guarantee Maximum Likelihood estimation.

Another remark is that a different initialization is possible as well. The training as described will cause problems if some transition, say a_{ij} of state S, never occurs in the training data. This will result in the expected number of transitions $\tau_{ij} = 0$ (or at least $\ll 1$). This does not mean that in reality this transition is not allowed to occur, but the re-estimation formulas (5.16) will give a zero result for τ_{ij}. A way to cope with this situation is to set $\tau_{ij} = a_{ij}$ instead of initializing $\tau_{ij} = 0$. If transition a_{ij} does not occur in the training data, a_{ij} will not change during the training. If a_{ij} occurs many times in the training data then $\tau_{ij} \gg 1$, so adding a_{ij} has only little influence. It looks like another training sequence has been added which follows exactly the current transition probabilities. Another advantage of this initialization is that the denominator of (5.16) is guaranteed to be ≥ 1, hence division by zero will not occur.

Conclusion:

- The Recursive Forward Backward Training Algorithm is able to train the matrices A of all states, given multiple training sequences $Y_1..Y_T$.
- Multiple training sequences and State Sharing are handled by sharing variables τ_{ij} in the training algorithm between all shared states and all training sequences. No fundamental change in the algorithm itself is needed.
- The training for each non-elementary state S only uses local variables of state S and all its child states S_i, except for $P^f_F(\mathrm{Root},T+1)$.

5.3 The Recursive Viterbi algorithm

The actual problem to solve is the recognition problem. Given a symbol sequence, the corresponding word sequence with the highest probability has to be determined. In HMM each word has its own model, but in RMM the total library is represented by the Root state and the words are represented by sub-states as well.

We can re-formulate the recognition problem as: *given a symbol sequence* $Y_1..Y_T$, *what is the state-sequence (on all levels) with the highest probability of producing* $Y_1..Y_T$.

The recognition can be produced by the Recursive Viterbi algorithm. Like the Recursive Forward Backward algorithm, this algorithm consists of two parts. The first part calculates the probabilities of the best path reaching any state for t=1..T. The second part back-traces the states in these paths, so that afterwards the best state sequence can be reproduced.

To start, only the calculation of the probabilities of the best path reaching any state will be considered. After that, the tracing algorithm will be explained, showing that at each time t the state sequence of the best path can be reproduced. The probabilities used in the algorithm will be called the Viterbi probabilities, because they are calculated in a Viterbi-type recursion. This algorithm will have a structure very similar to the Recursive Forward algorithm. Therefore the needed variables are defined in a similar way, hereby enabling the same structure used in the previous section to be used in the Recursive Viterbi algorithm.

The Viterbi probabilities are defined by:

$$\boxed{\begin{aligned} P^V_I(S,t) &= P(Y_1..Y_{t-1}, \text{ best path initiating } S \text{ at time } t) \\ P^V_F(S,t) &= P(Y_1..Y_{t-1}, \text{ best path finishing } S \text{ at time } t) \\ P^V(S,t) &= P(Y_1..Y_t\ , \text{ best path activating } S \text{ at time } t) \end{aligned}} \qquad (5.17)$$

As before, we again differentiate between elementary and non-elementary states:

a) An **elementary state** S <u>initiated</u> at time t will produce a symbol Y_t and finish at time t+1. The best path leading towards state S at time t is also the best path finishing S at time t+1. The Viterbi probabilities of elementary state S have the relation:

$$P^V_F(S,t+1) = P^V(S,t) = P^V_I(S,t) \cdot P_E(S,Y_t) \qquad (5.18)$$

b) **Non-elementary states.** Again three situations are possible:

1) For a **non-elementary state** S the best path <u>initiating</u> S_j at time t must be selected from the best path initiating S at time t and the paths finishing other states S_i, and making a transition to S_j:

$$P^V_I(S_j,t) = \max\left[a_{Ij} \cdot P^V_I(S,t) \text{ , } \max_i \left[a_{ij} \cdot P^V_F(S_i,t) \right] \right] \qquad (5.19)$$

2) A **non-elementary state** S, can only have <u>finished</u> if its sub-states have finished and a transition is made to the finishing node of S. To calculate the probability of the best paths finishing S, the probability of the paths finishing S_i must be calculated and the maximum selected:

$$P^V_F(S,t+1) = \max_i \left[a_{iF} \cdot P^V_F(S_i,t+1) \right] \qquad (5.20)$$

3) The best path that <u>activates</u> **non-elementary state** S at time t, can only be one of the best paths that activates S_i:

$$P^V(S,t) = \max_i \left[P^V(S_i,t) \right] \qquad (5.21)$$

Comparing these relations with (5.4)..(5.7), we see that the relations between the Viterbi probabilities are similar to the relations of the Forward probabilities. If in (5.4)..(5.7) the index 'f' is replaced by 'v' and all additions are replaced by the maximum operator. This yields exactly the Viterbi relations.

Previously we have derived the Recursive Forward Algorithm. The known relations had to be arranged in such order that variables are only used after they have been calculated. Because of the similarity, we can formulate the Recursive Viterbi algorithm right away by replacing

'Forward' with 'Viterbi' probabilities and additions by the maximum-operator in the Recursive Forward algorithm:

The Recursive Viterbi Algorithm

```
Given: $P^V_I(S,t)$, $P^V_F(S_j,t)$ for every sub-state $S_j$
calculate: $P^V(S,t)$ and $P^V_F(S,t+1)$
------------------------------------------------------------
  1. for every child $S_j$:
        $P^V_I(S_j,t) = \max\left[a_{Ij}\cdot P^V_I(S,t)\ ,\ \max_i\left(a_{ij}\cdot P^V_F(S_i,t)\right)\right]$
        calculate $P^V(S_j,t)$, $P^V_F(S_j,t+1)$ (Recursive)
  2. finally:   $P^V(S,t) = \max_i\left(P^V(S_i,t)\right)$
                $P^V_F(S,t+1) = \max_i\left(a_{iF}\cdot P^V_F(S_i,t+1)\right)$
```

After all Viterbi probabilities have been calculated for t=1..T, we want to find the best path up to time T from these data. Starting with the Root state, we can compare $P^V(S_i,T)$ for all sub-states S_i of the Root. The largest value indicates the state that contains the most probable path. Then all sub-states of the selected state S_i can be compared in the same way. When an elementary state has been reached (say S_E), this state is the most likely state at time T. Knowing that the best path is initiating S_E at time t, we can trace it back, comparing the already calculated Viterbi probabilities. This method contains two disadvantages:

- all Viterbi probabilities must be stored
- the best path only can be determined after all symbols $Y_1..Y_T$ have been processed.

An alternative for the previous algorithm is to store for each state a list of previous states. This list can be updated with every max-operation, and pruned when the state is pruned. Additional memory is needed to store these lists for all states, but the advantage is that Viterbi probabilities for previous times do not need to be stored any more. The same memory locations used at time t, can be used at time t+1 again. At any time t during the Recursive Viterbi algorithm the best state up to time t can be found, and the best path leading to this state is immediately available. This search strategy is implemented, and will be treated in more detail in Chapter 7.

5.4 Training of elementary state parameters

In this section elementary states are considered. Until now we simply defined a function $P_E(S,U)$ for every elementary state. For the Forward-Backward and Viterbi algorithm, this function has to be evaluated for all elementary states and all symbols $Y_1..Y_T$. If the function contains parameters that have to be trained, additional re-estimation formulas for the elementary states have to be specified.

The probability density function of many stochastic processes that occur in nature, can be approximated well by the Gaussian distribution. If many different Gaussian distributions are available , each with a different mean and covariance, these can be combined to a multivariate Gaussian distribution. Any continuous probability distribution can be approximated at any desired accuracy by multiple Gaussian distributions with different mean and covariance matrix. Therefore a suitable choice for an elementary state S would be (* = transpose):

$$P_E(S,\bar{u}) = \frac{|\Sigma|^{-1/2}}{(2\pi)^{N/2}} \cdot \exp\left[- \frac{(\bar{u}-\bar{\mu})^{*}\cdot\Sigma^{-1}\cdot(\bar{u}-\bar{\mu})}{2} \right] \qquad (5.22)$$

$$\text{with: } \bar{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_N \end{pmatrix}; \quad \bar{\mu} = \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_N \end{pmatrix}; \quad \Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12}..\sigma_{1N} \\ \sigma_{21} & \sigma_2^2\,..\sigma_{2N} \\ \vdots & \vdots \\ \sigma_{N1} & \sigma_{N2}..\sigma_N^2 \end{pmatrix}$$

This probability function can be applied safely to the RMM.

In Section 5.2 we considered multiple training sequences $Y_1..Y_T$. These are used in the Recursive Forward Backward Training algorithm to estimate the matrices A for all non-elementary states. If these symbols Y_t are discrete, they are fixed and cannot be trained. In this section we will consider training of the parameters $\bar{\mu}$ and Σ of each elementary state, using multiple training sequences $\bar{y}_1..\bar{y}_T$ which are now continuous N-dimensional vectors.

By specifying an N-dimensional mean vector $\bar{\mu}$ and a positive definite N x N covariance matrix Σ for each elementary state, multiple states can create any mixed Gaussian distribution. State sharing must also be allowed, so multiple elementary states can share the same $\bar{\mu}$ and Σ. Again index S is dropped from $\bar{\mu}$ and Σ, because all update formulas only refer to a single state S. It is implicitly assumed that $\bar{\mu}$ and Σ refer to this state S.

Consider any elementary state S, that is assumed to produce an N-dimensional vector $\bar{u}$ any time it is initialized. This vector is randomly selected using a Gaussian distribution with mean $\bar{\mu}$ and a positive definite covariance matrix Σ. For this state we want new estimates $\hat{\mu}$ and $\hat{\Sigma}$, that will increase the likelihood $P^f_F(\text{Root},T+1)$ for the given vector sequence $\bar{y}_1..\bar{y}_T$.

For each of the vectors $\bar{y}_1..\bar{y}_T$ it is possible to calculate an estimate for the probability that $\bar{y}_t$ is produced by S. We define w_t as the probability that S is active at time t, given that the Root state is initiated at t=1, finished at t=T+1, and produces $\bar{y}_1..\bar{y}_T$. This can be expressed in both already known forward and backward probabilities, that were used in the Recursive Backward Training algorithm:

$$w_t = \frac{P^f_I(S,t)\cdot P^b_I(S,t)}{P^f_F(\text{Root},T+1)} = \frac{P^f_F(S,t+1)\cdot P^b_F(S,t+1)}{P^f_F(\text{Root},T+1)} \tag{5.23}$$

The expected number of times that S was active can be expressed as:

$$\tau = \sum_{t=1}^{T} w_t \tag{5.24}$$

A heuristic way of estimating a new mean vector, is to consider $\bar{y}_1..\bar{y}_T$ as a set of observations and $w_1..w_T$ as a set of weight-factors. As an estimate for the mean vector the weighted average of the observations is used:

$$\hat{\mu} = \frac{\sum_{t=1}^{T} w_t\cdot\bar{y}_t}{\sum_{t=1}^{T} w_t} = \frac{1}{\tau}\cdot\sum_{t=1}^{T} w_t\cdot\bar{y}_t \tag{5.25}$$

For the covariance matrix also the standard estimation formula can be used, with w_t as weight factors (* denotes transpose)

$$\hat{\Sigma} = \frac{\sum_{t=1}^{T} w_t \cdot (\bar{y}_t - \hat{\mu}) \cdot (\bar{y}_t - \hat{\mu})^*}{\sum_{t=1}^{T} w_t} = \frac{1}{\tau} \cdot \left(\sum_{t=1}^{T} w_t \cdot \bar{y}_t \cdot \bar{y}_t^* \right) - \hat{\mu} \cdot \hat{\mu}^* \qquad (5.26)$$

These estimation formulas contain terms that are a sum of individual contributions at time t=1..T. These terms are useful as intermediate variables, because they can be updated like τ_{ij}, τ_{Ij} and τ_{iF} in the Backward Training algorithm. We define:

$$\bar{m} = \sum_{t=1}^{T} w_t \cdot \bar{y}_t = \begin{pmatrix} m_1 \\ \vdots \\ m_N \end{pmatrix}; \qquad V = \sum_{t=1}^{T} w_t \cdot \bar{y}_t \cdot \bar{y}_t^* = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1N} \\ v_{21} & v_{22} & \cdots & v_{2N} \\ \vdots & \vdots & & \vdots \\ v_{N1} & v_{N2} & \cdots & v_{NN} \end{pmatrix} \qquad (5.27)$$

The Backward Training algorithm for such an elementary state can be formulated as follows. Let τ^t, $\bar{m}^t$ and V^t denote partial summations for τ, $\bar{m}$ and V, with the summation is performed from t..T. We then have by (5.25) to (5.27):

Backward Training for elementary state with Gaussian distribution

```
Given: $\hat{P}^b_F(S,t+1)$
calculate:   $\hat{P}^b(S,t)$ and $\hat{P}^b_I(S,t)$
--------------------------------------------------
1. calculate $w_t$:   $w_t = \frac{P^f_F(S,t+1) \cdot P^b_F(S,t+1)}{P^f_F(Root,T+1)}$
2. Update $\tau$:     $\tau^t = \tau^{t+1} + w_t$
3. Update $\bar{m}$:  $\bar{m}^t = \bar{m}^{t+1} + w_t \cdot \bar{y}_t$
4. Update V:          $V^t = V^{t+1} + w_t \cdot \bar{y}_t \cdot \bar{y}_t^*$
5. $P^b_I(S,t) = P^b(S,t) = P^b_F(S,t+1) \cdot P_E(S,\bar{y}_t)$
```

This training is included in the Recursive Backward Training Algorithm for the elementary states. In Appendix A, it is proved that this algorithm supplies the Maximum Likelihood estimate for the elementary state coefficients.

If the training is performed while τ, $\overline{m}$ and V are shared between multiple elementary states, the contributions of individual states are added together in the algorithms. When training is performed with more than one training sequence, the individual contributions are also added. If all sequences and all states are processed, τ, $\overline{m}$ and V for state S collect the sum of the contributions from all training sequences. From these the new mean vector and covariance matrix can be calculated:

$$\boxed{\hat{\mu} = \frac{\overline{m}}{\tau}; \qquad \hat{\Sigma} = \frac{V}{\tau} - \hat{\mu}\cdot\hat{\mu}^*;} \tag{5.28}$$

Note that because V and Σ are symmetrical, only half of their elements have to be calculated and stored.

It is not so easy to employ matrix Σ directly in the calculation of $P_E(S,t)$ because a matrix inversion would be involved. To store the inverse of the matrix is neither suitable, because a matrix inversion has to be performed after each re-estimation. It is much more practical to use a lower triangular matrix L, defined as:

$$L\cdot L^* = \Sigma; \qquad L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ \vdots & l_{22} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{N1} & \cdots & \cdots & l_{NN} \end{pmatrix} \tag{5.29}$$

This matrix can be seen as a transformation from a random variable $\overline{x}$ with a unit covariance matrix into the random variable $(\overline{u}-\overline{\mu})$ with covariance matrix Σ:

$$\overline{u}-\overline{\mu} = L\,\overline{x}; \quad \text{or } \overline{x} = L^{-1}\,(\overline{u}-\overline{\mu}); \tag{5.30}$$

Because L is triangular, it is calculated from Σ by Cholesky decomposition. After that, $\overline{x}$ and $|L|$ are easily calculated by:

$$x_i = \frac{u_i - \mu_i - \sum_{j=1}^{i-1} l_{ij}\cdot x_j}{l_{ii}} \qquad \{i=1..N\} \tag{5.31}$$

$$|L| = \prod_{i=1}^{N} l_{ii} \tag{5.32}$$

Using (5.31) and (5.32), an easier way to calculate $P_E(S,\bar{u})$ is first to transform $\bar{u}$ in an intermediate random variable $\bar{x}$. Then $P_E(S,\bar{u})$ is calculated by:

$$P_E(S,\bar{u}) = \frac{|L|^{-1}}{(2\pi)^{N/2}} \cdot \exp\left(-\frac{\bar{x}^* \cdot \bar{x}}{2}\right) \tag{5.33}$$

The transformation matrix L and the mean vector $\bar{\mu}$ are treated as model parameters, like the matrix A in non-elementary states. The scalar τ, the vector $\bar{m}$ and the matrix V are updated like the variables τ_{ij}, τ_{Ij} and τ_{iF} in non-elementary states. Therefore, the training algorithm for elementary states can be included very easily in the Recursive Backward Training algorithm derived in the previous sections.

Normally, the variables τ, $\bar{m}$ and V are initialized at zero, just as the coefficients τ_{Ij}, τ_{ij} and τ_{iF} could be initialized to be zero. As already described, this will cause problems when the training data do not contain all possible transitions. In case of the A-matrix this can solved by initializing $\tau_{ij} = a_{ij}$, as already explained. A similar procedure also holds for elementary states. One could pretend that all elementary states have already produced a symbol once, fulfilling exactly the given parameters.

Thus, an alternative initialization could be (being the reverse of (5.28) with τ=1):

$$\tau = 1; \qquad \bar{m} = \bar{\mu}; \qquad V = \Sigma + \bar{\mu}\cdot\bar{\mu}^*$$

If no symbol from the concerned elementary state is present in the training data, τ, $\bar{m}$ and V are not updated. Using (5.28) now results in $\hat{\mu} = \bar{\mu}$ and $\hat{\Sigma} = \Sigma$. This is the best possible estimation in these circumstances. If the elementary state is used in the training data many times ($\tau \gg 1$) this initialization has very little influence.

5.6 Efficient calculation of various statistical measures

A measure for the complexity of a finite state machine is the entropy as defined by Shannon. A particular method for calculating this entropy has been derived by Sondhi and Levinson. In the finite state machine that they use, the transition probabilities are undefined, making it impossible to calculate the language entropy. However, two measures can be calculated: the equal probability entropy and the maximum entropy.

The method described by Sondhi is only valid when the used language is finite. But for infinite languages a useful approximation can be formulated by halting after a sufficient number of iterations.

RMM is in fact a finite state machine with given transition probabilities, so it is possible to calculate the entropy. In general the RMM contains self-loops, which means that the language is infinite. In the following a new recursive method will be developed for calculating the entropy and the average duration of any state in the RMM.

First some assumptions must be made. For calculating the entropy it is necessary that all elementary states are uncorrelated. In general they are not, because many elementary states produce the same symbol. Let us suppose however, that we have a perfect speech recognizer, which is able to construct from every possible symbol sequence the corresponding unique state sequence without any mistake. Thus, the information contained in the symbol sequence (= the entropy) is equal to the information in the state sequence. The calculated entropy is only valid when the speech recognizer is perfect.

If elementary states in the RMM have correlation, the state sequence cannot be uniquely reconstructed. This means that the actual entropy will be lower than the calculated entropy, indicating the loss of information in the speech production/recognition process.

Suppose we have an RMM with given transition probabilities. This RMM produces a continuous random symbol sequence, starting at $t=-\infty$. Every time the Root state has finished, it is initiated again. Equivalently, the final node is again connected with the initial node. This model is continuously producing symbols. Now we can define, for any state S:

$P(S)$: probability that the RMM is in state S
$P_F(S)$: probability that the RMM is leaving state S
$P_I(S)$: probability that the RMM is initiating state S

These probabilities have become independent of time t, because the RMM is producing symbols continuously. Any state which is initiated will be finished some time later. This means that for any state S (not only for the root) the following relation holds:

$$P_F(S) = P_I(S) \qquad (5.34)$$

In order to calculate the entropy let us first consider the definition. Suppose from some node in a finite state machine a transition is made. There are N possible paths each with a probability a_j ($1 \leq j \leq N$). Then the entropy involved in this transition is:

$$H(a) = -\sum_j a_j \cdot \log_2(a_j) \qquad (5.35)$$

A large model contains many of these transitions, and we do not know which transition is made at time t. In this case we multiply the entropy of each possible transition with its relative probability of occurrence.

In a large RMM, the same principle can be applied. The entropy of any state S is an information measure, describing the information concerned with all possible transitions within that state. Many transitions are possible at the same time, which partly belong to lower level states. If we split the entropy in multiple terms, partly from transition occurring inside lower level states and partly from transitions inside state S, the result yields formula (5.36). Each term in it has to be multiplied with the relative likelihood for each transition, because they are not equally likely. These relative likelihood factors are yet unknown, but we will describe a procedure to calculate them.

$$H(S) = \sum_i \frac{P_I(S_i)}{P_I(S)} \cdot H(S_i) \quad - \quad \sum_j a_{Ij} \cdot \log_2(a_{Ij})$$
$$- \sum_i \frac{P_I(S_i)}{P_I(S)} \cdot \left(\sum_j a_{ij} \cdot \log_2(a_{ij}) + a_{iF} \cdot \log_2(a_{iF}) \right) \qquad (5.36)$$

The first term is the weighted entropy of the child states. The second is the entropy involved in the first transition into one of the child states. The last term is the entropy involved in transitions between child states and final transitions. Now we only have left to calculate $P_I(S_i)/P_I(S)$ for every state.

for this purpose define: $u_i = \dfrac{P_I(S_i)}{P_I(S)}$ (5.37)

With this definition the entropy per state becomes:

$$H(S) = \sum_i u_i \cdot \left(H(S_i) - \sum_j a_{ij} \cdot \log_2(a_{ij}) - a_{iF} \cdot \log_2(a_{iF}) \right)$$
$$- \sum_j a_{Ij} \cdot \log_2(a_{Ij}) \qquad (5.38)$$

The relation between $P_I(S_i)$, $P_F(S_i)$ and $P_I(S)$ at any time t is:

$$\begin{pmatrix} P_I(S_1) \\ P_I(S_2) \\ \vdots \\ P_I(S_N) \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{N1} \\ a_{12} & a_{22} & \cdots & a_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1N} & a_{2N} & \cdots & a_{NN} \end{pmatrix} \cdot \begin{pmatrix} P_F(S_1) \\ P_F(S_2) \\ \vdots \\ P_F(S_N) \end{pmatrix} + \begin{pmatrix} a_{I1} \\ a_{I2} \\ \vdots \\ a_{IN} \end{pmatrix} \cdot P_I(S) \qquad (5.39)$$

Dividing by $P_I(S)$ and using (5.34) and (5.37), this can be reformulated as:

$$\begin{pmatrix} a_{11}-1 & a_{21} & \cdots & a_{N1} \\ a_{12} & a_{22}-1 & \cdots & a_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1N} & a_{2N} & \cdots & a_{NN}-1 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = - \begin{pmatrix} a_{I1} \\ a_{I2} \\ \vdots \\ a_{IN} \end{pmatrix} \qquad (5.40)$$

These are N equations with N unknowns, therefore, in general u_i can be solved. The result can be used in the entropy formula (5.38).

Suppose we let the RMM produce T symbols, then any state S will produce on average T·P(S) symbols. In the same time period state S will start a new sequence $T \cdot P_I(S)$ times. Hence, the average state duration L(S) of state S can be defined as:

$$L(S) = P(S) \;/\; P_I(S) \qquad (5.41)$$

To calculate the average duration of state S we use:

$$P(S) = \sum_i P(S_i) \tag{5.42}$$

Using the definitions of L(S) and u_i we can formulate this as:

$$L(S) = \sum_i u_i \cdot L(S_i) \tag{5.43}$$

If the elementary states are uncorrelated, they contain no information and they produce one symbol each. For all elementary states S_E we then have:

$$H(S_E) = 0 \; ; \quad L(S_E) = 1; \tag{5.44}$$

In this way we derived a recursive method to calculate the entropy and the average duration of any state inside a RMM. For each state S, first $u_1..u_N$ are calculated by solving (5.40). The entropy and average state duration then can be calculated from the entropy and average state duration of its child states, using (5.38) and (5.43).

Solving equation (5.40) in general requires $O(N^3)$ multiplications. For many practical languages, however, most coefficients are zero. For instance, if only forward- and self-transitions ($a_{ij}=0$ for i>j) exist, then equation (5.40) can be solved explicitly by:

$$u_j = \frac{a_{1j} + \sum_{i=1}^{j-1} a_{ij} \cdot u_i}{1 - a_{jj}} \qquad \{\, j = 1..n \,\} \tag{5.45}$$

More calculations can be saved employing State Sharing, because the entropy and average duration of any state lower in the hierarchy are not dependent on higher level states. If any state model is used more than once in the RMM, the entropy and the average state duration only have to be calculated once.

6. SCALING AND STATE PRUNING IN RMM

The algorithms described in the previous chapter correctly perform training and speech recognition. Still, two main problems will occur when these methods are used in a practical speech recognition systems with a large vocabulary. Solutions to these problems are provided in this chapter.

As a first problem, the derived Recursive algorithms suffer from numerical underflow in exactly the same way as the algorithms normally used in HMM. The most common approach in HMM is to represent all probabilities by their logarithms. In this way, all multiplications change to additions, and for implementing additions, a procedure had already been developed. Another less common solution is to use appropriate scaling. In this chapter an alternative scaling method is introduced that solves the numerical underflow problem, and has several additional advantages as well.

A second problem in developing speech recognition systems with a large vocabulary is that the used model can become very large. State sharing already saves a lot of work and complexity, but in the Recursive algorithms still too many variables have to be processed. The scaling, introduced in this chapter, allows for a new state pruning approach. The Recursive Forward Backward and Viterbi algorithm are re-formulated in terms of the scaled variables. Normally, the number of variables to be processed grows linear with the vocabulary size. State sharing and pruning allow to reduce this size, by using the redundancy in the applied language model.

6.1 The Principles of Scaling and Pruning

A common problem in HMM is numerical underflow. During each time step all probabilities are multiplied by transition probabilities (for non-elementary states) or output probabilities (for elementary states), which are all smaller than 1, and thus eventually result in underflow. On any computer, no matter what floating point format is used, for training sequences being long enough, all probabilities will approach zero until they are actually rounded to zero.

Two solutions for this problem have been proposed by various authors. One is to use the logarithms of all variables. This changes all multiplications to additions. For adding numbers that are represented logarithmically an efficient method has been described by Kingsbury and Rayner [26]. A short summary of this method can be found in Holmes [20] in Section 8.11 (page 149/150). The disadvantage of this method is the loss of accuracy, but this does not seem to be a severe problem in actual practice.

Another way to solve this problem is to use scaling. As soon as probabilities become lower than some threshold, a scale factor is used to bring them back into range. All training and recognition algorithms must be adapted for the use of this scale factor. The scale factors used during the forward algorithm, for instance, can be stored in memory, thereby enabling correction during the backward algorithm.

One of the advantages of RMM is that the total library is represented by a single state, and the final probability of this state, P^f_F(Root,T+1), is used already as scaling factor in the update of τ_{ij}. If, during the forward algorithm, all probabilities are multiplied by a constant, then P^f_F(Root,T+1) is also multiplied by the same constant. Hence, the update of τ_{ij} does not change at all. In case of recognition using the Recursive Viterbi algorithm we are only interested in the identity of the best path. Multiplication by a constant factor does not influence the result of this algorithm. This observation suggests that it should be possible to introduce scale factors in the algorithm, such that the update formulas become insensitive for that scaling but for which the product of all scale factors is not needed in the algorithm. This algorithm called the Scaled Recursive Forward-Backward Training Algorithm is derived in the following section. In the remaining sections the other algorithms of Chapter 5 are re-formulated in the same way.

In addition it will be shown that scaling can be used for state pruning (ignoring states with too low probabilities) resulting in more efficiency in the computations and reduction in storage requirements.

6.2 The Scaled Recursive Forward Backward algorithm

Consider the defined probabilities for any state S and its child states S_i. According to (5.7) we can immediately conclude that:

$$P^f(S_i,t) \leq P^f(S,t) \tag{6.1}$$

If we replace all probabilities by scaled probabilities, dividing them by $P^f(S,t)$, the previous Recursive Forward and Backward Training algorithm can be re-formulated in terms of the new variables. This does not hold when $P^f(S,t)$ is zero, but then the solution to the problem is trivial: $P^f(S_i,t)=0$;

If $P^f(S,t) \neq 0$, the following new variables are defined:

$$\boxed{\tilde{P}^f(S_i,t)=\frac{P^f(S_i,t)}{P^f(S,t)};\ \tilde{P}^f_I(S_i,t)=\frac{P^f_I(S_i,t)}{P^f(S,t)};\ \tilde{P}^f_F(S_i,t+1)=\frac{P^f_F(S_i,t+1)}{P^f(S,t)}} \tag{6.2}$$

This definition assures that:

$$\sum_i \tilde{P}^f(S_i,t) = 1 \quad \Rightarrow \quad \tilde{P}^f(S_i,t) \leq 1 \tag{6.3}$$

This definition is not valid for the Root state, because the Root has no parent state. Instead we use the state probability of the Root state as its own scale factor, yielding:

$$\boxed{\tilde{P}^f(\text{Root},t)=1;\ \tilde{P}^f_I(\text{Root},t)=\frac{P^f_I(\text{Root},t)}{P^f(\text{Root},t)};\ \tilde{P}^f_F(\text{Root},t+1)=\frac{P^f_F(\text{Root},t+1)}{P^f(\text{Root},t)}} \tag{6.4}$$

A scaled state probability of one of the sub-states S_i of S approaching zero, implies the existence of another sub-state having a higher state probability. Hence, when S_i is not considered any more in the calculation, hardly any calculation error will be made. This observation forms the basis of the state pruning. A threshold ε is selected ($\varepsilon \ll 1$), and all scaled probabilities smaller than ε are set to zero during the algorithm. If ε is chosen small enough, the error made by this approximation is negligible. However, the amount of work to be performed is drastically reduced, improving the efficiency by an order of magnitude.

In order to be able to re-formulate the previous algorithms we also need a new set of variables, that are scaled according to time t-1. Three different definitions are necessary for three different situations:

If $\tilde{P}^f(S,t\text{-}1) > \varepsilon$ (so automatically $P^f(S,t\text{-}1) \neq 0$) we define:

$$\boxed{\hat{P}^f(S_i,t)=\frac{P^f(S_i,t)}{P^f(S,t\text{-}1)};\ \hat{P}^f_I(S_i,t)=\frac{P^f_I(S_i,t)}{P^f(S,t\text{-}1)};\ \hat{P}^f_F(S_i,t+1)=\frac{P^f_F(S_i,t+1)}{P^f(S,t\text{-}1)}} \tag{6.5}$$

If $\tilde{P}^f(S,t\text{-}1) \leq \varepsilon$ and $\tilde{P}^f_I(S,t) > \varepsilon$:

$$\boxed{\hat{P}^f(S_i,t)=\frac{P^f(S_i,t)}{P^f_I(S,t)};\ \hat{P}^f_I(S_i,t)=\frac{P^f_I(S_i,t)}{P^f_I(S,t)};\ \hat{P}^f_F(S_i,t+1)=\frac{P^f_F(S_i,t+1)}{P^f_I(S,t)}} \tag{6.6}$$

Otherwise:

$$\boxed{\hat{P}^f(S_i,t) = \hat{P}^f_I(S_i,t) = \hat{P}^f_F(S_i,t+1) = 0} \tag{6.7}$$

These three situations have the following interpretation:

1) State S is not pruned at time t-1.
2) State S is pruned at time t-1, but will be considered again at time t.
3) State S is pruned at time t-1, and also at time t.

With these new variables, regardless whether in situation 1, 2 or 3, the **elementary** state relations become:

$$\boxed{\hat{P}^f_F(S,t+1) = \hat{P}^f(S,t) = \hat{P}^f_I(S,t) \cdot P_E(S,Y_t)} \tag{6.8}$$

For **non-elementary** states in all three situations the Recursive Forward Backward algorithms must be re-formulated in terms of the new variables. This adaptation leads to the following set of algorithms:

The Scaled Recursive Forward Algorithm (situation 1)

```
Given:      $\hat{P}^f_I(S,t)$, $\tilde{P}^f_F(S_j,t)$ for every sub-state $S_j$
calculate:  $\hat{P}^f(S,t)$ and $\hat{P}^f_F(S,t+1)$
--------------------------------------------------------------------------
1. for every child $S_j$:
    $\hat{P}^f_I(S_j,t) = a_{Ij}\cdot\frac{\hat{P}^f_I(S,t)}{\tilde{P}^f(S,t-1)} + \sum_i a_{ij}\cdot\tilde{P}^f_F(S_i,t)$
    calculate $\hat{P}^f(S_j,t)$, $\hat{P}^f_F(S_j,t+1)$ (Recursive)
2. finally: $\hat{P}^f(S,t) = \tilde{P}^f(S,t-1)\cdot\sum_i \hat{P}^f(S_i,t)$
            $\hat{P}^f_F(S,t+1) = \tilde{P}^f(S,t-1)\cdot\sum_i a_{iF}\cdot\hat{P}^f_F(S_i,t+1)$
```

These formulas can be checked easily by substituting the definitions of the scaled variables. $\tilde{P}^f(S,t)$, $\tilde{P}^f_I(S,t)$ and $\tilde{P}^f_F(S,t)$ have to be calculated as well, but this is not possible yet because for their calculation the state probability of the parent of S is needed. Yet, for the sub-states this is possible:

$$\tilde{P}^f(S_j,t) = \frac{\hat{P}^f(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}; \qquad \tilde{P}^f_I(S_j,t) = \frac{\hat{P}^f_I(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}; \tag{6.9}$$

$$\tilde{P}^f_F(S_j,t+1) = \frac{\hat{P}^f_F(S_j,t+1)}{\sum_i \hat{P}^f(S_i,t)};$$

These relations can also be checked by using the definitions. The complete Scaled Recursive Forward Algorithm as used in situation 1) then becomes:

Scaled Recursive Forward algorithm (situation 1) with normalization

```
Given:      $\hat{P}^f_I(S,t)$, $\tilde{P}^f_F(S_j,t)$ for every sub-state $S_j$
calculate:  $\hat{P}^f(S,t)$ and $\hat{P}^f_F(S,t+1)$
---------------------------------------------------------------
1. for every child $S_j$:
     $\hat{P}^f_I(S_j,t) = a_{Ij}\cdot\frac{\hat{P}^f_I(S,t)}{\tilde{P}^f(S,t-1)} + \sum_i a_{ij}\cdot\tilde{P}^f_F(S_i,t)$
     calculate $\hat{P}^f(S_j,t)$, $\hat{P}^f_F(S_j,t+1)$ (Recursive)
2. re-normalize:
     $\tilde{P}^f(S_j,t) = \frac{\hat{P}^f(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}$;
     $\tilde{P}^f_I(S_j,t) = \frac{\hat{P}^f_I(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}$;  $\tilde{P}^f_F(S_j,t+1) = \frac{\hat{P}^f_F(S_j,t+1)}{\sum_i \hat{P}^f(S_i,t)}$;
3. finally: $\hat{P}^f(S,t) = \tilde{P}^f(S,t-1)\cdot\sum_i \hat{P}^f(S_i,t)$
     $\hat{P}^f_F(S,t+1) = \tilde{P}^f(S,t-1)\cdot\sum_i a_{iF}\cdot\hat{P}^f_F(S_i,t+1)$
```

In situation 2) a different scale factor is used. Because at time t state S was pruned, all previous probabilities are also supposed to be zero. From step 1. only the first term is non-zero. The result is:

Scaled Recursive Forward algorithm in situation 2)

```
Given:      $\hat{P}^f_I(S,t)$
calculate:  $\hat{P}^f(S,t)$ and $\hat{P}^f_F(S,t+1)$
---------------------------------------------------------------
1. for every child $S_j$:      $\hat{P}^f_I(S_j,t) = a_{Ij}$;
     calculate $\hat{P}^f(S_j,t)$, $\hat{P}^f_F(S_j,t+1)$ (Recursive)
2. re-normalize:
     $\tilde{P}^f(S_j,t) = \frac{\hat{P}^f(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}$;
     $\tilde{P}^f_I(S_j,t) = \frac{\hat{P}^f_I(S_j,t)}{\sum_i \hat{P}^f(S_i,t)}$;  $\tilde{P}^f_F(S_j,t+1) = \frac{\hat{P}^f_F(S_j,t+1)}{\sum_i \hat{P}^f(S_i,t)}$
3. Finally: $\hat{P}^f(S,t) = \hat{P}^f_I(S,t)\cdot\sum_i \hat{P}^f(S_i,t)$
     $\hat{P}^f_F(S,t+1) = \hat{P}^f_I(S,t)\cdot\sum_i a_{iF}\cdot\hat{P}^f_F(S_i,t+1)$
```

In situation 3) all probabilities are assumed to be zero. Thus, the solution becomes:

$$\boxed{\tilde{P}^f_F(S,t+1) = \tilde{P}^f(S,t) = \tilde{P}^f_I(S,t) = 0} \tag{6.10}$$

From the above formulas all the scaled forward probabilities can be calculated.

In the same way the Recursive Backward algorithm has to be reformulated, using new Scaled Backward probabilities that will be defined in the sequel. In addition, the relations for updating τ_{Ij}, τ_{ij} and τ_{iF} need to be changed in order to be able to apply the new variables.

If $P^b(S,t) \neq 0$, we define:

$$\boxed{\tilde{P}^b(S_i,t)=\frac{P^b(S_i,t)}{P^b(S,t)}\ ;\ \tilde{P}^b_I(S_i,t)=\frac{P^b_I(S_i,t)}{P^b(S,t)}\ ;\ \tilde{P}^b_F(S_i,t+1)=\frac{P^b_F(S_i,t+1)}{P^b(S,t)}} \tag{6.11}$$

If $\tilde{P}^b(S,t+1) > \varepsilon$ (so automatically $P^b(S,t+1) \neq 0$) we define:

$$\boxed{\hat{P}^b(S_i,t)=\frac{P^b(S_i,t)}{P^b(S,t+1)};\ \hat{P}^b_I(S_i,t)=\frac{P^b_I(S_i,t)}{P^b(S,t+1)};\ \hat{P}^b_F(S_i,t+1)=\frac{P^b_F(S_i,t+1)}{P^b(S,t+1)}} \tag{6.12}$$

If $\tilde{P}^b(S,t+1) \leq \varepsilon$ and $\tilde{P}^b_F(S,t+1) > \varepsilon$:

$$\boxed{\hat{P}^b(S_i,t)=\frac{P^b(S_i,t)}{P^b_F(S,t+1)};\ \hat{P}^b_I(S_i,t)=\frac{P^b_I(S_i,t)}{P^b_F(S,t+1)};\ \hat{P}^b_F(S_i,t+1)=\frac{P^b_F(S_i,t+1)}{P^b_F(S,t+1)}} \tag{6.13}$$

Otherwise:

$$\boxed{\hat{P}^b(S_i,t) = \hat{P}^b_I(S_i,t) = \hat{P}^b_F(S_i,t+1) = 0} \tag{6.14}$$

For the Root state again two different situations have to be distinguished, because this state is never pruned. We define for t=T:

$$\boxed{\hat{P}^b(\mathrm{Root},T)=\frac{P^b(\mathrm{Root},T)}{P^b_F(\mathrm{Root},T+1)};\quad \begin{array}{l} P^b_I(\mathrm{Root},T)=\dfrac{P^b_I(\mathrm{Root},T)}{P^b_F(\mathrm{Root},T+1)} \\[2ex] \hat{P}^b_F(\mathrm{Root},T+1)=1 \end{array}} \tag{6.15}$$

and for t<T:

$$\boxed{\hat{P}^b(\text{Root},t)=\frac{P^b(\text{Root},t)}{P^b(\text{Root},t+1)}; \quad \begin{array}{l} \hat{P}^b_I(\text{Root},t) = \dfrac{P^b_I(\text{Root},t)}{P^b(\text{Root},t+1)} \\ \hat{P}^b_F(\text{Root},t+1)=\dfrac{P^b_F(\text{Root},t+1)}{P^b(\text{Root},t+1)} \end{array}} \tag{6.16}$$

For updating τ_{ij}, τ_{1j} and τ_{iF} a new scale factor is defined:

if $\tilde{P}^b(S,t+1) > \varepsilon$: (situation 1)

$$C_t(S_i) = \frac{P^f(S,t) \cdot P^b(S,t+1)}{P^f_F(\text{Root},T+1)} \tag{6.17}$$

Otherwise (situation 2):

$$C_t(S_i) = \frac{P^f(S,t) \cdot P^b_F(S,t+1)}{P^f_F(\text{Root},T+1)} \tag{6.18}$$

This scale factor for the Root state has to be initialized at t=T and updated for other times T. This can be done with the following relations. Note that at t=T we have situation 2), otherwise we have situation 1):

$$C_T(\text{Root}) = \frac{P^f(\text{Root},T)}{P^f_F(\text{Root},T+1)} \cdot P^b_F(\text{Root},T+1) = \frac{1}{\tilde{P}^f_F(\text{Root},T+1)} \tag{6.19}$$

$$C_t(\text{Root}) = C_{t+1}(\text{Root}) \cdot \frac{\hat{P}^b(\text{Root},t)}{\hat{P}^f(\text{Root},t+1)} \qquad \{\ t<T\ \} \tag{6.20}$$

Using all previous definitions, the Scaled Recursive Backward Training algorithm can now be formulated. Its correctness can again be checked by substitution of the definition of all variables (situation 1) in the unscaled Recursive Backward Training algorithm.

The **elementary** state backward relations become:

$$\boxed{\hat{P}^b_I(S,t) = \hat{P}^b(S,t) = \hat{P}^b_F(S,t+1) \cdot P_E(S,Y_t)} \tag{6.21}$$

For **non-elementary** states the algorithm becomes :

Scaled Recursive Backward Training algorithm (situation 1)

```
Given: $\hat{P}^b_F(S,t+1)$, $\tilde{P}^b_I(S_i,t+1)$ for every sub-state $S_i$
calculate:   $\hat{P}^b(S,t)$ and $\hat{P}^b_I(S,t)$
--------------------------------------------------------------------------------
1. new scale factor for child states and update $\tau_{ij}$:
   $C_t(S_i) = C_t(S)\cdot\tilde{P}^f(S,t)\cdot\tilde{P}^b(S,t+1)$
   $\tau^t_{ij} = \tau^{t+1}_{ij} + C_t(S_i)\cdot\tilde{P}^f_F(S_i,t+1)\cdot a_{ij}\cdot\tilde{P}^b_I(S_j,t+1)$
2. for every child $S_i$:
   $\hat{P}^b_F(S_i,t+1) = a_{iF}\cdot\frac{\hat{P}^b_F(S,t+1)}{\tilde{P}^b(S,t+1)} + \sum_j a_{ij}\cdot\tilde{P}^b_I(S_j,t+1)$
   calculate $\hat{P}^b(S_i,t)$, $\hat{P}^b_I(S_i,t)$ (Recursive)
3. Update $\tau_{Ij}$ and $\tau_{iF}$:
   $\tau^t_{Ij} = \tau^{t+1}_{Ij} + C_t(S)\cdot\tilde{P}^f_I(S,t)\cdot a_{Ij}\cdot\hat{P}^b_I(S_j,t)\cdot\tilde{P}^b(S,t+1)$
   $\tau^t_{iF} = \tau^{t+1}_{iF} + C_t(S)\cdot\tilde{P}^f_F(S_i,t+1)\cdot\tilde{P}^f(S,t)\cdot a_{iF}\cdot\hat{P}^b_F(S,t+1)$
4. re-normalize:   $\tilde{P}^b(S_i,t) = \frac{\hat{P}^b(S_i,t)}{\sum_j \hat{P}^b(S_j,t)}$;
   $\tilde{P}^b_I(S_i,t) = \frac{\hat{P}^b_I(S_i,t)}{\sum_j \hat{P}^b(S_j,t)}$;   $\tilde{P}^b_F(S_i,t+1) = \frac{\hat{P}^b_F(S_i,t+1)}{\sum_j \hat{P}^b(S_j,t)}$;
5. Finally:  $\hat{P}^b(S,t) = \tilde{P}^b(S,t+1)\cdot\sum_j \hat{P}^b(S_j,t)$
             $\hat{P}^b_I(S,t) = \tilde{P}^b(S,t+1)\cdot\sum_j a_{Ij}\cdot\hat{P}^b_I(S_j,t)$
```

In situation 2) all previous probabilities are supposed to be zero. Therefore, all expressions that contain these zero probabilities can be simplified. It turns out that the update of τ_{ij} will always be zero, but not for τ_{Ij} and τ_{iF}. The algorithm becomes:

Scaled Recursive Backward Training algorithm (situation 2)

```
Given: $\hat{P}^b_F(S,t+1)$
calculate:   $\hat{P}^b(S,t)$ and $\hat{P}^b_I(S,t)$
--------------------------------------------------------------
1. new scale factor for child states:
     $C_t(S_i) = C_t(S)\cdot\tilde{P}^f(S,t)\cdot\hat{P}^b_F(S,t+1)$
2. for every child $S_i$:   $\hat{P}^b_F(S_i,t+1) = a_{iF}$

     calculate $\hat{P}^b(S_i,t)$, $\hat{P}^b_I(S_i,t)$ (Recursive)
3. Update $\tau_{Ij}$ and $\tau_{iF}$:
     $\tau^t_{Ij} = \tau^{t+1}_{Ij} + C_t(S)\cdot\tilde{P}^f_I(S,t)\cdot a_{Ij}\cdot\hat{P}^b_I(S_j,t)\cdot\hat{P}^b_F(S,t+1)$
     $\tau^t_{iF} = \tau^{t+1}_{iF} + C_t(S)\cdot\tilde{P}^f_F(S_i,t+1)\cdot\tilde{P}^f(S,t)\cdot a_{iF}\cdot\hat{P}^b_F(S,t+1)$
4. re-normalize:
     $\tilde{P}^b(S_i,t) = \frac{\hat{P}^b(S_i,t)}{\sum_j \hat{P}^b(S_j,t)}$;
     $\tilde{P}^b_I(S_i,t) = \frac{\hat{P}^b_I(S_i,t)}{\sum_j \hat{P}^b(S_j,t)}$;   $\tilde{P}^b_F(S_i,t+1) = \frac{\hat{P}^b_F(S_i,t+1)}{\sum_j \hat{P}^b(S_j,t)}$;
5. Finally:  $\hat{P}^b(S,t) = \hat{P}^b_F(S,t+1)\cdot\sum_j \hat{P}^b(S_j,t)$
             $\hat{P}^b_I(S,t) = \hat{P}^b_F(S,t+1)\cdot\sum_j a_{Ij}\cdot\hat{P}^b_I(S_j,t)$
```

Using the state pruning of the Recursive Forward Algorithm, a large number of calculations can be saved when applying the Recursive Backward Training algorithm. Let us suppose that $\tilde{P}^f(S,t) \le \varepsilon$, then in the forward algorithm $\tilde{P}^f(S,t)$, $\tilde{P}^f_I(S,t)$ and $\tilde{P}^f_F(S,t+1)$ are set to zero. In the backward algorithm this means that all updates of τ_{Ij}, τ_{ij} and τ_{iF} will be zero, independent of the values of the backward probabilities. Hence, it is of no use to calculate these backward probabilities, which are set to zero, in order to force this state to be pruned in the Backward Training algorithm. The values calculated for the backward probabilities will be wrong, but because we are only interested in τ_{Ij}, τ_{ij} and τ_{iF} this does not matter.

With this algorithm we can calculate τ_{Ij}, τ_{ij} and τ_{iF} for all states in the RMM. Only local variables are used in the re-estimation formulas. State sharing and multiple training sequences can be handled in the same way as the unscaled version of the Recursive Forward Backward Training algorithm. The most important advantage of the scaling is that state pruning can be used. Later on in this thesis we will show that this state pruning can reduce the number of calculations to a great extent.

6.3 The Scaled Recursive Viterbi algorithm

The state pruning as used in the Recursive Forward algorithm also can now be used in the Recursive Viterbi algorithm. First we define the scaled viterbi variables as follows, (analogous to the scaled forward variables):

If $P^V(S,t) \neq 0$:

$$\boxed{\tilde{P}^V(S_i,t)=\frac{P^V(S_i,t)}{P^V(S,t)};\ \tilde{P}^V_I(S_i,t)=\frac{P^V_I(S_i,t)}{P^V(S,t)};\ \tilde{P}^V_F(S_i,t+1)=\frac{P^V_F(S_i,t+1)}{P^V(S,t)}} \tag{6.22}$$

This definition assures that:

$$\max_i \tilde{P}^V(S_i,t) = 1 \quad \Rightarrow \quad \tilde{P}^V(S_i,t) \leq 1 \tag{6.23}$$

Only in case of the Root state this definition is not valid because the Root has no parent state. Instead we define:

$$\boxed{\tilde{P}^V(Root,t)=1;\ \tilde{P}^V_I(Root,t)=\frac{P^V_I(Root,t)}{P^V(Root,t)};\ \tilde{P}^V_F(Root,t+1)=\frac{P^V_F(Root,t+1)}{P^V(Root,t)}} \tag{6.24}$$

Also, if $\tilde{P}^V(S,t-1) > \varepsilon$ (so automatically $P^V(S,t-1)\neq 0$) we define:

$$\boxed{\hat{P}^V(S_i,t)=\frac{P^V(S_i,t)}{P^V(S,t-1)};\ P^V_I(S_i,t)=\frac{P^V_I(S_i,t)}{P^V(S,t-1)};\ \hat{P}^V_F(S_i,t+1)=\frac{P^V_F(S_i,t+1)}{P^V(S,t-1)}} \tag{6.25}$$

If $\tilde{P}^V(S,t-1) \leq \varepsilon$ and $\tilde{P}^V_I(S,t) > \varepsilon$:

$$\boxed{\hat{P}^V(S_i,t)=\frac{P^V(S_i,t)}{P^V_I(S,t)};\ \hat{P}^V_I(S_i,t)=\frac{P^V_I(S_i,t)}{P^V_I(S,t)};\ \hat{P}^V_F(S_i,t+1)=\frac{P^V_F(S_i,t+1)}{P^V_I(S,t)}} \tag{6.26}$$

Otherwise:

$$\boxed{\hat{P}^V(S_i,t) = \hat{P}^V_I(S_i,t) = \hat{P}^V_F(S_i,t+1) = 0} \tag{6.27}$$

With these new variables, regardless whether situation 1, 2 or 3 is concerned, the **elementary** state relations become:

$$\boxed{\hat{P}^V_F(S,t+1) = \hat{P}^V(S,t) = \hat{P}^V_I(S,t) \cdot P_E(S,Y_t)} \tag{6.28}$$

These definitions are similar to (6.2)..(6.8) for the forward probabilities. The Scaled Recursive Viterbi algorithm has the same structure as the Scaled Forward algorithm, except that all summations are replaced by the maximum-operator. With this similarity in mind, we can formulate the Scaled Recursive Viterbi algorithm for **non-elementary** states right away:

Scaled Recursive Viterbi algorithm (situation 1)

Given: $\hat{P}^V_I(S,t)$, $\tilde{P}^V_F(S_j,t)$ for every sub-state S_j

calculate: $\hat{P}^V(S,t)$ and $\hat{P}^V_F(S,t+1)$

1. for every child S_j:

$$\hat{P}^V_I(S_j,t) = \max\left(a_{Ij}\cdot\frac{\hat{P}^V_I(S,t)}{\tilde{P}^V(S,t-1)}\ ,\ \max_i\left(a_{ij}\cdot\tilde{P}^V_F(S_i,t)\right)\right)$$

calculate $\hat{P}^V(S_j,t)$, $\hat{P}^V_F(S_j,t+1)$ (Recursive)

2. re-normalize:

$$\tilde{P}^V(S_j,t) = \frac{\hat{P}^V(S_j,t)}{\max_i \hat{P}^V(S_i,t)};$$

$$\tilde{P}^V_I(S_j,t) = \frac{\hat{P}^V_I(S_j,t)}{\max_i \hat{P}^V(S_i,t)};\quad \tilde{P}^V_F(S_j,t+1) = \frac{\hat{P}^V_F(S_j,t+1)}{\max_i \hat{P}^V(S_i,t\ \ )};$$

3. finally:

$$\hat{P}^V(S,t) = \tilde{P}^V(S,t-1)\cdot\max_i\left(\hat{P}^V(S_i,t)\right)$$

$$\hat{P}^V_F(S,t+1) = \tilde{P}^V(S,t-1)\cdot\max_i\left(a_{iF}.\hat{P}^V_F(S_i,t+1)\right)$$

In situation 2) a different scale factor is used, like in the Scaled Recursive Forward algorithm:

Scaled Recursive Viterbi algorithm in situation 2)

```
Given:      $\hat{P}^V_I(S,t)$
calculate:  $\hat{P}^V(S,t)$ and $\hat{P}^V_F(S,t+1)$
  1. for every child $S_j$:   $\hat{P}^V_I(S_j,t) = a_{Ij}$;
       calculate $\hat{P}^V(S_j,t), \hat{P}^V_F(S_j,t+1)$ (Recursive)
  2. re-normalize:
       $\tilde{P}^V(S_j,t) = \frac{\hat{P}^V(S_j,t)}{\max_i \hat{P}^V(S_i,t)}$;
       $\tilde{P}^V_I(S_j,t) = \frac{\hat{P}^V_I(S_j,t)}{\max_i \hat{P}^V(S_i,t)}$; $\tilde{P}^V_F(S_j,t+1) = \frac{\hat{P}^V_F(S_j,t+1)}{\max_i \hat{P}^V(S_i,t)}$
  3. Finally:   $\hat{P}^V(S,t) = \hat{P}^V_I(S,t) \cdot \max_i \left( \hat{P}^V(S_i,t) \right)$
       $\hat{P}^V_F(S,t+1) = \hat{P}^V_I(S,t) \cdot \max_i \left( a_{iF} \cdot \hat{P}^V_F(S_i,t+1) \right)$
```

This algorithm enables us to calculate the probability of the best path at each time t. Just like in the unscaled version of this algorithm, a backtracking algorithm has to be implemented, in order to be able to define the state sequence of the best path. This backtracking algorithm needs to know at each time, which of the states have the highest probabilities. The result of the scaling is, that the absolute probabilities are no longer available. If the scaled variables are used instead, this makes no difference for the tracing.

6.4 Scaled training of elementary state parameters

The training of elementary states with a continuous output distribution, as described in Section 5.4, can also be expressed by the scaled forward and backward probabilities.

If relation (5.23) is re-formulated, the result is:

$$w_t = C_t(S)\cdot\tilde{P}^f_I(S,t)\cdot\hat{P}^b_I(S,t) = C_t(S)\cdot\tilde{P}^f_F(S,t+1)\cdot\hat{P}^b_F(S,t+1) \qquad (6.29)$$

All other variables are already expressed in terms of w_t. Apart from replacing (5.23) by (6.29) the algorithm remains unchanged:

Backward Training for elementary state with Gaussian distribution

```
Given: $\hat{P}^b_F(S,t+1)$
calculate:   $\hat{P}^b(S,t)$ and $\hat{P}^b_I(S,t)$
----------------------------------------------------------------
 1. calculate $w_t$:   $w_t = C_t(S)\cdot\tilde{P}^f_F(S,t+1)\cdot\hat{P}^b_F(S,t+1)$
 2. Update $\tau$:     $\tau^t = \tau^{t+1} + w_t$
 3. Update $\overline{m}$:     $\overline{m}^t = \overline{m}^{t+1} + w_t\cdot\overline{y}_t$
 4. Update V:     $V^t = V^{t+1} + w_t\cdot\overline{y}_t\cdot\overline{y}_t^*$
 5. $\hat{P}^b_I(S,t) = \hat{P}^b(S,t) = \hat{P}^b_F(S,t+1)\cdot P_E(S,\overline{y}_t)$
```

6.5 Conclusions

- All recursive algorithms from the previous chapter can be reformulated to use scaled probabilities instead of absolute probabilities. Basically, the algorithm remains unchanged.
- If ε is chosen subbiciently small, the results of the scaled training and recognition algorithms are largely identical to the results of the unscaled algorithms.
- The scaled algorithms do not suffer from numerical underflow.
- The choice of ε influences the training and recognition processes. If ε is chosen smaller, less states are pruned (= more accurate) but more states will be involved in the calculations (= more calculation time). For selectiong the right ε, a balance must be made between required accuracy and acceptable calculation time. Especially for large vocabularies this is important.

7. IMPLEMENTATION

The Schur algorithm as derived in Chapter 2 was implemented on a Texas Instruments TMS320C25 digital signal processor (DSP), which was chosen because of availability and past experience. A small software library was set up with routines that allowed to sample incoming speech using a fixed sampling rate. It used partly overlapping frames multiplied by a Hamming window, and calculated the autocorrelation coefficients, performing the Schur algorithm. The resulting parameters are written to a D/A-converter (for inspection on an oscilloscope) and written to memory (for further processing).

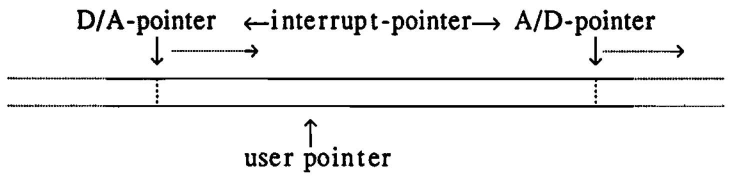
Because the DSP has only limited memory (24576 x 16 bits), the remaining software was implemented in ANSI-C on an APOLLO DN3000 workstation. All routines necessary for using the Recursive Markov Model (RMM) are written and tested on artificial data files. In order to demonstrate the use of RMMs, models for the numbers from 1 to 999999 and for a small Speech Controlled Robot (SCR) language are created. These examples show that the RMM-model is very powerful as a tool in speech recognition systems. Especially the flexible way of constructing hierarchical models that can be employed right away for training and recognition shows the usefulness of the RMM model.

7.1 LPC-analysis on the TMS320C25

An assembler program is implemented on the TMS320C25 processor, performing the Schur algorithm as derived in Chapter 2. A number of separate functions are written for different tasks that have to be performed. They are collected in a small library, called LPCBIOS. This library serves three functions:

- I/O, reading and writing blocks of samples from the A/D-converter or to the D/A-converter. Pre-multiplication by a window function is also supported.
- Mathematical block-operations like Hamming window, autocorrelation coefficients and reflection coefficients calculations.
- Other mathematical functions like the logarithm, exponential, square root, bllog and blexp (see Chapter 2).

The basis for sample handling is a cyclic buffer, to be imagined as an infinite buffer, from which samples are read and written, interrupt-basis:



Every clock tick a sample is read from the A/D converter and stored in the infinite buffer at the A/D-pointer. At the same time the value at the D/A-pointer is written to the D/A-converter. This process is fully transparent to the user, because it is interrupt-based. The user only has access to a user pointer which lies somewhere between the D/A- and the A/D-pointer. The user can freely read samples from the buffer, which where delivered by the A/D-converter some time back in the process. Samples can also freely be written to the buffer. After some time delay these samples will be sent to the D/A-converter.

Because in reality the buffer is cyclic, the A/D- and the D/A-pointer are identical. All LPCBIOS functions are accessed through macros, in order to make parameter passing transparent for the user. Besides, macros are easier to remember if suitable names are used. For the user the following macros are available:

PROGRAM *start*

This sets up the vector table for the TMS320C25. The interrupt vector is set to the assigned service routine, while the reset vector is set to *start*.

INIT *smptime,bufbeg,bufend*

A timer interrupt is enabled to generate an interrupt each *smptime*·0.1 µs. The area between *bufbeg* and *bufend* will be reserved for the sample buffer.

start,smptime,bufbeg and *bufend* must be constants.

MODPTR *num*

Modify user pointer by *num*. If *num* is negative, then the pointer is set back in time. This function tests whether the A/D-pointer is passed. If it is, MODPTR waits until all requested samples are available. Therefore, the user does not have to place a wait loop anywhere in the program. An automatic wait will be generated if samples required from the A/D-converter are not yet available.

WRSMP *samples,num*

Write *num* samples to the buffer. The user pointer will be incremented by *num* (using MODPTR). The samples are expected to start at address *samples*

RDSMP *samples,num*

Read *num* samples from the buffer, incrementing the user pointer with *num*. The samples are written starting with address *samples*.

RWSMP *samples,num,window.*

The same as RDSMP, only the samples are pre-multiplied by a window function. *window* indicates the starting address of the used window.

Before the Schur algorithm can be executed, the autocorrelation coefficients $R_0..R_n$ have to be calculated first:

AUCOR *samples,num,fltord*

Calculate autocorrelation coefficients. *num* samples starting at address *samples* are expected. *fltord* coefficients are calculated, overwriting addresses *samples* and further. All values are first calculated in 32 bits, then shifted in such a way that $0.5 \leq R_0 < 1.0$ and then rounded to 16 bits. The used shift value is returned in the accumulator.

SCHUR *samples,fltord*

Calculate reflection coefficients from the autocorrelation coefficients. The coefficients R_i are replaced by k_i.

samples, num, window and *fltord* are addresses of variables.

Other useful functions are:

INITH *window,num*

Create Hamming window of *num* samples, for use in RWSMP.

$$w_i = \left[0.54 - 0.46 \cdot \cos\left(\frac{2\cdot\pi\cdot(i+1/2)}{num}\right)\right]\cdot\sqrt{\frac{1}{num}}$$

The square root factor is used to make R_0 independent from the chosen window length and at the same time to prevent overflow in AUCOR. The cos(..) is approximated using a polynomial.

SQRT

Calculate square root of accumulator content. Used in INITH.

LOG

Calculate ${}^2\log$ of accumulator content.

EXP

Inverse function of LOG

BLLOG

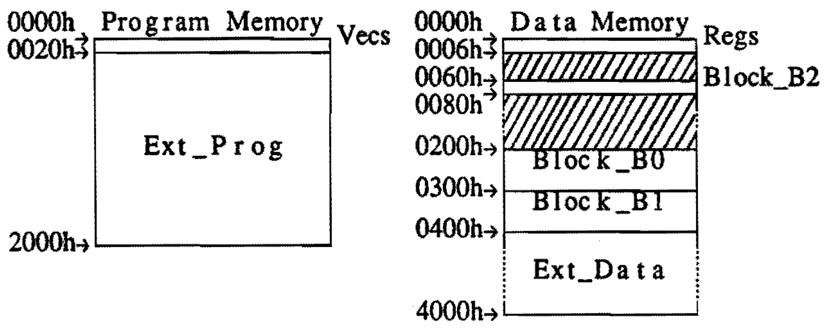
Calculation of $\log\left(\frac{1+x}{1-x}\right)$, using the piecewise linear approximation of Table 2.1.

BLEXP

Calculation of $\left(\frac{1+\exp(x)}{1-\exp(x)}\right)$, using the same piecewise linear approximation.

The main program consists of initialization, followed by the main loop. Inside this loop a block of samples is read from the buffer. The autocorrelation, reflection and log area ratio coefficients are calculated and written back to the buffer and to the data memory. If the data memory is full a TRAP instruction is reached which is able to create a breakpoint and stop the program. This makes it possible to display the memory contents or store them in a disk file for further processing.

The TMS320C25 board contains 8192 words program memory and 16384 words data memory divided in several sections as follows:



Vecs:	Contains jump vectors, in our case only the Reset- and interrupt-vector.
Ext_Prog:	Contains the .data and .text segments (tables and program code)
Regs:	Contains the IMR (Interrupt Mask Register) and the PRD (Timer period) Register.
Block_B2:	Small region reserved for local LPCBIOS-variables.
Block_B0:	Freely available, but is used by AUCOR.
Block_B1:	Freely available.
Ext_Data:	Freely available. Now contains the window shape table, cyclic samplebuffer and resultvectors.

The LPCBIOS only uses several variables in Block_B2, and the cyclic buffer in the Ext_Data area. The Block_B0 area is used by the AUCOR function for speed reasons. Because, relatively speaking, the calculation of the autocorrelation coefficients costs most time, every optimization of this calculation is beneficial. The fastest way is first to copy all samples to Block_B0, then re-configure this block to reside in Program memory. Block_B0 is the only block for which this is possible. After this the MAC-instruction can be used, which multiplies a sample taken from Data memory by a sample from the Program memory, at the same time accumulating the result of the previous multiplication. The result of this trick is that every multiply-accumulate operation costs only 1 cycle (0.1 μs).

For the remaining part of the program Block_B1 and Ext_Data (except the buffer) are free to use. Block_B1 is preferred, because it is fast on-chip memory.

A special note has to be made about the Schur algorithm, because the internal structure of the TMS320C25 makes it possible to implement the matrix multiplication in equation (2.26) very efficiently. The main loop only needs 7 instructions. Initially a matrix is set up containing all autocorrelation coefficients. The auxiliary registers AR0 and AR2 are used for tracing the two columns. Before the loop starts, this matrix looks like:

$$\text{AR0}\rightarrow\begin{pmatrix} R_0 & d_0^{(0)} \\ n_0^{(0)} & d_1^{(0)} \\ \vdots & \vdots \\ n_{n-1}^{(0)} & d_n^{(0)} \end{pmatrix}\leftarrow\text{AR2}$$

After incrementing AR0, k_1 can be calculated by dividing $n_0^{(0)}$ by $d_0^{(0)}$, which are pointed at by AR0 and AR2. After this k_1 is placed in the T-register as preparation for a number of multiplications. Then $n_0^{(0)} \cdot k_1$ is calculated and stored in the P-register. $n_0^{(0)}$ is no longer needed, so k_1 can be written to this place. After this the following code is performed. As a reference, the contents of the accumulator (A) and the P-register (P) are given after each instruction.

```
SCHURLP:ZALR *+         ; A=d_i^(p-1); P=k_p*n_i^(p-1);
        MPYA *-         ; A=d_i^(p-1)+k_p*n_i^(p-1); P=k_p*d_i+1^(p-1);
        SACH *+,AR0     ; d_i^(p)=A; AR2 points to d_i+1^(p-1);
        ZALR *          ; A=n_i+1^(p-1) ;
        MPYA *          ; A=n_i+1^(p-1)+k_p*d_i^(p-1); P=k_p*n_i+1^(p-1);
        SACH *+,AR1     ; n_i^(p)=A; AR0 points to n_i+1^(p+1);
        BANZ SCHURLP,*-,AR2; Repeat this for next i
```

The MPYA instruction performs a multiplication and an addition at the same time. For each row in the matrix two multiplications and two additions are needed. These can be implemented using only two instructions. Reading and writing of one element from each column cannot be done in less than 4 instructions. All these instructions are single cycle. Together with the branch instruction (BANZ), which is 3 cycles, the total loop costs only 9 cycles (= 0.9 μs) for each row.

After performing this code and resetting the pointers AR0 and AR2, the matrix looks like:

$$\begin{array}{r} \\ AR0\rightarrow \\ \\ \\ \\ \end{array}\begin{pmatrix} R_0 & d_0^{(1)} \\ k_1 & d_1^{(1)} \\ n_0^{(1)} & \vdots \\ \vdots & d_{n-1}^{(1)} \\ n_{n-2}^{(1)} & d_n^{(0)} \end{pmatrix}\begin{array}{l} \leftarrow AR2 \\ \\ \\ \\ \\ \end{array}$$

Again after incrementing AR0, k_2 can be calculated in the same way as k_1. Exactly the same code can be used for each succeeding order. After having performed this n times, the final content of the matrix is:

$$\begin{array}{r} \\ \\ \\ AR0\rightarrow \end{array}\begin{pmatrix} R_0 & d_0^{(n)} \\ k_1 & \vdots \\ \vdots & d_{n-1}^{(1)} \\ k_n & d_n^{(0)} \end{pmatrix}\begin{array}{l} \leftarrow AR2 \\ \\ \\ \\ \end{array}$$

As a comparison, the same code has already been implemented on the TMS32010 before [45]. Because the ZALR and MPYA instructions are not available in this processor, the fastest possible code on the TMS32010 uses 12 instructions (13 cycles). Because of the lower clock speed, the time involved is 13·0.2μs = 2.6μs. The speed gain between these two processors for the Schur algorithm is almost a factor 3, while the clock rate only increased a factor 2. This gain became possible by using the new pipelined instruction MPYA.

7.2 Structure-based handling of the RMM

In Chapter 5 has been shown that in a RMM each state has a number of child states and a number of parameters. All states can be stored in a tree structure. In a higher level language like C it is possible to exploit and implement such kinds of tree structures by means of pointers. Each state has its own name. It would be very useful if in the program each state could be addressed by its name instead of a number.

A collection of procedures has been written for various purposes. Together these form a library, that is used to manage RMM's and to perform all kinds of algorithms. The available functions are divided in several files:

rmm.h	Contains all structure definitions and the headers of all global functions. This file is included in all other files.
buildmm.c	Contains all functions for reading, writing, creating and accessing states of a RMM.
entropy.c	Contains the procedure for calculation of the entropy and state duration, as described in Section 5.6.
crtest.c	Contains all functions for reading, writing, creating and using discrete and continuous symbol sequences.
forback.c	Contains the Recursive Forward Backward Training algorithm as described in Chapter 6.
viterbi.c	Contains the Recursive Viterbi algorithm as described in Chapter 6.
main.c	This is the main program, that is accessing the other functions.

The structure M_MODEL contains the following fields:

type:	type of the state. Four different types are implemented: (elem,full,para,forw). Also options (unit, train) are stored in this field.
nchild:	number of child states
name:	state name
child:	pointer array to child state model
parm:	pointer to parameters, depending on type

More fields are present for various purposes, but they are not important for the further description. The definition of all structures can be found in the file RMM.H, which has been added to this thesis as appendix B. New fields can always be added as desired. However, all files must then be re-compiled.

The following functions are implemented in buildmm.c:

read_model(*filename*):
: read a collection of states from the given file.

write_model(*filename*):
: write complete model to disk.

find_model(*name*):
: returns a pointer to a state with the given *name*. If this state does not exist, a new one is created.

set_dsymbol(*symbol*):
: calculates $P_E(S_E,symbol)$ for all elementary states, where *symbol* is any discrete symbol.

set_csymbol(*vector*):
: calculates $P_E(S_E,vector)$ for all elementary states, where *vector* is any parameter vector. At the same time the vector elements and its square are stored in the variable *cvector* as preparation for the RMM training.

For example, if one needs to know how many child states the "Root" state in file "words.rmm" has, this can be performed using the following code:

```
read_model("words.rmm");
model = find_model("Root");
n = model->nchild;
```

To print the name of the first child state, this requires:

```
printf(model->child[0]->name);
```

Remember that in C an N-dimensional array uses indices 0..N-1 and not 1..N. The ASCII-file containing the RMM, that is accessed using read_model(..) and write_model(..), contains a list of descriptions for each state in the RMM, separated by empty lines. Two examples of such files can be found in appendix C. Each state description has the following format:

```
<name> <type> <options>
<description>
```

<name> can be any string except "elem", "full", "forw", "para", "train" and "unit" because these have another function. <name> can be surrounded by single or double quotes. Quotes can be used in the <name> by preceding it by a backslash. A backslash must be replaced by a double backslash. This convention is almost the same as used in C language strings. The backslash only serves this special function if it is the first character or if the name is surrounded by single or double quotes.

Therefore the following names are considered identical:

```
a  = \a  = 'a'  = "a" = '\a' = "\a"
\' = "'" = '\'' = "\'"
\\ = '\\' = "\\"
```

While writing an RMM to a disk file, the names are converted according to the following rules:

- If the name contains no spaces or tabs, no conversion occurs. Except if the first character is a backslash, or a single or double quote, the name is preceded by a backslash.
- Otherwise, if the name contains single quotes but no double quotes, the name is surrounded by double quotes and all backslashes are doubled.
- Otherwise, it is surrounded by single quotes. All backslashes and single quotes are preceded by a backslash.

Four types of states have been implemented. One of these is the elementary state, the others are three types of non-elementary states. This has been done because matrix A in many cases contains many zero elements, which makes it possible to reduce the storage space for these states.

If discrete symbols are used, elementary states need not be present in the file. Any state name that is referred to but is never defined, must be an elementary state. The associated symbol is supposed to be the first character of the state name. If the name is longer than one character, a warning is given. This makes it possible to use the output symbol of an elementary state as its name without declaring it. Elementary states with a continuous output vector have the following format:

```
<name> [elem] <options>
 μ1  μ2  μ3 ....
 σ1  σ2  σ3 ....
```

In the current implementation not the whole matrix Σ is stored for each state. Only the square root of the diagonal elements (σ_i, the standard deviation) are used, the others are supposed to be zero. This state will, whenever it is activated, emit a symbol $\bar{\mu}$ with each element μ_i disturbed by an uncorrelated Gaussian noise source with standard deviation σ_i. The reason for this choice is that Cholesky decomposition of matrix Σ (solving equation (5.29)) need not to be implemented yet, while the behavior of the RMM-training can still be fully tested. Less parameters need to be trained, which reduces the possibility of undertraining. If in the future correlated noise needs to be modelled, this extension is straightforward, as already described in Chapter 6.

The next state type is the "full" model. The syntax is:

```
<name> full <options>
         <child1> <child2> . . .
             aI1       aI2    . . .
<child1>     a11       a12    . . .  a1F
<child2>     a21       a22    . . .  a2F
   :          :         :             :
```

Many times the full A-matrix is not needed. The "forw" model only contains transitions from state i to i, i+1 and i+2. This model is also known as the Bakis model. Another useful simplification is the "para" model, in which all states are parallel. The syntax of these two state types is:

```
<name> forw <options>
                   aI1  aI2
<child1>     a11   a12  a13
<child2>     a22   a23  a24
   :          :     :    :
<childn>     ann   anF
```

<name> para <options>
<child1> a_{11}
<child2> a_{12}
: :
<childn> a_{1n}

By now, two possible options have been implemented:

"train"	Indicates that this state must be trained. If this option is not present, calculations are saved.
"unit"	This option is only used by the Viterbi algorithm and the entropy calculation. It means that child states are not traced any more, which also saves calculations and storage.

Additionally, the routine read_model(*filename*) calculates the entropy and state duration using the algorithm described in Chapter 5. Because for a "unit" state (a state with the option "unit" set) the internal structure is no longer traced, the entropy of a "unit" state is set to zero.

7.3 Implementation of the Recursive Forward-Backward algorithm

The Recursive Forward Backward Algorithm as described in the previous chapter has been implemented in the file "forback.c". For storage of the forward and backward variables another structure (M_PROB) is used with four fields:

ip: stores $P_I^f(S,t)$ or $P_I^b(S,t)$

sp: stores $P^f(S,t)$ or $P^b(S,t)$

fp: stores $P_F^f(S,t+1)$ or $P_F^b(S,t+1)$

ch: pointer to probabilities of child states.

Child states belonging to the same parent state are always stored on succeeding memory locations, implying that (**ch**) points to the first child state, (**ch**+1) to the second, and so on. The number of child states is not stored here, because it can be found in the m_model structure as described in 7.2.

Two functions are implemented for dynamic allocation and freeing of these structures:

new_mprob(*mprob,model*):
: Storage is allocated for the child states of *mprob*, where *model* is a pointer to the model description. This function is not allowed for elementary states, having no child states.

free_mprob(*mprob,model*):
: Free child states of *mprob*. If *mprob* still contains lower level child states these are freed too.

The recursive forward and backward training algorithms must be prepared by one of set_dsymbol(..) or set_csymbol(..). These will calculate P_E(S,symbol) for all elementary states, the results of which are needed in all further calculations. The different parts and versions of the Recursive Forward Backward Training algorithm are implemented in the following routines:

build_forw(*model,prob*):
: Implements the recursive forward algorithm without using previous time information. This routine can only be used in case of elementary states, or when previously the current state has been pruned. The resulting tree will be written in *prob*, freeing and allocating memory as needed, using both preceding routines. Before calling this or following functions, the current symbol must be set using set_dsymbol(..) or set_csymbol(..).

calc_forw(*model,prob,prev*):
: Implements the recursive forward algorithm using previous time probabilities. If these are not available or the state is elementary, build_forw(*model,prob*) is called instead. If *prob* and *prev* (which are pointers) point to the same tree, the previous time probabilities will be overwritten with the new values. This is allowed.

build_backw(*model,backw,forw,scale*):
: Similar to build_forw(..), only the training is integrated in the backward algorithm. Therefore, additionally, the forward probabilities at the same time and the scale factor, as described in Chapter 6, are needed. Again, this procedure only operates on elementary states, or else when previously the current state was pruned.

calc_backw(*model,backw,forw,scale*):
: Implements the backward training algorithm. For elementary states or when previous time probabilities are not available, build_backw(..) is called. Before entering this procedure, *backw* points to the previous time probabilities. Afterwards, the current time probabilities are stored in the same place.

These functions are implemented in the file "forback.c" and are normally not used directly. Because the recursive forward and backward training algorithms are always executed on a whole sequence, and not on a single symbol at a single time, four functions are available that are easier to use than the ones above:

d_forward(*model,prob,sequence*):
: Now *prob* is an array where for each element the recursive forward algorithm is executed. Afterwards, the array contains all forward probabilities of all states (as far as they are not pruned). Discrete symbols are expected in *sequence*. The Forward score, P^f_F(*model*,T+1), is returned.

d_backward(*model,prob,sequence*):
: The forward probabilities are used to execute the recursive backward training algorithm. All states with the option "train" set are trained. During the backward pass, *prob* is cleared as far as it is no longer needed. The Backward score, P^b_I(*model*,1), is returned. This value should be the same as the Forward score; only some small differences are possible because of state pruning. The parameters are not yet updated in order to make training with multiple sequences possible. This update is done by another function: update_model(..);

c_forward(*model,prob,vectors,length*)
: The same as d_forward, except that the sequence is expected to consist of continuous vectors.

c_backward(*model,prob,vectors,length*)
: The same as d_backward, except the sequence is expected to consist of continuous vectors.

After the Recursive Forward Backward algorithm has calculated all expected numbers of transitions (τ_{ij}) for non-elementary states, or the updating parameters ($\tau,\overline{m},V$) for elementary states, the RMM can be updated by:

update_model():
: calculates all updated parameters for all states, being trained using the Recursive Forward Backward algorithm.

Let us suppose we have a file "sequence.t" containing a symbol sequence that is supposed to be produced by an RMM contained in the file "grammar.rmm" with root state "Root". The training code looks like:

```
M_PROB prob[MAX]; /* declare storage array */
char sequence[MAX+1]; /* declare array for sequence */
M_MODEL *model; /* declare pointer to model */
FILE *f; /* declare file variable */
read_model("grammar.rmm");
model = find_model("Root");
f = fopen("sequence.t","r");
read_dseq(f,sequence); /* declared in crtest.c */
fclose(f);
d_forward(model,prob,sequence);
d_backward(model,prob,sequence);
update_model();
```

Multiple training sequences can be handled by executing read_dseq(..) (from "crtest.c"), d_forward(..) and d_backward(..) for each sequence. update_model(..) must be called after all sequences are handled. Multiple iterations can be executed, repeating the whole process (without loading "grammar.rmm" again, of course). It is possible to match multiple training sequences with different states. If training sequences are present which are known to be produced by another state in the model (the "Root" state is not the only possibility), training can be performed using only a subset of the RMM. This can be used for so called **supervised training**. Because of the hierarchical structure of the model, it makes no difference at all whether the whole RMM or only a part of it is trained. Another way to imagine this, is to assign to each sequence its own root state. In appendix A is shown that this still assures Maximum Likelihood training.

7.4 Implementation of the Recursive Viterbi algorithm

Basically, the Recursive Viterbi algorithm has the same structure as the Recursive Forward algorithm. However, since there is no backward pass, the previous time probabilities can always be overwritten when new ones are calculated. This is a simplification, because no separate tree is needed for current time and previous time probabilities. In order to trace the states that have been visited in the past, an additional tree structure is needed. The structure V_PROB contains the following fields:

ip: stores $P_I^V(S,t)$

sp: stores $P^V(S,t)$

fp: stores $P_F^V(S,t+1)$

ch: pointer to probabilities of child states.

vtree: pointer to history information.

The correspondence with the structure M_PROB is evident. Only an additional field is used for reconstructing the whole history. This history information is stored like a linked list, where for each list element the structure V_TREE is used, containing the following fields:

nref:	number of pointers pointing to this address
prev:	link to past history. For the last element of the list, this is zero.
model:	pointer to model of visited state. If this field has the value 0, this indicates that the current state has been left. Only "unit" states and elementary states are supposed to finish immediately, without indicating this in the history list.

Each state in the Viterbi algorithm has its own history list, but if multiple lists are partially the same, the identical parts are only stored once. Therefore, it is possible that multiple pointers point to the same list element.

If during the Viterbi algorithm states are pruned, it is necessary that the corresponding history list is removed. If the list elements, however, are still in use by other states this is not allowed. For this the field **nref** is used. Only if **nref** becomes zero, the memory occupied by this list element is released. This memory management is very important, because otherwise the lists could easily become very large.

The option "unit" of each state is used in the Viterbi algorithm for further reduction of the history list. "Unit" states are the lowest level states included in the history list. Only the identity of each word is usually needed for recognition. Therefore it is not necessary for the list to contain the detailed information about the visiting of each state.

The functions new_vprob(..), free_vprob(..), new_vtree(..) and free_vtree(..) need no further explanation. They behave exactly as expected, as suggested by their appointed names. The function new_vtree(..) has an additional parameter, indicating the list it should be linked to. The field **nref** is incremented, indicating the creation of an additional pointer. Also free_vtree(..) checks **nref**, refusing deletion if **nref**>1.

The functions build_viterbi(..) and calc_viterbi(..) are counterparts of build_forw(..) and calc_forw(..). Instead of additions, the maximum operator is used, and the history list of field **vtree** is updated as well. Because unit states and its lower level states do not require updating of the history list, additional functions build_lower(..) and calc_lower(..) are implemented, identical to build_viterbi(..) and calc_viterbi(..) except that the history list can only be passed to other states or be pruned, but not extended.

As is the case with the forward backward algorithm, two functions are available that are able to perform the Viterbi algorithm for a whole sequence. These are:

d_viterbi(*model,prob,sequence,file*)
: *prob* is a single tree, used for storing all Viterbi probabilities dynamically. The Viterbi algorithm is executed for the whole sequence. The result of the training is written to *file*, together with some diagnostics information.

c_viterbi(*model,prob,vectors,length,file*)
: The same, only continuous symbols are used.

Finally, the code to recognize a sequence stored in file "sequence.t", using an RMM stored in file "grammar.rmm", writing the recognition result to file "main.lst" looks like:

```
M_PROB prob; /* declare storage */
char sequence[MAX+1]; /* declare array for sequence */
M_MODEL *model; /* declare pointer to model */
FILE f; /* declare file variable */
read_model("grammar.rmm");
model = find_model("Root");
f = fopen("sequence.t","r");
read_dseq(f,sequence);
fclose(f);
f = fopen("main.lst","w");
d_viterbi(model,prob,sequence,f);
fclose(f);
```

7.5 Simulations

For the reading and creation of artificial data files, another module is implemented: "crtest.c". It contains two internal functions and four functions that are accessible from outside. The internal functions are:

sel_trans(*parm,n*)
Select a random number, and, accordingly, select a transition (0..n), guided by the probability distribution given in *parm.*

gnoise() Gaussian random generator with mean 0 and standard deviation 1. This is approximated by summing 9 uniform distributions.

The global functions are:

read_dseq(*file,sequence*):
Read discrete symbol sequence from disk. One line from disk is read into a string.

read_cseq(*file,sequence,vectors*):
Read a discrete and continuous symbol sequence from disk. Each line in the file contains a discrete symbol (only for readability), followed by a vector of floating point values. An empty line or the end of file indicates the end of the sequence.

outp_dmm(*model,file*)
Create discrete output symbols by simulating the operation of an RMM. Each time random transitions are made, using sel_trans(..), and for each elementary state, a symbol is written to *file.*

outp_cmm(*model,file*)
This function is identical to the function above, only now each elementary state produces a continuous symbol vector, using the mean vector $\overline{\mu}$ (μ ..μ), disturbed by uncorrelated Gaussian noise with standard deviation σ .

main.c contains the main program, currently performing the following:

- Files on the command line are read, using read_model(..). Some diagnostics like the size, the entropy etc. are printed.
- The user is asked for the number of training sequences. If a positive number is given, these are created and stored in the file "sequence.t". If zero is given, no new sequences are generated.

- The user is asked for the value of epsilon, used for pruning in the Viterbi and the Forward Backward algorithm. Practical values are around 10^{-5}.
- The Recursive Viterbi, Forward and Backward Training algorithms are executed at each sequence in "sequence.t". Some diagnostics, like the time duration and the tree size during the Viterbi algorithm, are printed in the file "main.lst".
- Some diagnostics are provided about the total memory requirement of the Viterbi and Forward Backward algorithms.
- Finally update_model() is called, and the updated RMM is written to the file "trained.rmm"

Appendix C contains the listings of "words.rmm", "grammar.rmm", "scr.rmm" and the directed graph of "grammar.rmm".

"words.rmm" is a set of states representing all kinds of words. Each word is expected to consist of its ASCII-characters. The "forw" model is used, which models additional transitions, repeating and skipping characters. The state "anychar" is also included to represent random characters. The probabilities of repeating, skipping and replacing characters are all set to 2%. Each ASCII-character represents a vector (A,0,0,0,0), where each vector is disturbed by Gaussian white noise with standard deviation (0.2, 0.2, 0.2, 0.2, 1.0).

"grammar.rmm" contains the grammar of all numbers 1..999999. This model describes how any number in this range can be represented by 21 units. These units are present in "words.rmm"

"scr.rmm" contains the grammar of a small SCR-language. The specification has been borrowed from Brown and Wilpon [11], having made some minor changes. In our system it is advantageous to use a hierarchical description (the original language is not hierarchical), many intermediate states are introduced to simplify the description.

8. RESULTS AND CONCLUSION

8.1 Simulation results

The command: main words grammar
starts the main program, loading the words from "words.rmm" and the grammar from "grammar.rmm" (see Appendix C for a listing). A specified number of test sequences can be generated, stored in "sequence.t". An example test sequence, generated using "grammar.rmm", is:

```
e  4.6341  0.1010  0.0879  0.2974  0.4864
i  9.0702  0.3027 -0.1280  0.1231 -0.1542
g  6.8387  0.1930 -0.0174 -0.4797 -1.2646
h  7.7956 -0.1152  0.0777 -0.1716 -0.1518
t 19.7507 -0.1222 -0.1848  0.1071  0.3885
y 24.9349  0.3453  0.0799  0.2096 -1.4035
o 15.1136  0.0717  0.2137  0.0877 -1.2298
n 13.6729 -0.1127  0.1025  0.0469  0.0961
e  5.1522 -0.1755  0.0579  0.0106  0.8079
```

Running the Viterbi and the Forward Backward algorithm ($\varepsilon=10^{-4}$) on this sequence, the following output is generated:

```
sequence:  "eightyone"

size after    1:426
size after    2:252
size after    3:24
size after    4:30
size after    5:35
size after    6:56
size after    7:120
size after    8:6
size after    9:6

Best sequence:
1Root
1     1-999999
1          1-999
1               1-99
1                    20-90
1                         thir-9
1                              eigh
1                         ty
1                    1-9
7                         1
```

```
viterbi   score: 4.77061e-05,    3.917  sec
forward   score: 4.80335e-05,    2.833  sec
Backward  score: 4.80335e-05,    0.333  sec
```

The re-trained parameters of some states have been determined:

```
e            elem train
            4.9288 -0.0248  0.0486  0.1027  0.4314
            0.2462  0.1624  0.1211  0.1798  0.6661

i            elem train
            9.0351  0.1514 -0.0640  0.0616 -0.0771
            0.1457  0.2071  0.1552  0.1542  0.7113

thir-9     para train
thir       0.0714286
fif        0.0714286
eigh        0.571429
4,6,7,9    0.285714
```

We can see that despite the random disturbance of the vectors the word "eightyone" is correctly recognized. From the output we can conclude how this sequence is produced by different states at different levels. The number at the beginning of each line indicates how many pointers are pointing to this item in the history list. If this is 1, it means that all other possible paths are pruned, which means that this state is the only possibility left.

The state parameters are updated to confirm better to the given training sequence. Of course, one sentence is a too small amount of data, but we can already see the effect of training.

During each time step of the Viterbi algorithm the history lists of all states are continuously updated. Therefore, it is not necessary to wait until the end of the sentence (which will never come when no segmentation has been performed). If all history lists share the same beginning, this beginning can already be output as the recognition result. The recognition system needs no pre-segmentation any more, and the result is delivered while the Recursive Viterbi algorithm is running somewhat delayed. A word may even be recognized while it is not even fully pronounced yet, as soon as all alternative word probabilities are lower than ε.

For example, the sequence:

```
t 19.8673 -0.0278  0.1005  0.5362  1.7200
w 22.9870 -0.2995  0.1347  0.0221 -0.3224
o 15.1734 -0.1049 -0.1848 -0.1719  0.5485
t 20.0667 -0.0522  0.1411  0.0067 -0.0256
h  8.1792 -0.1579  0.0526  0.0049 -0.9493
o 14.9541  0.0295  0.3610 -0.0808 -2.1353
u 20.9735  0.0021  0.2191  0.0065 -1.1071
s 18.8304  0.2839 -0.2459 -0.1759  1.9774
```

gives as recognition result:

```
sequence:  "twothous"

size after    1:462
size after    2:336
size after    3:218
size after    4:21
size after    5:6
size after    6:6
size after    7:7
size after    8:7

Best sequence:
1Root
1     1-999999
1           1-999
1                 1-99
1                       1-9
1                            2
6           1000

viterbi   score: 0,    3.050 sec
forward   score: 0,    2.217 sec
Backward score: 0,     0.050 sec
```

The word "thousand" is already recognized, even when it has not yet finished. The Forward, Backward and Viterbi scores are all zero, because "twothous" is not a valid number at all. Still, the history list of the best state already contains "1000" as recognized state.

8.2 Conclusion

The aim of this thesis has been to search for efficient algorithms for automatic speech recognition. As this is a very wide area, it cannot be covered completely by one single person. Therefore, the search has been restricted to algorithms that can be implemented on a digital signal processor (DSP) and adress the various aspects as presented in this thesis, summarized below.

First, the calculation of the log area parameters has been implemented on a DSP (the TMS320C25). If these parameters are used as a start of a speech recognition system, the only reasonable judgment possible is the score of the total speech recognition system. As long as this system has not been completed, real evaluation is not possible.

A segmentation algorithm based on the log area parameters has been developed and tested. The idea behind it was that speech consists of a number of stationary segments, connected by transitions. Using the correlation between nearby frames, the borders between segments could be estimated quite accurately. The only problem was that many borders were missing, and many unwanted borders had been inserted. If additional information would be available about the structure of the pronounced sentence, this could be solved. However, this information is only available after recognition has taken place, while the segmentation will be used before recognition. The only possible conclusion is that accurate segmentation of speech is not possible without using higher level knowledge. The segmentation should be integrated in the recognition process, instead of being performed in advance.

A new algorithm has been derived that is able to perform recognition without segmentation. It has been based on a hierarchical stochastic model of the whole speech production process. This model is called the Recursive Markov Model (RMM). It is an extension to the Hidden Markov Model (HMM), that is currently very popular in speech recognition research. Another new algorithm, the Recursive Forward Backward Training algorithm, had been developed, which has proved to be able to train all parameters in the RMM, fully supporting state sharing, multiple training sequences and supervised training. These algorithms have been implemented using ANSI-C on an APOLLO DN3000 workstation. The results show that the RMM is especially advantageous in large vocabulary speech recognition using syntax. The current system is not able to perform recognition in real time, because the Apollo system on which it has been tested is too slow. Neither could it be implemented on the available DSP-system because it has insufficient memory. Since the current calculation times on the APOLLO are seconds rather than hours, real time large vocabulary speech recognition is not far away.

The following experiment shows that the memory requirement and calculation time needed for a speech recognition system based on RMM grows much less than linear with the library size. Let us suppose we have a recognition system with N words, that needs to be increased to 2N words. The library is doubled in the worst case, so, in principle, the memory requirement and calculation time will also be doubled. However, many words in the two libraries contain identical syllables. Using State Sharing for these syllables reduces the memory requirement for this new system. State Pruning allows that not the whole model needs to be searched for recognition. In practice this will mean that many of the added words will be pruned before the word to be recognized is fully pronounced. This reduces the number of calculations. The larger the library, the more State Sharing and Pruning can be used. This makes RMM speech recognition especially attractive for large vocabulary speech recognition.

The current implementation assumes that the graph describing the state relations (as in appendix C for the numbers 1-999999) is a directed graph without loops. If the Forward, Backward or Viterbi algorithm reach such a loop, it will be traced deeper and deeper without ending in an elementary state. This situation occurs when a state contains itself. For instance a sentence might contain another sentence ("John said: 'I am ill' "), or a PASCAL procedure might contain another procedure. Grammars using this feature (like PASCAL) are not supported by the current implementation, but this can be solved by limiting the number of levels. This is reasonable, since "John said: 'Mary said: ‘I am ill‘'" is not a very likely sentence, and a PASCAL program is not very likely to use procedure nesting of more than 5 levels. During the entropy calculation a test will be executed to locate this kind of loops in the state relation graph. If loops are discovered a warning is given indicating the offending state name.

The following suggestions can be made for a follow-up:

Testing the algorithm using real speech. The LPC-analysis and recognition have now been performed on different systems, that are not linked to each other. Implementation of both processes on a fast system with sufficient memory plus the possibility to sample speech would provide a valid proof that the algorithms developed in this thesis operate practically and efficiently.

Implementing other state types. Apart from the four state types already implemented (elem,full,forw,para), various others are possible. The elementary state with correlated noise (currently only uncorrelated noise has been used) has already been mentioned. Other probability distributions than only Gaussian are possible too.

Using a neural net as elementary state seems very interesting. This has already been suggested for HMM by Niles [47], but the same approach is possible for the RMM. Neural nets cannot be trained using Maximum Likelihood, but corrective training is normally possible. All other states can still be trained using ML, because the Recursive Forward Backward algorithm only uses local variables that are still available.

Besides speech recognition, RMM can as well be used for other types of applications. The model has many properties in common with the Neural Net approach. Any pattern that can be subdivided in sub-patterns, while the pattern score is a linear combination of the sub-pattern scores, is suitable for RMM pattern recognition. The main difference is that the RMM will be especially useful if structural (hierarchical) information about the recognition problem is available, which a Neural Net does not require nor uses. On the other hand RMM has less parameters, which makes it better trainable.

APPENDIX A: ML-ESTIMATION

The training algorithm, derived in a heuristic way in chapter 5, is similar in structure to the training algorithm used in HMM. Baum [10] has proven that the Forward-Backward algorithm is guaranteed to converge to a local maximum of the likelihood function. This proof is extended by Liporace [39], who showed that this is also valid for a much larger class of re-estimation formulas. In RMM the model structure is further extended. The question arises whether this extension still guarantees convergence of the training algorithm. In this appendix will be shown that it does.

The outline of this proof follows the same lines as Liporace [39], containing three basic differences:

- State Sharing
- Hierarchical modelling
- Multiple training sequences

The state sharing and hierarchical modelling cause extra terms in the re-estimation formulas, but the structure is not changed. To allow for multiple training sequences the definition of the auxiliary function $Q(\lambda,\hat{\lambda})$ as used by Liporace had to be changed such that this only causes extra terms in the re-estimation formulas.

The outline of the proof is as follows:

- The likelihood function is defined for the RMM.
- An auxiliary function is defined, and some properties of this function that are needed later on are unfolded.
- Prove that increase of the auxiliary function assures increase of the likelihood function.
- Derive a re-estimate of A-matrix for each state by maximizing the auxiliary function, using the Lagrange multiplier method.
- Derive re-estimate of $\bar{\mu}$ and Σ for elementary states by maximizing the auxiliary function.
- Prove that the critical point of the auxiliary function is a unique maximum.

The Maximum Likelihood training must maximize the probability that the RMM produces the given training sequences. This function is defined as $P^f_F(\text{Root},T+1)$ (see chapter 5 for details) for one training sequence produced by the Root state. More general is, to consider more training sequences Y^k {k=1..K} that are supposed to be produced by (possibly the same) Root states R^k. In this appendix we will use the following notation:

$P_\lambda(R^k,Y^k)$ = probability of R^k to produce Y^k, given parameters λ

The likelihood function for multiple training sequences is the product of the likelihood of each training sequence, because they are supposed to be independent:

$$P_\lambda(R,Y) = \prod_{k=1}^{K} P_\lambda(R^k,Y^k)$$

Where R is the set of Root states, and Y the set of training sequences. Further let $\mathcal{S}(R^k)$ be the set of all possible paths in state R^k, and $P_\lambda(R^k,Y^k,s)$ the probability of a single path s in R^k to produce Y^k. Then:

$$P_\lambda(R,Y) = \sum_{s^1\in\mathcal{S}(R^1)} \cdot\cdot \sum_{s^K\in\mathcal{S}(R^K)} P_\lambda(R,Y,s^1..s^K)$$

$$P_\lambda(R^k,Y^k) = \sum_{s\in\mathcal{S}(R^k)} P_\lambda(R^k,Y^k,s)$$

We define the auxiliary function $Q(\lambda,\hat{\lambda})$ as:

$$Q(\lambda,\hat{\lambda}) = \frac{\sum_{s^1\in\mathcal{S}(R^1)} \cdot\cdot \sum_{s^K\in\mathcal{S}(R^K)} P_\lambda(R,Y,s^1..s^K)\cdot\log P_{\hat{\lambda}}(R,Y,s^1..s^K)}{P_\lambda(R,Y)}$$

This can be rewritten as the sum of individual contributions of the training sequences. The first step is splitting all terms as the product over all training sequences:

$$Q(\lambda,\hat{\lambda}) = \frac{\sum_{s^1\in\mathcal{S}(R^1)} \cdot\cdot \sum_{s^K\in\mathcal{S}(R^K)} \prod_{m=1}^{K} P_\lambda(R^m,Y^m,s^m)\cdot\log \prod_{k=1}^{K} P_{\hat{\lambda}}(R^k,Y^k,s^k)}{\prod_{m=1}^{K} P_\lambda(R^m,Y^m)}$$

The product over k can be pulled out of the log function, as a summation over k in front of the fraction:

$$Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_{s^1\in S(R^1)} \cdot\cdot \sum_{s^K\in S(R^K)} \prod_{m=1}^{K} P_\lambda(R^m,Y^m,s^m)\cdot \log\ P_{\hat{\lambda}}(R^k,Y^k,s^k)}{\prod_{m=1}^{K} P_\lambda(R^m,Y^m)}$$

Because the training sequences are independent, the summations in the nominator can be rearranged, except for sequence k:

$$Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\prod_{m\neq k} P_\lambda(R^m,Y^m)\cdot \sum_{s^k\in S(R^k)} P_\lambda(R^k,Y^k,s^k)\cdot \log\ P_{\hat{\lambda}}(R^k,Y^k,s^k)}{\prod_{m=1}^{K} P_\lambda(R^m,Y^m)}$$

The nominator and denominator contain a large number of common factors, that can be removed:

$$Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_{s\in S(R^k)} P_\lambda(R^k,Y^k,s)\cdot \log\ P_{\hat{\lambda}}(R^k,Y^k,s)}{P_\lambda(R^k,Y^k)}$$

This expression will be used in further processing, because each term in the sum contains only information of one training sequence Y^k.

In order to show that $Q(\lambda,\hat{\lambda}) > Q(\lambda,\lambda)$ implies that $P_{\hat{\lambda}}(R,Y) > P_\lambda(R,Y)$, note that $\log(x) \leq x-1$, with equality if and only if $x = 1$. Therefore:

$$0 < Q(\lambda,\hat{\lambda}) - Q(\lambda,\lambda) =$$

$$= \frac{\sum_{s^1\in S(R^1)} \cdot\cdot \sum_{s^K\in S(R^K)} P_\lambda(R,Y,s^1..s^K)\cdot \log\left(\frac{P_{\hat{\lambda}}(R,Y,s^1..s^K)}{P_\lambda(R,Y,s^1..s^K)}\right)}{P_\lambda(R,Y)}$$

$$\leq \frac{\sum_{s^1\in S(R^1)} \cdot\cdot \sum_{s^K\in S(R^K)} P_\lambda(R,Y,s^1..s^K)\cdot \left(\frac{P_{\hat{\lambda}}(R,Y,s^1..s^K)}{P_\lambda(R,Y,s^1..s^K)} - 1\right)}{P_\lambda(R,Y)}$$

$$= \frac{\sum_{s^1\in S(R^1)} \cdot\cdot \sum_{s^K\in S(R^K)} \left(P_{\hat{\lambda}}(R,Y,s^1..s^K) - P_\lambda(R,Y,s^1..s^K)\right)}{P_\lambda(R,Y)}$$

$$= \frac{P_{\hat{\lambda}}(R,Y)}{P_\lambda(R,Y)} - 1 \quad \Rightarrow \quad \frac{P_{\hat{\lambda}}(R,Y)}{P_\lambda(R,Y)} > 1 \quad \Rightarrow \quad P_{\hat{\lambda}}(R,Y) > P_\lambda(R,Y)$$

All we have to do is to determine the maximum of $Q(\lambda,\hat{\lambda})$. This would provide us with at least an increase of the likelihood. By iterating this procedure, the maximum of the likelihood function can be established. If after a sufficient number of iterations $P_{\hat{\lambda}}(R,Y)=P_{\hat{\lambda}}(R,Y)$, then we can conclude that also $Q(\lambda,\lambda)=Q(\lambda,\hat{\lambda})$. Because $\hat{\lambda}$ is the maximum of $Q(\lambda,\hat{\lambda})$ and this maximum is unique (which will be proved later), $\lambda=\hat{\lambda}$. So the algorithm always converges to a critical point of $Q(\lambda,\hat{\lambda})$.

For the parameters $\hat{A}$ of non-elementary state S the maximum of the auxiliary function is determined applying the Lagrange multiplier method, because an extra restriction is put on this matrix that all row elements must sum to 1.

If some recursive state sequence s (a state sequence on all levels of the recursion) is given, the probability of the RMM following this path producing the given sequence Y^k can be expressed by:

$$P_\lambda(R^k,Y^k,s) = \prod_{t=1}^{T}\left\{ \prod_{a\in s_t} a \cdot P_E(S_E(s_t),Y_t^k) \right\}$$

where $S_E(s_t)$ is the elementary state visited in path s on time t, and s_t is the set of transitions visited in path s on time t. Thus, we have:

$$Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_s P_\lambda(R^k,Y^k,s) \cdot \sum_{t=1}^{T}\left\{ \sum_{a\in s_t} \log a + \log P_E(S_E(s_t),Y_t^k)\right\}}{P_\lambda(R^k,Y^k)}$$

A typical initial probability a_{Ij} of some state S can be re-estimated applying the Lagrange multiplier method, by solving:

$$0 = \frac{\partial}{\partial \hat{a}_{Ij}} \left[Q(\lambda,\hat{\lambda}) - \theta\ (\sum_m \hat{a}_{Im} - 1)\right]$$

$$= \sum_{k=1}^{K} \frac{\sum_s P_\lambda(R^k,Y^k,s) \cdot \sum_{t=1}^{T} \sum_{a_{Ij}\in s_t} 1/\hat{a}_{Ij}}{P_\lambda(R^k,Y^k)} - \theta$$

$$= \frac{1}{\hat{a}_{Ij}} \sum_{k=1}^{K} \sum_{t=1}^{T} \frac{\sum_s \sum_{a_{Ij}\in s_t} P_\lambda(R^k,Y^k,s)}{P_\lambda(R^k,Y^k)} - \theta$$

The summation in the nominator is taken over all paths that contain a transition a_{Ij} at time t. All probabilities of these paths are added. We could also reverse this summation. For all copies of transition a_{Ij} (because of State Sharing) the probabilities of all paths that make transition a_{Ij} at time t are added. This results in the equation:

$$\frac{1}{\hat{a}_{Ij}} \sum_{k=1}^{K} \sum_{t=1}^{T} \frac{\sum\limits_{a_{Ij}} \sum\limits_{S_t \ni a_{Ij}} P_\lambda(R^k, Y^k, S)}{P_\lambda(R^k, Y^k)} - \theta = 0$$

The fraction in this equation in fact comprises nothing more than the probability that transition a_{Ij} is made at time t, divided by the probability that any transition is made. This is the definition of τ_{Ij}, that can be calculated by the Recursive Forward Backward algorithm. The summation over all copies of a_{Ij} is automatically performed, because τ_{Ij} is shared among all copies of state S. Hence, we can write:

$$\frac{1}{\hat{a}_{Ij}} \cdot \tau_{Ij} - \theta = 0 \quad \Rightarrow \quad \hat{a}_{Ij} = \frac{\tau_{Ij}}{\theta}$$

To determine the value of θ, we sum for all j:

$$\sum_j \frac{\tau_{Ij}}{\theta} = \sum_j \hat{a}_{Ij} = 1 \quad \Rightarrow \quad \theta = \sum_j \tau_{Ij}$$

The re-estimation formula becomes:

$$\hat{a}_{Ij} = \frac{\tau_{Ij}}{\sum\limits_m \tau_{Im}}$$

In exactly the same way we find the re-estimation formulas:

$$\hat{a}_{ij} = \frac{\tau_{ij}}{\sum\limits_m \tau_{im} + \tau_{iF}} \; ; \; \hat{a}_{iF} = \frac{\tau_{iF}}{\sum\limits_m \tau_{im} + \tau_{iF}}$$

These are the same as expected in equation (5.16).

In order to show that $Q(\lambda,\hat{\lambda})$ has a unique maximum, let us consider the second derivative in respect to all parameters a_{ij}, a_{Ij} and a_{iF}. To handle the restriction that rows of matrix A must sum to one, we calculate this derivative along the line between two valid parameter sets λ^1 and λ^2.

First, let us compare two parameter sets λ^1 and λ^2, with the only difference that λ^1 contains a^1_{Ij} and λ^2 contains a^2_{Ij} for state S. Let λ^3 be a linear convex combination, containing $a^3_{Ij} = \theta a^1_{Ij}+(1-\theta)a^2_{Ij}$ $\{ 0 < \theta < 1\}$:

$$\frac{\partial^2}{\partial\theta^2} Q(\lambda,\lambda^3) = \frac{\partial^2}{\partial\theta^2}\sum_{k=1}^{K} \frac{\sum_{s} P_\lambda(R^k,Y^k,s)\cdot\sum_{t=1}^{T}\sum_{j}\sum_{a_{Ij}\in s_t}\log(a^3_{Ij})}{P_\lambda(R^k,Y^k)}$$

$$= \frac{\partial^2}{\partial\theta^2}\sum_{j=1}^{N}\log\left[\theta a^1_{Ij}+(1-\theta)a^2_{Ij}\right]\sum_{k=1}^{K}\sum_{t=1}^{T}\frac{\sum_{s}\sum_{a_{Ij}\in s_t} P_\lambda(R^k,Y^k,s)}{P_\lambda(R^k,Y^k)}$$

$$= \frac{\partial^2}{\partial\theta^2}\sum_{j=1}^{N}\tau_{Ij}\cdot\log\left[a^2_{Ij}+\theta\cdot(a^1_{Ij}-a^2_{Ij})\right] = \frac{\partial}{\partial\theta}\sum_{j=1}^{N}\frac{\tau_{Ij}\cdot(a^1_{Ij}-a^2_{Ij})}{a^2_{Ij}+\theta\cdot(a^1_{Ij}-a^2_{Ij})}$$

$$= -\sum_{j=1}^{N}\frac{\tau_{Ij}\cdot(a^1_{Ij}-a^2_{Ij})^2}{\left[a^2_{Ij}+\theta\cdot(a^1_{Ij}-a^2_{Ij})\right]^2} = -\sum_{j=1}^{N}\frac{\tau_{Ij}\cdot(a^1_{Ij}-a^2_{Ij})^2}{\left[a^3_{Ij}\right]^2}$$

Following the same steps for a_{ij} and a_{iF}, we find:

$$\frac{\partial^2}{\partial\theta^2} Q(\lambda,\lambda^3) = -\sum_{j=1}^{N}\frac{\tau_{ij}\cdot(a^1_{ij}-a^2_{ij})^2}{\left[a^3_{ij}\right]^2} - \frac{\tau_{iF}\cdot(a^1_{iF}-a^2_{iF})^2}{\left[a^3_{iF}\right]^2}$$

We can see that the second derivative in respect to θ is always negative for a_{Ij}, a_{ij}, $a_{iF} \neq 0$, regardless of its direction. Except when $a^1_{Ij} = a^2_{Ij}$ for all j, or $a^1_{ij} = a^2_{ij}$ for all j and $a^1_{iF} = a^2_{iF}$ but then no line exists between λ^1 and λ^2. If there is a critical point on the line between λ^1 and λ^2, this must be a maximum. Because the second derivative is strictly negative, this maximum is unique.

Also, note that this result is independent of the function $P_E(S,\bar{u})$ used in any elementary state. The derivative of $P_E(S,\bar{u})$ for any state S in respect to any parameter a_{ij} of another state is zero. Hence, different training algorithms could be used for different states. When elementary states are trained using another criterion than ML, still a_{ij} can be trained using the ML criterion. Still, for a_{ij}, convergence to a local maximum of the likelihood function is guaranteed, if the training algorithms for the other states also converge.

The mean vector $\bar{\mu}$ and the covariance matrix Σ for elementary states with Gaussian distribution can be trained by determining the values that maximize $Q(\lambda,\hat{\lambda})$. We do not need the Lagrange multiplier method now, because $\bar{\mu}$ has no restriction and Σ will turn out to be positive definite automatically. At least, for new estimates is required that the derivative of the auxiliary function in respect to all parameters is zero. For these calculations the derivative of log $P_E(S,\bar{u})$ to all its parameters is needed. For the definition of $P_E(S,\bar{u})$ see (5.22).

$$\log P_E(S,\bar{u}) = \frac{1}{2}\left(\log |\Sigma|^{-1} - N\cdot\log(2\pi) - (\bar{u}-\bar{\mu})^*\cdot\Sigma^{-1}\cdot(\bar{u}-\bar{\mu}) \right)$$

$$\Rightarrow \frac{\partial}{\partial\mu_i}\log P_E(S,\bar{u}) = \frac{1}{2}\left(\bar{e}_i^*\cdot\Sigma^{-1}\cdot(\bar{u}-\bar{\mu}) + (\bar{u}-\bar{\mu})^*\cdot\Sigma^{-1}\cdot\bar{e}_i \right)$$

$$= \bar{e}_i^*\cdot\Sigma^{-1}\cdot(\bar{u}-\bar{\mu}) \qquad \text{where } \bar{e}_i = \text{the } i^{th} \text{ unit vector.}$$

This is the ith element of $\Sigma^{-1}\cdot(\bar{u}-\bar{\mu})$. If $\partial/\partial\bar{\mu}$ denotes the vector with elements $\partial/\partial\mu_i$, we can write this as:

$$\frac{\partial}{\partial\bar{\mu}} \log P_E(S,\bar{u}) = \Sigma^{-1}\cdot(\bar{u}-\bar{\mu})$$

Defining $C = \Sigma^{-1}$, we must recall that:

$$|C| = \sum_i C_{ij}\cdot B_{ij}$$

for any column j, where B_{ij} is the cofactor associated with C_{ij}. Therefore, we can write:

$$\frac{\partial}{\partial C_{ij}} \log |C| = |C|^{-1}\cdot\frac{\partial}{\partial C_{ij}}|C| = |C|^{-1}\cdot B_{ij}$$

Because $|C|^{-1}\cdot B_{ij}$ is the $(i,j)^{th}$ element of $C^{-1} = \Sigma$, we can combine this into one equation:

$$\frac{\partial}{\partial C} \log |C| = \Sigma$$

where $\partial/\partial C$ denotes the N x N matrix which $(i,j)^{th}$ element is $\partial/\partial C_{ij}$. Now we can calculate the derivative of log $P_E(S,\bar{u})$ to C:

$$\frac{\partial}{\partial C} \log P_E(S,\bar{u}) = \frac{1}{2}\left(\Sigma - (\bar{u}-\bar{\mu})\cdot(\bar{u}-\bar{\mu})^* \right)$$

New estimates of $\bar{\mu}$ can be determined by solving:

$$0 = \frac{\partial}{\partial\hat{\mu}} Q(\lambda,\hat{\lambda}) = \frac{1}{2}\sum_{k=1}^{K} \frac{\sum_{s} P_\lambda(R^k,Y^k,s)\cdot\sum_{t\in s}\hat{\Sigma}^{-1}\cdot(\bar{y}_t-\hat{\mu})}{P_\lambda(R^k,Y^k)}$$

By $\partial/\partial\hat{\mu}$ is meant the vector that has $\partial/\partial\hat{\mu}_i$ as its elements. The summation over t is taken for all times that state S is visited. Multiplying both sides with $2\cdot\hat{\Sigma}$ and interchanging the summations yields:

$$0 = \sum_{k=1}^{K} \sum_{t=1}^{T} \frac{\sum_{s_t \ni S} P_\lambda(R^k,Y^k,s)}{P_\lambda(R^k,Y^k)} \cdot (\bar{y}_t-\hat{\mu})$$

In this fraction we recognize the weight factor w_t as defined in chapter 5, so we can write:

$$0 = \sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot\bar{y}_t - \hat{\mu}\cdot \sum_{k=1}^{K} \sum_{t=1}^{T} w_t \quad\Rightarrow\quad \hat{\mu} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot\bar{y}_t}{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t}$$

Again, this is identical to equation (5.25), except that for multiple training sequences the individual contributions have to be added.

If we define $\hat{C} = \hat{\Sigma}^{-1}$, then we can write:

$$0 = \frac{\partial}{\partial\hat{C}} Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_{s} P_\lambda(R^k,Y^k,s) \cdot \sum_{t\in S} \frac{1}{2}\left(\hat{\Sigma} - (\bar{y}_t-\hat{\mu})(\bar{y}_t-\hat{\mu})^*\right)}{P_\lambda(R^k,Y^k)}$$

Where $\partial/\partial\hat{C}$ stands for the matrix with elements $\partial/\partial\hat{C}_{ij}$.
Interchanging the summation and multiplying by 2, we find:

$$0 = \sum_{k=1}^{K} \sum_{t=1}^{T} \frac{\sum_{s_t \ni S} P_\lambda(R^k,Y^k,s)}{P_\lambda(R^k,Y^k)} \cdot \left(\hat{\Sigma} - (\bar{y}_t-\hat{\mu})(\bar{y}_t-\hat{\mu})^*\right) =$$

$$\hat{\Sigma}\cdot \sum_{k=1}^{K} \sum_{t=1}^{T} w_t - \sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot(\bar{y}_t-\hat{\mu})(\bar{y}_t-\hat{\mu})^* \quad\Rightarrow$$

$$\hat{\Sigma} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot(\bar{y}_t-\hat{\mu})(\bar{y}_t-\hat{\mu})^*}{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot\bar{y}_t\cdot\bar{y}_t^*}{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t} - \hat{\mu}\cdot\hat{\mu}^*$$

This is, again, the same as in (5.26). Note that automatically $\hat{\Sigma}$ is positive definite, because for any vector $\bar{x}$:

$$\bar{x}^*\cdot\hat{\Sigma}\cdot\bar{x} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t\cdot\left(\bar{x}^*\cdot(\bar{y}_t-\hat{\mu}) \right)^2}{\sum_{k=1}^{K} \sum_{t=1}^{T} w_t} \geq 0$$

For the elements of the mean vector $\hat{\mu}$ and the matrix $\hat{\Sigma}$ we can calculate the second derivative to each parameter. For this we also need the second derivative of log $P_E(S,\bar{u})$:

$$\frac{\partial^2}{\partial\mu_i^2} \log P_E(S,\bar{u}) = \frac{\partial}{\partial\mu_i} \bar{e}_i^* \cdot \Sigma^{-1} \cdot (\bar{u}-\bar{\mu}) = - \bar{e}_i^* \cdot \Sigma^{-1} \cdot \bar{e}_i = - (\Sigma^{-1})_{ii}$$

where $\bar{e}_i$ = the ith unit vector.

So:

$$\frac{\partial^2}{\partial\bar{\mu}^2} \log P_E(S,\bar{u}) = - \operatorname{diag}(\Sigma^{-1}) \Rightarrow$$

$$\frac{\partial^2}{\partial\hat{\mu}^2} Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_{s} P_\lambda(R^k,Y^k,s) \cdot \sum_{\iota\in s} - \operatorname{diag}(\hat{\Sigma}^{-1})}{P_\lambda(R^k,Y^k)}$$

$$= - \operatorname{diag}(\hat{\Sigma}^{-1}) \cdot \sum_{k=1}^{K} \sum_{t=1}^{T} w_t = - \tau \cdot \operatorname{diag}(\hat{\Sigma}^{-1})$$

Because $\hat{\Sigma}^{-1}$ is positive definite, this is strictly negative. At the critical point the derivative is zero, so this must be an absolute maximum of $Q(\lambda,\hat{\lambda})$. The same can be done for $C = \Sigma^{-1}$:

$$\frac{\partial^2}{\partial C_{ij}^2} \log |C| = \frac{\partial}{\partial C_{ij}} \left(\hat{\Sigma}_{ij} \right) = \frac{\partial}{\partial C_{ij}} \frac{B_{ij}}{|C|} = = \frac{\partial}{\partial C_{ij}} \frac{B_{ij}}{\sum_k C_{kj} \cdot B_{kj}}$$

$$= - \left(\frac{B_{ij}}{\sum_k C_{kj} \cdot B_{kj}} \right)^2 = - \left(\frac{B_{ij}}{|C|} \right)^2 = - \left(\Sigma_{ij} \right)^2 \Rightarrow$$

$$\frac{\partial^2}{\partial C_{ij}^2} \log P_E(S,\bar{u}) = - \frac{1}{2} \cdot \left(\Sigma_{ij} \right)^2 \Rightarrow$$

$$\frac{\partial^2}{\partial \hat{C}_{ij}^2} Q(\lambda,\hat{\lambda}) = \sum_{k=1}^{K} \frac{\sum_{s} P_\lambda(R^k,Y^k,s) \cdot - \frac{1}{2} \left(\hat{\Sigma}_{ij} \right)^2}{P_\lambda(R^k,Y^k)} = - \frac{\tau}{2} \cdot \left(\hat{\Sigma}_{ij} \right)^2$$

Again, we see that this is always negative or zero.

The conclusion of all this is that the given re-estimated parameters $\hat{\lambda}$ are a maximum of the auxiliary function, and this maximum is unique. After sufficient iterations $\hat{\lambda}$ will converge to a solution where the likelihood does not increase any more. In this way a local maximum of the likelihood function can be reached.

APPENDIX B: RMM-DEFINITIONS

```
/*                                                          */
/*        RMM.H              common definitions             */
/*                                                          */

#include <stdio.h>
#include <math.h>

#define T_ELEM          0
#define T_FULL          1
#define T_PARA          2
#define T_FORW          3
#define T_TYPE          7

#define T_UNIT          8
#define T_TRAIN         16

#define VEC_LEN         5
#define MAX_CHILDS      50

typedef struct m_model{ int type;
                        int nchild;
                        char *name;
                        struct m_prob *empty;
                        struct v_prob *vprob;
                        struct m_model **child;
                        float *parm;
                        float *parmupd;
                        float entropy;
                        float duration;
                      } M_MODEL;

typedef struct m_prob { float ip;
                        float sp;
                        float fp;
                        struct m_prob *ch;
                      } M_PROB;

typedef struct v_prob { float ip;
                        float sp;
                        float fp;
                        struct v_prob *ch;
                        struct v_tree *vtree,*vprev;
                      } V_PROB;

typedef struct v_tree { int nref;
                        struct v_tree *prev;
                        struct m_model *model;
                      } V_TREE;
```

```
/*      FUNCTIONS IN BUILDMM.C          */
M_MODEL*find_model(char *name);
int     read_model(char *filename);
int     write_model(char *filename);
void    write_name(FILE *f, M_MODEL *model);
double  read_num(FILE *f);
double  set_dsymbol(int sym);
double  set_csymbol(float *vector);
int     num_vecs(void);

/*      FUNCTIONS IN CRTEST.C           */
void    read_dseq(FILE *file, char *sequence);
int     read_cseq(FILE *file, char *sequence,float *csequence);
void    outp_dmm( M_MODEL *model,FILE *output);
void    outp_cmm( M_MODEL *model,FILE *output);

/*      FUNCTIONS IN FORBACK.C          */
void    update_model(void);
void    mprob_diagn(void);
double  d_forward(M_MODEL *mm,M_PROB *mpr,char *sentence);
double  c_forward(M_MODEL *mm,M_PROB *mpr,float *f,int n);
double  d_backward(M_MODEL *mm,M_PROB *mpr,char *sentence);
double  c_backward(M_MODEL *mm,M_PROB *mpr,float *f,int n);

/*      FUNCTIONS IN VITERBI.C          */
void    fprf_vprob(FILE *file,int t,M_MODEL *mm,V_PROB *mp);
void    fprf_maxvprob(FILE *file,int t,M_MODEL *mm,V_PROB *mp);
V_PROB *new_vprob(V_PROB *mp,M_MODEL *mm);
int     free_vprob(V_PROB *mp,M_MODEL *mm);
void    vprob_diagn(void);
double  d_viterbi(M_MODEL *mm,V_PROB *mpv,char *sentence,FILE *f);
double  c_viterbi(M_MODEL *mm,V_PROB *mpv,float *csequence,int n,FILE *f);
double  build_viterbi(M_MODEL *mm,V_PROB *mpv,V_TREE *vtree);
double  calc_viterbi(M_MODEL *mm,V_PROB *mpv,V_TREE *vtree);

/*      FUNCTIONS IN ENTROPY.C          */
void    calc_entropy(M_MODEL *mm);
```

APPENDIX C: GRAMMARS

Content of "word.rmm" (partly):

```
a       train
        1       0       0       0
        0.2     0.2     0.2     0.2

b       train
        2       0       0       0
        0.2     0.2     0.2     0.2

c       train
        3       0       0       0
        0.2     0.2     0.2     0.2

d       train
        4       0       0       0
        0.2     0.2     0.2     0.2
```

< the same for other ASCII-characters >

```
a-e     para
a       1/5
b       1/5
c       1/5
d       1/5
e       1/5

f-j     para
f       1/5
g       1/5
h       1/5
i       1/5
j       1/5

k-o     para
k       1/5
l       1/5
m       1/5
n       1/5
o       1/5

p-u     para
p       1/6
q       1/6
r       1/6
s       1/6
t       1/6
u       1/6
```

v-#	para
v	1/6
w	1/6
x	1/6
y	1/6
z	1/6
' '	1/6

anychar	para
a-e	5/27
f-j	5/27
k-o	5/27
p-u	6/27
v-#	6/27

A	para
a	0.98
anychar	0.02

B	para
b	0.98
anychar	0.02

C	para
c	0.98
anychar	0.02

D	para
d	0.98
anychar	0.02

< the same for other ASCII-characters >

0	forw	unit	train
		0.98	0.02
Z	0.02	0.96	0.02
E	0.02	0.96	0.02
R	0.02	0.96	0.02
O	0.02	0.98	

1	forw	unit	train
		0.98	0.02
O	0.02	0.96	0.02
N	0.02	0.96	0.02
E	0.02	0.98	

2	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
W	0.02	0.96	0.02
O	0.02	0.98	

3	forw	unit	train
	0.98	0.02	
T	0.02	0.96	0.02
H	0.02	0.96	0.02
R	0.02	0.96	0.02
E	0.02	0.96	0.02
E	0.02	0.98	

4	forw	unit	train
		0.98	0.02
F	0.02	0.96	0.02
O	0.02	0.96	0.02
U	0.02	0.96	0.02
R	0.02	0.98	

5	forw	unit	train
		0.98	0.02
F	0.02	0.96	0.02
I	0.02	0.96	0.02
V	0.02	0.96	0.02
E	0.02	0.98	

6	forw	unit	train
		0.98	0.02
S	0.02	0.96	0.02
I	0.02	0.96	0.02
X	0.02	0.98	

7	forw	unit	train
		0.98	0.02
S	0.02	0.96	0.02
E	0.02	0.96	0.02
V	0.02	0.96	0.02
E	0.02	0.96	0.02
N	0.02	0.98	

8	forw	unit	train
		1	
eigh	0	0.98	0.02
T	0.02	0.98	

9	forw	unit	train
	0.98	0.02	
N	0.02	0.96	0.02
I	0.02	0.96	0.02
N	0.02	0.96	0.02
E	0.02	0.98	

10	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
E	0.02	0.96	0.02
N	0.02	0.98	

11	forw	unit	train
		0.98	0.02
E	0.02	0.96	0.02
L	0.02	0.96	0.02
E	0.02	0.96	0.02
V	0.02	0.96	0.02
E	0.02	0.96	0.02
N	0.02	0.98	

12	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
W	0.02	0.96	0.02
E	0.02	0.96	0.02
L	0.02	0.96	0.02
V	0.02	0.96	0.02
E	0.02	0.98	

thir	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
H	0.02	0.96	0.02
I	0.02	0.96	0.02
R	0.02	0.98	

fif	forw	unit	train
		0.98	0.02
F	0.02	0.96	0.02
I	0.02	0.96	0.02
F	0.02	0.98	

eigh	forw	unit	train
		0.98	0.02
E	0.02	0.96	0.02
I	0.02	0.96	0.02
G	0.02	0.96	0.02
H	0.02	0.98	

teen	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
E	0.02	0.96	0.02
E	0.02	0.96	0.02
N	0.02	0.98	

twen	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
W	0.02	0.96	0.02
E	0.02	0.96	0.02
N	0.02	0.98	

ty	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
Y	0.02	0.98	

100	forw	unit	train
		0.98	0.02
H	0.02	0.96	0.02
U	0.02	0.96	0.02
N	0.02	0.96	0.02
D	0.02	0.96	0.02
R	0.02	0.96	0.02
E	0.02	0.96	0.02
D	0.02	0.98	

1000	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
H	0.02	0.96	0.02
O	0.02	0.96	0.02
U	0.02	0.96	0.02
S	0.02	0.96	0.02
A	0.02	0.96	0.02
N	0.02	0.96	0.02
D	0.02	0.98	

&	forw	unit	train
		0.98	0.02
A	0.02	0.96	0.02
N	0.02	0.96	0.02
D	0.02	0.98	

that	forw	unit	train
		0.98	0.02
T	0.02	0.96	0.02
H	0.02	0.96	0.02
A	0.02	0.96	0.02
T	0.02	0.98	

what	forw	unit	train
		0.98	0.02
W	0.02	0.96	0.02
H	0.02	0.96	0.02
A	0.02	0.96	0.02
T	0.02	0.98	

< and in the same way a lot of other words are contained >

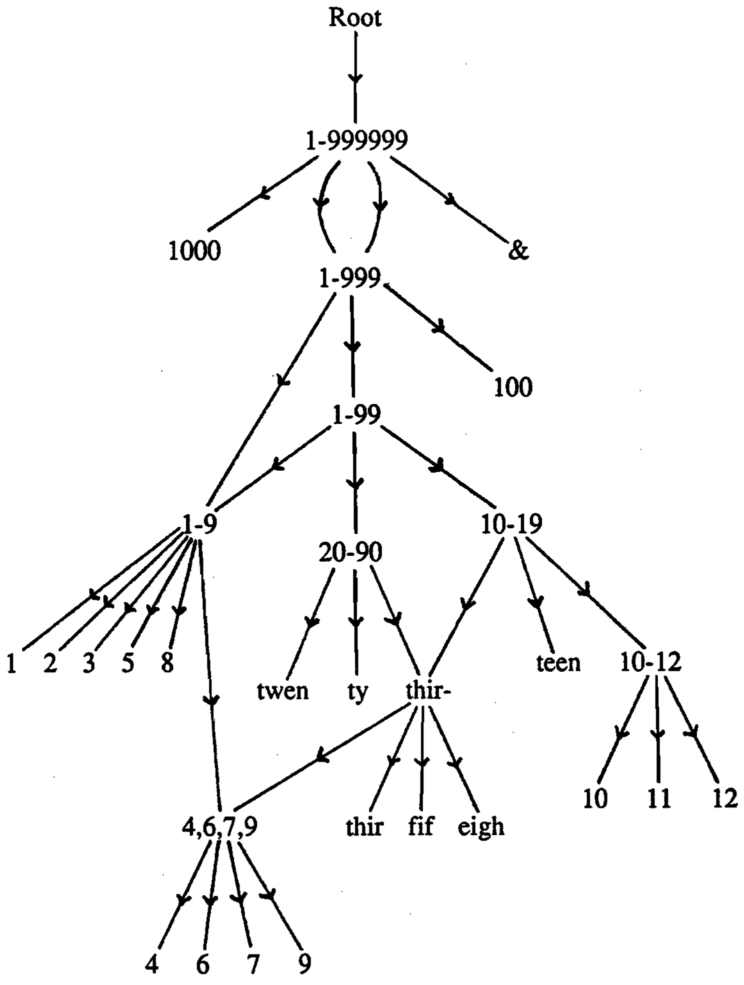
Content of "grammar.rmm" (complete):

```
4,6,7,9     para    train
4           1/4
6           1/4
7           1/4
9           1/4

thir-9      para    train
thir        1/7
fif         1/7
eigh        1/7
4,6,7,9     4/7

1-9         para    train
1           1/9
2           1/9
3           1/9
5           1/9
8           1/9
4,6,7,9     4/9

10-12       para    train
10          1/3
11          1/3
12          1/3

10-19       forw    train
                       0.7        0.3
thir-9      0          0          1
10-12       0          0          1
teen        0          1

20-90       forw    train
            1/8        7/8
twen        0          0          1
thir-9      0          1
ty          0          1

1-99        full    train
            20-90      10-19      1-9
            0.4        0.1        0.5
20-90       0          0          0.9        0.1
10-19       0          0          0          1
1-9         0          0          0          1

1-999       full    train
            1-9        100        1-99
            0.3        0.1        0.6
1-9         0          1
100         0          0          0.9        0.1
1-99        0          0          0          1
```

1-999999	full	train			
	1-999	1000	&	1-999	
	1				
1-999	0	0.5	0	0	0.5
1000	0	0	0.98	0	0.02
&	0	0	0	1	
1-999	0	0	0	0	1

Root	para
1-999999	1

Content of "grammar.rmm" as a directed graph:



Content of "scr.rmm" (complete):

```
integer    para   train
1          1/9
2          1/9
3          1/9
4          1/9
5          1/9
6          1/9
7          1/9
8          1/9
9          1/9

2_int      forw   train
                    1
integer    0        1/2      1/2
integer    0        1

3_int      forw   train
                    1
2_int      0        1/3      2/3
integer    0        1

1_digit    para   train
0          0.1
integer    0.9

2_digits   para   train
0          0.03
2_int      0.97

3_digits   para   train
0          0.01
3_int      0.99

obj_descr         para   train
SIZE              1/6
HEIGHT            1/6
TEXTURE           1/6
LUSTER            1/6
WEIGHT            1/6
COLOR             1/6

obj_description   full   train
                  &        obj_descr
                  0        1
&                 0        1
obj_descr         0.3      0.3      0.4

that_is_obj       forw
                           1
that              0        1
is                0        1
obj_description   0        1
```

OBJECT	forw	train	
		0.5	0.5
obj_description	0	1	
SHAPE	0	0.5	0.5
that_is_obj	0	1	

that_are_obj	forw	
		1
that	0	1
are	0	1
obj_description	0	1

OBJECTS	forw	train	
		0.5	0.5
obj_description	0	1	
SHAPES	0	0.5	0.5
that_are_obj	0	1	

left_right	para	train
left	0.5	
right	0.5	

front_back	para	train
front	0.5	
back	0.5	

POSITION	para	train
to_the_lrfb	0.5	
in_fb	0.5	

to_the_lrfb	forw	train	
		1	
to	0	1	
the	0	0.5	0.5
left_right	0	0	1
front_back	0	1	

in_fb	forw	train
		1
in	0	1
front_back	0	1

SIZE	para	train
large	1/3	
medium	1/3	
small	1/3	

HEIGHT	para	train
tall	1/2	
short	1/2	

COLOR	para	train
white	1/4	
black	1/4	
light_dark_col	1/2	

light_dark_col	forw	train
	0.5	0.5
light_dark	0	1
other_col	0	1

light_dark	para	train
light	1/2	
dark	1/2	

other_col	para	train
gray	1/6	
red	1/6	
blue	1/6	
green	1/6	
yellow	1/6	
brown	1/6	

SHAPE	para	train
can	1/13	
glass	1/13	
bottle	1/13	
insulator	1/13	
beaker	1/13	
flask	1/13	
cone	1/13	
pyramid	1/13	
prism	1/13	
cup	1/13	
block	1/13	
cylinder	1/13	
object	1/13	

SHAPES	para	train
cans	1/13	
glasses	1/13	
bottles	1/13	
insulators	1/13	
beakers	1/13	
flasks	1/13	
cones	1/13	
pyramids	1/13	
prisms	1/13	
cups	1/13	
blocks	1/13	
cylinders	1/13	
objects	1/13	

NAME	para	train
fred	1/4	
alice	1/4	
mary	1/4	
joe	1/4	

TEXTURE	para	train
smooth	1/2	
rough	1/2	

LUSTER	para	train
shiny	1/3	
matte	1/3	
dull	1/3	

WEIGHT	para	train
heavy	1/2	
light	1/2	

ATTRIBUTE	para	train
size	1/8	
height	1/8	
color	1/8	
shape	1/8	
name	1/8	
texture	1/8	
luster	1/8	
weight	1/8	

find_obj	forw	train	
		1	
find	0	0.5	0.5
all_of_obj	0	0	1
a_an_another_obj	0	1	

all_of_obj	forw	
		1
all	0	1
of	0	1
the	0	1
OBJECTS	0	1

a_an_another_obj	forw	
		1
a_an_another	0	1
OBJECT	0	1

a_an_another	para	train
a	1/5	
an	1/5	
another	1/5	
the_next	1/5	
a_new	1/5	

the_next	forw	
		1
the	0	1
next	0	1

a_new	forw	
		1
a	0	1
new	0	1

move_obj	forw	train
	0.5	0.5
can_you	0	1
move	0	1
it_this_object_robot	0	1
move_dest	0	1

can_you	forw	
		1
can	0	1
you	0	1

it_this_object_robot	para	train
it	1/5	
this	1/5	
NAME	1/5	
the_a_an_obj	1/5	
the_robot	1/5	

the_a_an_obj	forw	
	1	
the_a_an	0	1
OBJECT	0	1

the_a_an	para	train
the	1/3	
a	1/3	
an	1/3	

the_robot		forw
		1
the	0	1
robot	0	1

move_dest	para	train
position_of	1/3	
to_the_origin	1/3	
dist_along_axis	1/3	

position_of	forw	
		1
POSITION	0	1
of	0	1
it_this_object_robot	0	1

to_the_origin	forw	
		1
to	0	1
the	0	1
origin	0	1

dist_along_axis	forw	train	
		1	
2_digits	0	1	
centimeter_s	0	1	
along	0	1	
the	0	0.5	0.5
posi_negative	0	1	
x_y_z	0	1	
axis	0	1	

centimeter_s	para	train
centimeter	1/2	
centimeters	1/2	

posi_negative	para	train
positive	1/2	
negative	1/2	

x_y_z	para	train
x	1/3	
y	1/3	
z	1/3	

rotate_obj	forw	
		1
rotate	0	1
it_this_object_robot	0	1
3_digits	0	1
degree_s	0	1
counter_clockwise	0	1

degree_s	para	train
degree	1/2	
degrees	1/2	

counter_clockwise	para	train
clockwise	1/2	
counterclockwise	1/2	

what_attr_is	forw	train	
		1	
what	0	0.5	0.5
ATTRIBUTE	0	1	
is	0	1	
it_this_object_robot	0	1	

is_it_?	forw	
		1
is	0	1
it_this_object_robot	0	1
obj_pos_descr	0	1

obj_pos_descr	para	train
a_an_obj	1/3	
position_of	1/3	
obj_description	1/3	

a_an_obj	forw	train	
		0.5	0.5
a	0	0	1
an	0	1	
OBJECT	0	1	

this_where_is	forw	
		1
this_where	0	1
is	0	1
it_this_object_robot	0	1

this_where	para	train
this	1/2	
where	1/2	

what_howmany_objforw		
		1
what_how_many	0	1
there_descr	0	1

what_how_many	para	train
what_obj_is	1/2	
what_obj_are	1/2	

what_obj_is	forw	
		1
what	0	1
OBJECT	0	1
is	0	1

what_obj_are	forw	
		1
what_howmany	0	1
OBJECTS	0	1
are	0	1

what_howmany	para	train
what	1/2	
how_many	1/2	

how_manyforw			
		1	
how	0	1	
many	0	1	

there_descr	para	train
there	1/4	
obj_description	1/4	
position_of	1/4	
in_location	1/4	

in_location	forw	
		1
in	0	1
location	0	1
2_digits	0	1

store_obj	forw	train	
		1	
store_showme	0	1	
it_this_object_s	0	0.5	0.5
that_is_pos_of	0	1	

store_showme	para	train
store	1/5	
remove	1/5	
delete	1/5	
describe	1/5	
show_me	1/5	

show_me	forw	
		1
show	0	1
me	0	1

it_this_object_s	para	train
it	1/4	
this	1/4	
NAME	1/4	
the_objects_s	1/4	

the_objects_s	forw	train	
		1	
the	0	0.5	0.5
object	0	0	1
objects	0	1	

that_is_pos_of	forw	
		1
that	0	1
is	0	1
POSITION	0	1
of	0	1
it_this_object_robot	0	1

noun_adj	para	train
noun	1/2	
adjective	1/2	

reset_the_robot	forw	
		1
reset	0	1
the	0	1
robot	0	1

yes_no	para	train
yes	1/2	
no	1/2	

ignore_it	forw	train	
		1	
ignore	0	0.5	0.5
it_this_sentence	0	1	

it_this_sentence	para	train
it	0.5	
this_sentence	0.5	

this_sentence	forw	train	
		1	
this	0	0.5	0.5
sentence	0		

Root	para	train
find_obj	1/14	
move_obj	1/14	
rotate_obj	1/14	
what_attr_is	1/14	
is_it_?	1/14	
this_where_is	1/14	
what_howmany_obj	1/14	
store_obj	1/14	
the_a_an_obj	1/14	
noun_adj	1/14	
ATTRIBUTE	1/14	
reset_the_robot	1/14	
yes_no	1/14	
ignore_it	1/14	

REFERENCES

[1] Atal,B.S. and Hanauer,S.L. (1971). 'Speech Analysis and Synthesis by Linear Prediction of the Speech Wave', J. Acoust. Soc. Amer., Vol.50, pp.637-655.

[2] Aubert,X., Bourlard,H., Kamp,Y., and Wellekens,C.J., (1988). 'Improved Hidden Markov Models for Speech Recognition', Philips Journal of Research 43, pp.224-245.

[3] Austin,S.C. and Fallside,F. (1988). 'Frame Compression in Hidden Markov Models', Proc. ICASSP 1988, pp.477-480.

[4] Averbuch,A et al. (1986). 'An IBM-PC Based Large Vocabulary Isolated Utterance Speech Recognizer', Proc. ICASSP 1986, pp.53-56.

[5] Baker,J.K. (1975). 'The DRAGON system - an overview', IEEE Trans. ASSP. Vol.23, pp.24-29.

[6] Bahl,L.R. et al. (1988). 'Speech Recognition with Continuous Parameter Hidden Markov Models', Proc. ICASSP 1988, pp.40-43.

[7] Bahl,L.R. et al. (1988). 'A New Algorithm for the Estimation of Hidden Markov Model Parameters', Proc. ICASSP 1988, pp.493-496.

[8] Bahl,L.R. et al. (1988). 'Acoustic Markov Models used in the Tangora Speech Recognition System', Proc. ICASSP 1988, pp.497-500.

[9] Baum,L.E. et al. (1970). 'A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains', Ann. Math. Stat. vol.41, pp.164-170.

[10] Baum,L.E. (1972). 'An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes', Inequalities III, pp.1-8.

[11] Brown,M.K. and Wilpon,J.G. (1991). 'A Grammar Compiler for Connected Speech Recognition', IEEE Trans. Signal Processing, Vol.39, no.1, pp.17-28.

[12] Carey,M.J. et al. (1988). 'A Speaker-Independent Speech Recogniser'. SRS1-E papers, Ensigma Ltd., Chepstow, Gwent. U.K.

[13] Chang,L. and Bayoumi,M.M. (1990). 'Parametric Modelling of State Transitions in Hidden Markov Model', Signal Processing V: Theories and Applications. Elsevier Science Publishers B.V., pp.1387-1390.

[14] Delsarte,D. and Genin,Y.V. (1986). 'The Split Levinson Algorithm', IEEE Trans. ASSP, vol.34, no.3, pp.470-478.

[15] Delsarte, D. and Genin,Y.V. (1988). 'A Survey of the Split Approach based Techniques in Digital Signal Processing Applications', Philips Journal of Research 43, pp.346-374.

[16] D'Orta,P et al. (1988). 'Large Vocabulary Speech Recognition: A System for the Italian Language', IBM J. Res. Develop. vol.32, no.2, pp.217-226.

[17] Ensigma SRS1-E data sheets (1988), Ensigma Ltd. Archway House, Welch St. Chepstow, Gwent. NP6 5LL. U.K.

[18] Gray,A.H. and Markel,J.D. (1976), 'Quantization and Bit Allocation in Speech Processing', IEEE Trans. ASSP, vol.24, pp.459-473.

[19] Hemert,J.P. van, 'Automatische Segmentering van Spraak in difonen', Philips Techn. T, nr 9, pp.249-258. (In Dutch)

[20] Holmes,J.N. (1988). *Speech Synthesis and Recognition*, Van Nostrand Reinhold (UK) Co.Ltd.

[21] Itakura, F. (1975). 'Minimum Prediction Residual Principle applied to Speech Recognition', IEEE Trans. ASSP, vol.23, pp.67-72.

[22] Janssen,C.H.H. (1988). *Spraakherkenning gebaseerd op Stochastische Modellering van Spraakproductie*, Master Thesis, Eindhoven Univ. of Technology, August 1988. (In Dutch)

[23] Jelinek,F. (1985). 'The Development of an Experimental Discrete Dictation Recognizer', Proc. IEEE, vol.73, no.11, pp.1616-1624.

[24] Kenny,P., Lennig,M. and Mermelstein,P. (1990). 'A Linear Predictive HMM for Vector Valued Observations with Applications to Speech Recognition', IEEE Trans. ASSP, vol.38, no.2, pp.220-225.

[25] Kerninghan,B.W. and Ritchie,D.M. (1988) *The C programming language*, second edition, Prentice Hall.

[26] Kingsbury,N.G. and Rayner,P.J.W. (1971). 'Digital filtering using logarithmic arithmetic. Electron. Lett. 7, pp. 56-58.

[27] Komo,J.J.,(1987). *Random Signal Analysis in Engineering Systems*, Academic Press, inc., Orlando, Florida.

[28] Lea,W.A., (1980). *Trends in Speech Recognition*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632.

[29] Lee,C and Rabiner,L.R. (1988). 'A Frame Synchronous Level Building (FSLB) Algorithm for Connected Word Recognition', Proc. IEEE ICASSP '88, pp.410-413.

[30] Lee,K.F., (1988). *Automatic Speech Recognition: The Development of the SPHINX System*, Boston, MA: Kluwer Academic.

[31] Lee,K.F. and Hon,H.W. (1989). 'Speaker Independent Phone Recognition using Hidden Markov Models', IEEE Trans. ASSP, vol.37, no.11, pp.1641-1648.

[32] Lee,K.F., Hon,H.W. and Reddy,R (1990). 'An Overview of the SPHINX Speech Recognition System', IEEE Trans. ASSP, vol.38, no.1, pp.35-45.

[33] Lee,K.F., (1990). 'Context-Dependent Phonetic Hidden Markov Models for Speaker-Independent Continuous Speech Recognition', IEEE Trans. ASSP, vol.38, no.4, pp.599-609.

[34] Levinson,S.E., (1985). 'Structural Methods in ASR', Proc. of the IEEE, vol. 73, no. 11, pp.1625-1650.

[35] Levinson,S.E., Rabiner,L.R., et al. (1985). 'An Introduction to the Applications of the Theory of Probabilistic Functions of a Markov Process to ASR', The Bell Techn. Journ., vol 62, no. 4, pp.1035-1650.

[36] Levinson, S.E. (1986). 'Continuously Variable Duration Hidden Markov Models for Automatic Speech Recognition', Computer Speech and Language, no. 1, pp.29-45.

[37] Levinson, S.E. (1986). 'Continuously Variable Duration Hidden Markov Models for Speech Analysis', Proc. ICASSP 1986, pp.1241-1244

[38] Levinson,S.E., Ljolje,A. and Miller,L.G. (1988). 'Large Vocabulary Speech Recognition using a Hidden Markov Model for Acoustic-Phonetic Classification', Proc. ICASSP 1988, pp.505-508

[39] Liporace,L.A. (1982). 'Maximum Likelihood Estimation for Multivariate Observations of Markov Sources', IEEE Trans. Information theory, vol. 28, no. 5, pp.729-734.

[40] Ljolje,A. and Levinson,S.E. (1991). 'Development of an Acoustic-Phonetic Hidden Markov Model for Continuous Speech Recognition', IEEE Trans. Signal Process. vol.39, January 1991, pp.29-39.

[41] Makhoul,J. (1975). 'Linear Prediction: A Tutorial Review', Proc. IEEE, vol.63, no.4, pp.561-580.

[42] Mariani,J. (1989). 'Recent Advances in Speech Processing', Proc. ICASSP 1989, pp.429-440.

[43] Mariño,J.B. et al. (1990). 'Recognition of Numbers by using Demisyllables and Hidden Markov Models', Signal Processing V: Theories and Applications. Elsevier Science Publishers B.V., pp.1363-1366.

[44] Mérialdo,B. (1988). 'Multilevel Decoding for Very Large Size Dictionary Speech Recognition'. IBM J. Res. Develop. vol.32, no.2, pp.227-237.

[45] Nijtmans,J.J. (1987). *Spraak-Analyse en Resynthese m.b.v. Lattice-Filters*, Master Thesis, Eindhoven Univ. of Technology, March 1987. (In Dutch)

[46] Nijtmans,J.J. (1991). 'A New Recursive Markov Model with a New State Pruning Approach for Large Vocabulary Speech Recognition', European Conference on Speech Technology, Eurospeech 91, Genova, pp.659-662.

[47] Niles,L.T. and Silverman,H.F. (1990). 'Combining Hidden Markov Model and Neural Network classifiers', Proc. ICASSP 1990, pp.417-420.

[48] Picone,J. (1990). 'Continuous Speech Recognition using Hidden Markov Models', IEEE ASSP magazine, July 1990, pp.26-41.

[49] Poritz,A.B. (1988). 'Hidden Markov Models: A Guided Tour', PROC. ICASSP 1988, pp.7-13.

[50] Rabiner,L.R. and Juang,B.H. (1986). 'An Introduction to Hidden Markov Models', IEEE ASSP magazine vol.3, no.1, pp.4-16.

[51] Rabiner,L.R. and Levinson,S.E. (1981). 'Isolated and Connected Word Recognition - Theory and Selected Applications', IEEE Trans. on Communications, vol.29, no.5, pp.621-659.

[52] Rabiner,L.R. and Schafer, R.W. (1978). *Digital Processing of Speech Signals*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey.

[53] Russell,M.J. and Moore,R.K. (1985). 'Explicit Modeling of State Occupancy in Hidden Markov Models for Automatic Speech Recognition', Proc. IEEE ICASSP'85, pp.5-8.

[54] Shannon,C.E. and Weaver,W. (1964). *The Mathematical Theory of Communication*. Urbana, IL. Univ. of Illinois Press.

[55] Silverman,H.F. and Morgan,D.P. (1990). 'The Application of Dynamic Programming to Connected Speech Recognition', IEEE ASSP Magazine, July 1990, pp.7-25.

[56] Sondhi,M.M. and Levinson S.E. (1978). 'Computing Relative Redundancy to Measure Grammatical Constraint in Speech Recognition Tasks', Proc. IEEE ICASSP '78, pp.409-412.

[57] Streit,R.L. (1990). 'The Moments of Matched and Mismatched Hidden Markov Models'. IEEE Trans. ASSP, vol.38, no.4, pp.610-622.

[58] Strobach,P. (1991). 'New Forms of Levinson and Schur Algorithms', IEEE SP Magazine, January 1991, pp.12-36.

[59] Texas Instruments (1987). *TMS3201x/TMS320C2x Assembly Language Tools User's Guide*, Houston, U.S.A.

[60] Texas Instruments (1988). *TMS320C2x Software Development System*, Houston, U.S.A.

[61] Texas Instruments (1989). *Second-Generation TMS320 User's Guide*, Houston, U.S.A.

[62] Viswanathan,R. and Makhoul,J. (1975). 'Quantization Properties of Transmission Parameters in Linear Predictive Systems', IEEE Trans. ASSP. vol.23, pp.309-321.

[63] Viterbi,A.J. (1967). 'Error bounds for convolutional codes and an asymptotically optimum decoding algorithm', IEEE Trans. Information theory IT-13, pp.260-269.

ACKNOWLEDGEMENT

It would be unfair not to mention some other names of people who contributed to this thesis. I am most grateful to my first promotor, Wim van Bokhoven, for his continuous support in many different ways. I would like to thank him for giving me the opportunity to carry out this research, for all his valuable stimulating discussions, and for his careful proofreading of earlier versions of this thesis. I would also like to thank my second promotor, Ton Backx, for his invaluable support in the finishing of this thesis.

Many discussions with other group members have been very valuable to me. I would like to mention my colleague Irene Gu, who has been a very nice and encouraging room-mate, and the student Stan Janssen who contributed a great deal of insight to the basic problems in speech recognition. Two other colleagues, Hans Hegt and Piet Sommen, repeatedly triggered my mind with all kinds of useful suggestions and critical remarks.

I am happy about the pleasant years I spent in the Circuit Design group, first as a student, later as research assistant. All of my colleagues and fellow students contributed to the nice atmosphere in the group, which makes that many former colleagues and students still like to pass by whenever they can find some excuse. I am sure that I can find many such 'excuses'.

Furthermore, the help of Heleen Oosterveld concerning the use of the english language has been invaluable to me. Also, without the moral support from my brother and my parents, this work would have been much more difficult to complete. The opportunity I got from my parents to do the study of my own choice, and their support during the past years, have finally been the main condition for the finishing of this thesis. This is more than enough reason to dedicate this thesis to them.

CURRICULUM VITAE

271-

The author of this thesis was born on June 4, 1964 in Oisterwijk, the Netherlands. He graduated at the St. Odulphus Lyceum Tilburg on his 18th birthday (June 4, 1982).
On May 21, 1987 he received the Ir. degree in electrical engineering from the Eindhoven University of Technology. His master report was entitled. 'Speech analysis and re-synthesis by means of Lattice filters'.
From June 1, 1987 to January 1, 1992 he has been working as a research assistant at the Circuit Design group (EEB) in the micro-electronics division, Faculty of Electrical Engineering at the Eindhoven University of Technology.

STELLINGEN

Behorende bij het proefschift
Speech Recognition by Recursive Stochastic Modelling
door J.J. Nijtmans

1 Bij segmentatie van natuurlijke spraak dient nadrukkelijk kennis over de syntax gebruikt te worden, om met voldoende zekerheid de aan- of afwezigheid van segmentgrenzen vast te kunnen stellen.
(Dit proefschrift, hoofdstuk 3)

2 Een toestandsmachine waarbij een toestand meerdere symbolen kan produceren is op geen enkele wijze in conflict met de Markov-keten theorie, daar altijd een nieuwe set van (samengestelde) symbolen gedefinieerd kan worden die direct samenhangt met de toestanden.
(Dit proefschrift, hoofdstuk 5)

3 De term "Hidden Markov Model" moet slechts gezien worden als een definitie, daar het woord "Hidden" triviaal is.
(L.E. Baum, 'An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes', Inequalities III, 1972, pp.1-8)

4 Gebruik van digram of trigram taalmodellen is slechts correct bij talen met een zeer beperkt geheugen, en daarom ongeschikt voor algemene spraakherkenningssystemen (zoals SPHINX, Tangora).
(K.F. Lee, *Automatic Speech Recognition: The Development of the SPHINX System*, Boston, MA: Kluwer Academic, 1988.
F. Jelinek, 'The Development of an Experimental Discrete Dictation Recognizer', Proc. IEEE, vol.73, no.11, 1985, pp.1616-1624.)

5 Gebruik van de entropy van een taal voor evaluatie van de complexiteit van een herkenningsprobleem is alleen correct voor constante woordlengte. Daarom is eigenlijk de entropy per eenheid van lengte (bijv. seconde of meter) een beter criterium.
(M.M. Sondhi and S.E. Levinson, 'Computing Relative Redundancy to Measure Grammatical Constraint in Speech Recognition Tasks', Proc. IEEE ICASSP '78, pp.12-36)

6 Zolang analoge simulatoren zoals SPICE niet in staat zijn ideale schakelaars en ideale op-amps (Nullors) te modelleren, zijn betrouwbare simulaties van niet-ideale componenten zeker niet te verwachten.
(A. Vladimirescu et al, *SPICE Version 2G user's guide*, Dept. of Electrical Engeneering and Computer Sciences, University of California, Berkeley, Ca., 94720.)

7 Computers produceren uit zichzelf geen nieuwe fouten, maar vergroten slechts de gevolgen van menselijke fouten en onvolkomenheden.

8 Afschaffing van de militaire dienstplicht is politiek noodzakelijk, maar economisch onverantwoord.

9 De gelijkzwevende toonschaal is optimaal voor mathematici maar niet voor musici, daar de verschillende toonsoorten niet in gelijke mate in de muziek voorkomen.

10 Zoals een computer in het algemeen geschikt is voor meer dan alleen spelletjes, is een accordeon geschikt voor meer dan alleen volksmuziek.
(Kaoma: Lambada, James Last: Biskaya)