Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Sistema de Reconhecimento Automático de Palavras Isoladas Utilizando Modelos Escondidos de Markov

Bruno de Assis Rolim

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Pedro de Azevedo Berger

Brasília
2008

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Coordenadora: Prof.ª Dr.ª Carla Maria Chagas e Cavalcante Koike

Banca examinadora composta por:

Prof. Dr. Pedro de Azevedo Berger (Orientador) – CIC/UnB
Prof. Dr. Guilherme Albuquerque Pinto – CIC/UnB
Prof. Dr. Carla Maria Chagas e Cavalcante Koike – CIC/UnB

CIP – Catalogação Internacional na Publicação

Rolim, Bruno de Assis.

Implementação de um Sistema de Reconhecimento Automático de Palavras Isoladas Utilizando Modelos Escondidos de Markov / Bruno de Assis Rolim. Brasília : UnB, 2008.
p. : il. ; 29,5 cm.

Monografia (Graduação) – Universidade de Brasília, Brasília, 2008.

1. Modelo de Markov, 2. HMM, 3. reconhecimento, 4. voz, 5. comandos de voz.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910–900
Brasília – DF – Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Sistema de Reconhecimento Automático de Palavras Isoladas Utilizando Modelos Escondidos de Markov

Bruno de Assis Rolim

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Pedro de Azevedo Berger (Orientador)
CIC/UnB

Prof. Dr. Guilherme Albuquerque Pinto
CIC/UnB

Prof. Dr. Carla Maria Chagas e Cavalcante Koike
CIC/UnB

Profª. Drª. Carla Maria Chagas e Cavalcante Koike
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 5 de dezembro de 2008

Resumo

O quadro atual sobre o reconhecimento de comandos de voz ainda não possui nenhum desfecho considerado satisfatório. Esse desafio motivou o desenvolvimento desse trabalho de graduação, que utiliza técnicas já conhecidas como *Hidden Markov Model* (Modelo de Markov Escondido) e coeficintes Mel-cepstrais para implementar o reconhecimento.

O trabalho utiliza uma solução clássica para o reconhecimento de palavras isoladas utilizando como vetor características da voz um conjunto de coeficientes Mel-cepstrais e seus derivados. O sistema desenvolvido tem um caráter bastante modularizado, assim, desde que seja especificado a taxa de amostragem, o tamanho do cabeçalho do arquivo WAVE de entrada e a duração da locução, é possível alterar parâmetros como: o número de estados do HMM, tamanho do vetor de características, tamanho da janela de análise e tamanho do alfabeto.

Analisando os testes realizados, observou-se resultados promissores para o sistema desenvolvido, uma vez que a taxa de reconhecimento correto para o conjunto de treinamento foi de 86,67% para uma configuração com janela de 128 amostras e um alfabeto de 256 vetores de código. Entretanto, as taxas de acerto com a mesma configuração foi em torno de 60% para o conjunto de validação. Análise mais profundas indicam que um algoritmo de extração de silêncio poderia tornar os resultados para o conjunto de validação compatível com os resultados para o conjunto de treinamento.

Palavras-chave: Modelo de Markov, HMM, reconhecimento, voz, comandos de voz.

Sumário

Lista de Figuras

Lista de Símbolos

Símbolos e abreviações do modelo probabilístico (modelo escondido de Markov):

HMM	abreviação para o modelo (do inglês, *Hidden Markov Model*)
Φ	o mesmo que HMM
N	número de estados do modelo
M	tamanho do alfabeto do modelo (número de saídas)
T	tamanho da seqüência de observações
X_i	i-ésimo elemento da observação X, que possui T elementos
S	uma seqüência de estados
Π	matriz de probabilidades de estados iniciais
A	matriz de probabilidades de transição entre estados
B	matriz de probabilidades de uma saída ocorrer dado o estado inicial
α	probabilidade *Forward*
β	probabilidade *Backward*

Símbolos e abreviações do modelo de sinal:

$MFCC$	abreviação para os coeficientes Mel-cepstrais (do inglês, *Mel-Frequency Cepstral Coefficients*)
X	*stream* de áudio janelada
N	tamanho da *stream* de áudio janelada
Y_w	segmento do janelamento
N_w	número de coeficientes de Fourier
W	tamanho do segmento de janelamento
DFT	transformada de Fourier discreta (do inglês, *Discrete Fourier Transform*)

FFT	alforitmo rápido da transformada de Fourier (do inglês, *Fast Fourier Transform*)
DCT	transformada inversa de Fourier (do inglês, *Discrete Cosine Transform*)
M	número de coeficientes da escala Mel
S	uma seqüência de estados
LBG	algoritmo de quantização vetorial - algoritmo de Linde-Buzo-Grey
L	número máximo de células que subdivide o espaço em uma quantização vetorial
T	número de vetores de treinamento em uma quantização vetorial
B	número de centróides no algoritmo LBG
D	distorção média global no algoritmo LBG

Capítulo 1

Introdução

A comunicação entre homem e máquina nos dias de hoje está baseada, quase em sua totalidade, em botões. A procura por canais mais intuitivos é visível em filmes de ficção científica e pesquisas atuais. Uma delas é a fala, que é utilizada na comunicação entre os próprios seres humanos. Conseguir que sistemas possam reconhecer comandos de voz é uma forma de humanizar essa interface entre máquina e homem. E, além disso, também é uma maneira de realizar a inclusão digital de pessoas com deficiências físicas motoras.

Existem várias aplicações que usam o reconhecimento de fala natural como interface de comunicação. Por exemplo, um editor de texto em que se dita o texto ao computador, que reconheceria as palavras e as colocaria na tela. Ou uma casa inteligente com a opção de ligar as luzes ou qualquer outro aparelho apenas com a voz.

Um discurso ou um comando por voz é um processo do mundo real caracterizado como contínuo. São sinais sonoros que podem possuir várias características, como por exemplo:

- ser puros (vindos apenas de uma fonte);
- ser corrompidos por outras fontes sonoras, fontes essas que caracterizam o ruído;
- ser distorcidos (pode ocorrer na transmissão do sinal).

O primeiro problema a ser enfrentado é a característica contínua do sinal sonoro. Por isso, faz-se necessário a sua digitalização. O próximo passo, então, é extrair características, ou atributos do sinal, para que este possa ser classificado. Assim, são usados modelos de sinais, que têm como principais finalidades:

- retirar ou diminuir ruído dos sinais;
- desfazer distorções geradas numa transmissão;
- caracterizar a fonte sonora por meio de parâmetros, pois quase sempre é muito custoso guardar todo o sinal sonoro.

Ou seja, os modelos de sinais são responsáveis por extrair do sinal sonoro características dele que, idealmente, sejam unívocas.

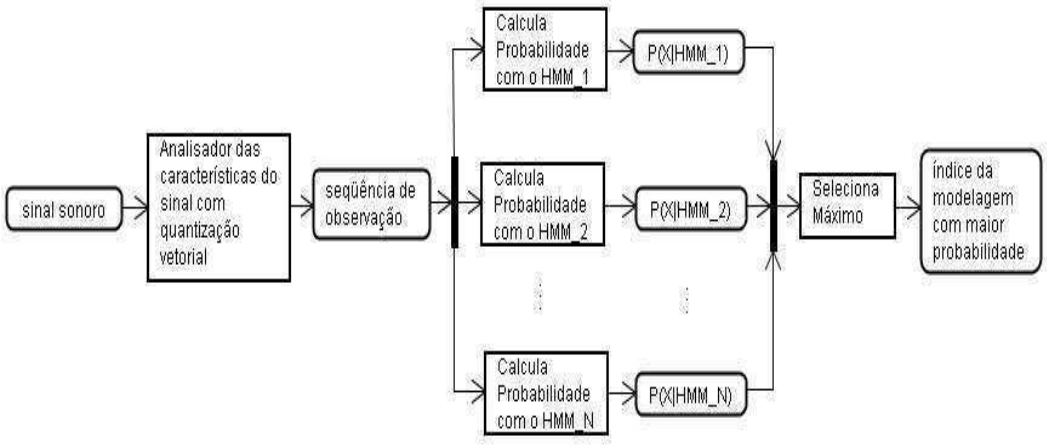

Figura 1.1: Diagrama de blocos para palavras isoladas usando coeficientes Mel-cepstrais, seus derivados e HMM

Um outro modelo entra na solução do seguinte problema: decidir se a seqüência de observação, caracterizada pelo modelo de sinal e representada no diagrama 1.1, é uma dentre a(s) esperada(s). Normalmente são utilizados modelos probabilísticos que precisam ser treinados previamente para decidir se o sinal recebido deve ser reconhecido ou rejeitado.

Hoje em dia essa é uma área de intensa pesquisa, pois o mercado mundial tem investido muito nessa tecnologia. Vários modelos se destacam para o reconhecimento de discursos pequenos (uma, duas, três ou quatro palavras). Um ótimo exemplo é um sistema baseado em coeficientes Mel-cepstrais e seus derivados, e no modelo escondido de Markov, como exemplificado no diagrama de blocos da figura 1.1 (Rabiner (1989)), onde os modelos de sinal e probabilístico estão bem esquematizados. A seguir, detalhes do processo:

- O modelo de sinal utilizado é composto por coeficientes Mel-cepstrais e seus derivados quantizados vetorialmente, representado pelo retângulo mais à esquerda da figura;

- O modelo probabilístico usado para reconhecimento do padrão do sinal sonoro recebido é o modelo escondido de Markov (HMM, do inglês *Hidden Markov Model*), que deve ser previamente treinado para cada palavra esperada. Ou seja, cada palavra possui um HMM próprio, representado pelos nomes HMM_1, HMM_2, ..., HMM_N. Esse modelo compara a seqüência de observações recebida em cada um dos HMM's, o que gera várias probabilidades (retângulos do centro da figura) do sinal sonoro ser pertencente àquele modelo;

- O retângulo mais a direita tem a função de comparar todas as probabilidades geradas no passo anterior e decidir qual é a maior probabilidade. Se um limite inferior existir e todos os modelos gerarem probabilidades baixas, a

resposta pode ser a rejeição do comando de voz recebido pelo modelo de sinal.

1.1 Objetivo

O objetivo deste trabalho é desenvolver um sistema de reconhecimento de palavras isoladas que utiliza o modelo escondido de Markov discreto. Neste caso pretende-se primar pela modularização do sistema, tendo em vista que as ferramentas desenvolvidas neste projeto servirão para o aprimoramento e no desenvolvimento de pesquisas futuras. Mais especificamente, pretende-se implementar algoritmos para:

- Processar (por exemplo, normalizar) sinais de áudio no formato WAVE;
- Extrair vetores de atributos de sinais de voz, utilizando coeificentes Mel-cepstrais e seus derivados;
- Quantizar vetorialmente os vetores de atributos;
- Construir modelos escondidos de Markov discretos;
- Reconhecer palavras isoladas utilizandos os coeficientes Mel-cepstrais quantizados como entrada dos modelos de Markov.

1.2 Organização do trabalho

O trabalho está disposto em 5 capítulos:

O capítulo 1 detalha a extração de características de um sinal de áudio. A ênfase está no cálculo e quantização vetorial dos coeficientes Mel-cepstrais e seus derivados, que são usados como entrada do modelo de classificação.

O capítulo 2 mostra o modelo escondido de Markov discreto. Sua parte principal a descrição dos algoritmos relacionados com o reconhecimento: *Forward*, *Forward-Backward* e *Viterbi*.

O capítulo 3 apresenta a forma de como foi desenvolvido o sistema de reconhecimento. A hierarquia e as descrições de cada módulo fornecem uma melhor compreensão do código produzido.

O capítulo 4 descreve o procedimento de teste usado para *software* especificado no capítulo 3. Os resultados obtidos são apresentados e analisados.

O capítulo 5 é o apêndice do trabalho e possui todo o código do sistema de reconhecimento, produzido em linguagem C.

Capítulo 2

Extração de atributos do sinal de voz

Para que sejam usados para reconhecimento da fala, os sinais de voz devem ser digitalizados e pré-processados. O objetivo do pré-processamento é obter uma representação paramétrica dos sinais, encontrando atributos que reduzam redundâncias e mantenham informações estatísticas suficientes para o reconhecimento. Um bom conjunto de atributos deve ser prático, robusto e seguro, o que significa que estas características devem ocorrer naturalmente e frequentemente no sinal de voz. Além disso, devem ser facilmente mensuráveis e não devem ser muito afetadas pela idade, saúde do locutor ou ruído de fundo.

Diversas representações paramétricas já foram experimentadas em sistemas de reconhecimento de voz, sendo que as que apresentam melhores resultados são os coeficientes Mel-cepstrais (do inglês, *Mel-Frequency Cepstral Coefficients*, MFCC) e seus derivados. Davis e Mermelstein (1980) apresentam uma revisão sobre essas representações e justificam a escolha dos MFCCs.

Neste capítulo será abordada a extração dos atributos de um sistema de reconhecimento de voz, que geralmente é realizada em três passos, a saber:

- Digitalização
- Extração de Coeficientes Mel-cepstrais e seus derivados
- Quantização Vetorial

2.1 Digitalização

Este passo inicial é responsável pela captura e armazenamento do sinal de voz em um formato digital. Um digitalizador pode ser representado pelo diagrama de blocos 2.1.

- *Microfone:* Recebe os sinais enviados pelo locutor através de ondas de pressão do ar $p(t)$ e os converte em sinais analógicos de tensão elétrica $x_c(t)$, onde t é a variável de tempo contínuo.
- *Pré-amplificador:* Filtro analógico de ganho positivo na entrada de $x_c(t)$.

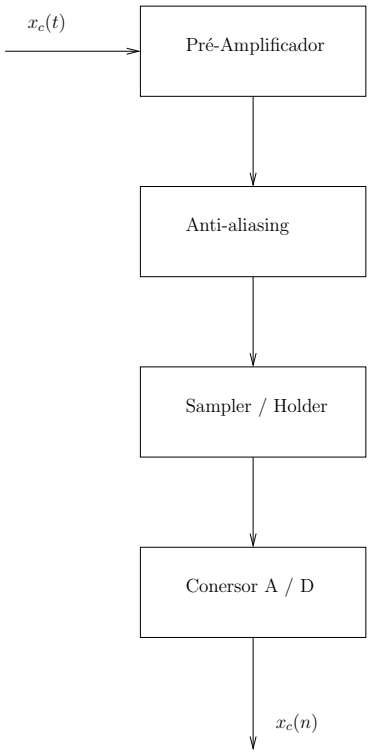

Figura 2.1: Exemplo de digitalização

- *Anti-aliasing:* Filtro analógico que corta freqüências altas (acima da largura de banda relevante).
- *Sampler and Holder:* Amostra (sampler) o sinal $x_c(t)$ em intervalos T_c, com freqüência de amostragem $f_c = 1/T_c$. O instante de amostragem t é discretizado, sendo dado por $t = nT_c$, onde n é a variável de tempo discreto da amostra. Dessa forma, o sinal de tempo contínuo $x_c(t)$ passa a ser representado por $x_c(n)$. Neste processo, o sinal $x_c(n)$ é mantido (holder) estável durante o intervalo de tempo necessário para a conversão A/D.
- *Conversor A/D:* Recebe os sinais amostrados $x_c(n)$ e os quantiza com uma determinada resolução em bits por amostra, gerando o sinal $x(n)$.
- *Banco de polifones:* Local de armazenamento persistente dos sinais digitalizados.

A figura 2.2 mostra um exemplo de sinal de voz digitalizado.

2.2 Extração de coeficientes Mel-Cepstrais e seus derivados

O cálculo de coeficientes Mel-cepstrais a partir do sinal de voz digitalizado pode ser resumido pelo diagrama de blocos da figura 2.3. A seguir são detalhados os passos para cálculo desses coeficientes.

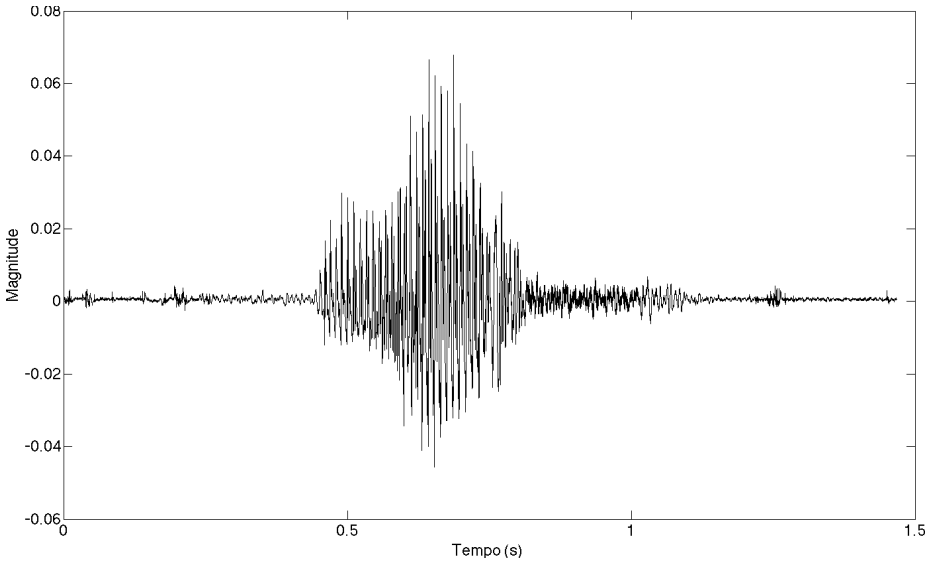

Figura 2.2: Exemplo de sinal de voz adquirido.

2.2.1 Pré-ênfase

Observa-se que, para sinais de voz, a energia carregada pelas altas freqüências é pequena quando comparada com as baixas freqüências. A pré-ênfase das freqüências altas é necessária para que se obtenha amplitudes mais homogêneas das freqüências formantes, pois informações importantes sobre a locução também estão presentes nas altas freqüências. Isto pode ser feito através de um filtro digital (Oppenheim (1999)), cuja função de transferência no domínio z é

$$H(z) = 1 - az^{-1}, 0 \leq a \leq 1 \tag{2.1}$$

onde a é o parâmetro responsável pela pré-ênfase, da ordem de 0.95.

No domínio do tempo, o filtro é implementado da seguinte forma

$$x'(n) = x(n) - ax(n-1) \tag{2.2}$$

O efeito da pré-ênfase no domínio do tempo pode ser observado comparando-se o sinal original (figura 2.2) e o sinal após a pré-ênfase (figura 2.4).

2.2.2 Janelamento

Aqui, as locuções são divididas em segmentos para viabilizar a análise de tempo curto do sinal. A segmentação é peça importante do processo de extração de características porque o espectro médio do sinal ao longo de toda a observação encobre variações espectrais que ocorrem durante intervalos curtos de tempo e que são de grande relevância para o reconhecimento. Desta forma, utiliza-se para análise uma janela de tempo suficientemente pequena (por exemplo, 20ms), onde as variações no espectro de um sinal de voz podem ser consideradas desprezíveis.

Seja $X = x(n), 0 \leq n \leq N-1$, onde N é o tamanho do *stream* de voz. No processo de janelamento, toma-se segmentos $Y_w = y_w(n)$ de tamanho $N_w < N$,

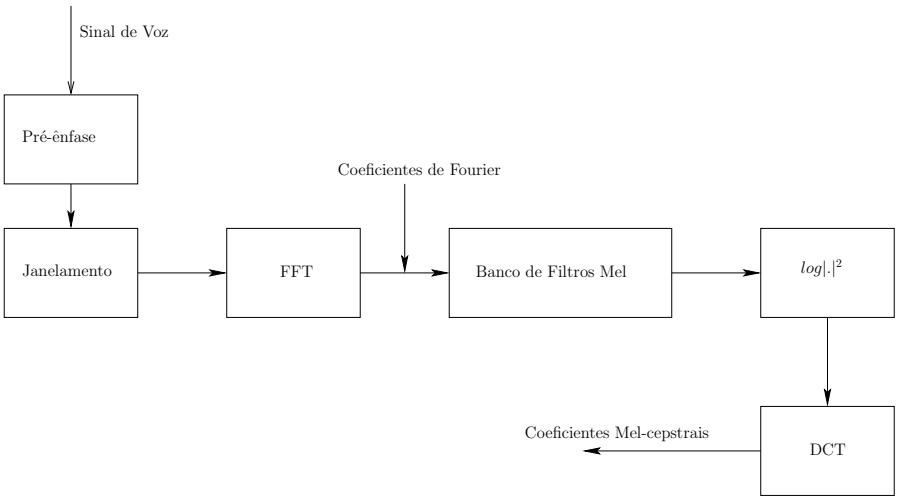

Figura 2.3: Diagrama de blocos da extração de coeficientes Mel-cepstrais.

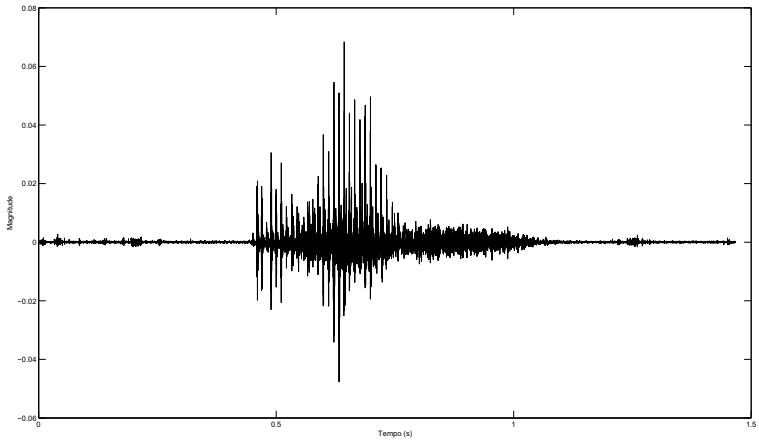

Figura 2.4: Sinal de voz após a pré-ênfase.

com N_w sendo uma potência de 2, sobrepostos segundo uma razão s

$$s = \frac{\text{número de amostras sobrepostas}}{N_w}, 0 \leq s \leq 1 \tag{2.3}$$

Os segmentos Y_w têm tamanho

$$W = \frac{N}{(1-s)N_w} \tag{2.4}$$

e

$$y_w(n) = e(n)x(n) \tag{2.5}$$

onde $e(n)$ é a função de janelamento.

Alguns exemplos de funções de janelamento são:

Janela de Hamming

$$e(n) = 0.54 - 0.46 \cos\left[\frac{2\pi}{N_w}\left(n - \frac{N_w}{2}\right)\right] \tag{2.6}$$

Janela de Barlet

$$e(n) = \begin{cases} \frac{2n}{N_w}, \text{ se } n < \frac{N_w}{2} \\ 2 - \frac{2n}{N_w}, \text{ se } n \geq \frac{N_w}{2} \end{cases} \tag{2.7}$$

Janela de Hanning

$$e(n) = 0.5 - 0.5 \cos\left[\frac{2\pi}{N_w}\left(n - \frac{N_w}{2}\right)\right] \tag{2.8}$$

Janela de Blackman

$$e(n) = 0.42 - 0.5 \cos\left[\frac{2\pi}{N_w}\left(n - \frac{N_w}{2}\right)\right] + 0.8 \cos\left[\frac{4\pi}{N_w}\left(n - \frac{N_w}{2}\right)\right] \tag{2.9}$$

A figura 2.5 apresenta uma janela do sinal de voz apresentado na figura 2.2, processada com a função de Hanning.

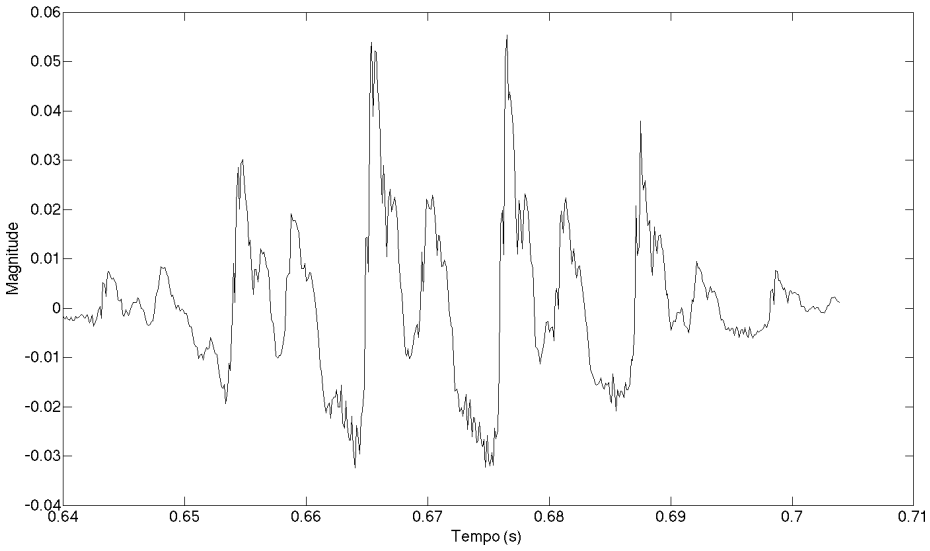

Figura 2.5: Décima janela do sinal de voz apresentado na figura 2.2, processada com a função de Hanning.

2.2.3 Análise de Fourier

Após o janelamento, cada segmento de voz é transformado para o domínio da freqüência utilizando a DFT (do inglês, *Discrete Fourier Transform*), com o algoritmo rápido da FFT (do inglês, *Fast Fourier Transform*) (Duhamel e Vetterli

, 1990; Malvar , 1992). Assim, são calculados os coeficientes $z_w(\omega)$, que estão representados na figura 2.6:

$$z_w(\omega) = \sum_{n=0}^{N_w-1} y_w(n) e^{-j\left(\frac{2\pi}{N_w}\right)\omega}, \tag{2.10}$$

A aplicação da DFT produz, para cada locução, um conjunto de janelas contendo $N_w/2$ componentes do espectro de potência do sinal. Somente a metade dos componentes é utilizada porque o sinal de entrada é composto por números reais e o espectro torna-se simétrico à freqüência de Nyquist, $f_c/2$.

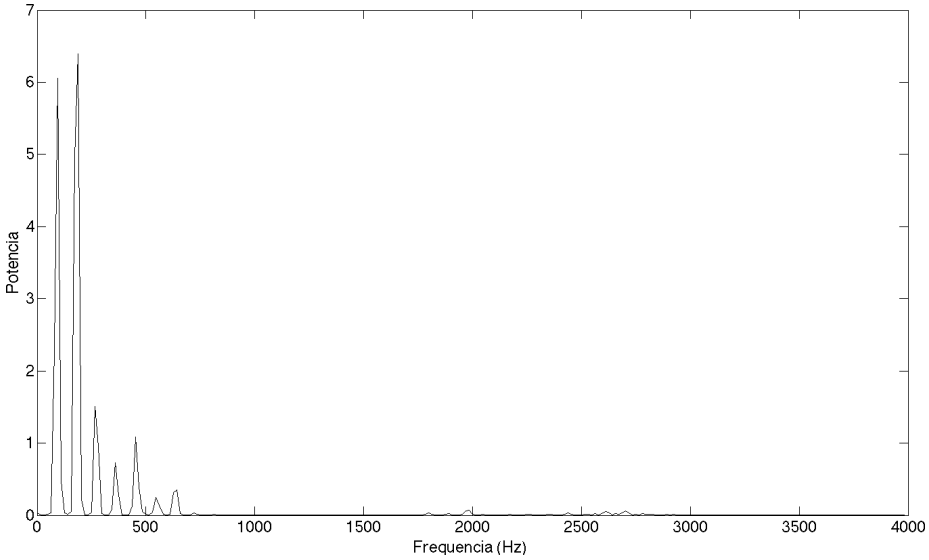

Figura 2.6: FFT do sinal de voz janelado apresentado na figura 2.5

2.2.4 Banco de Filtros Mel

No estudo da dinâmica do sistema auditivo humano, definiu-se uma escala psicoacústica de sensibilidade do ouvido para diversas freqüências do espectro audível, conhecida como escala Mel. A relação entre a freqüência recebida (f_{Hz}) e a freqüência percebida (f_{mel}) pelo ouvido humano é dada pela equação

$$f_{mel} = 1000 \frac{ln\left(1 + \frac{f_{Hz}}{700}\right)}{ln\left(1 + \frac{1000}{700}\right)} \tag{2.11}$$

Buscando um sistema de reconhecimento da fala que responde de forma semelhante ao sistema auditivo humano, aplica-se aos coeficientes $z_w(\omega)$ um banco de filtros triangulares (figura 2.7) para a conversão dos coeficientes no domínio da freqüência para a escala Mel.

Esses filtros fazem a soma dos coeficientes $z_w(\omega)$, ponderados por uma função triangular centrada na freqüência principal, dada pela equação 2.11, com valor 1 neste ponto. Ao resultado da soma, aplica-se o quadrado do módulo e o logaritmo na base 10.

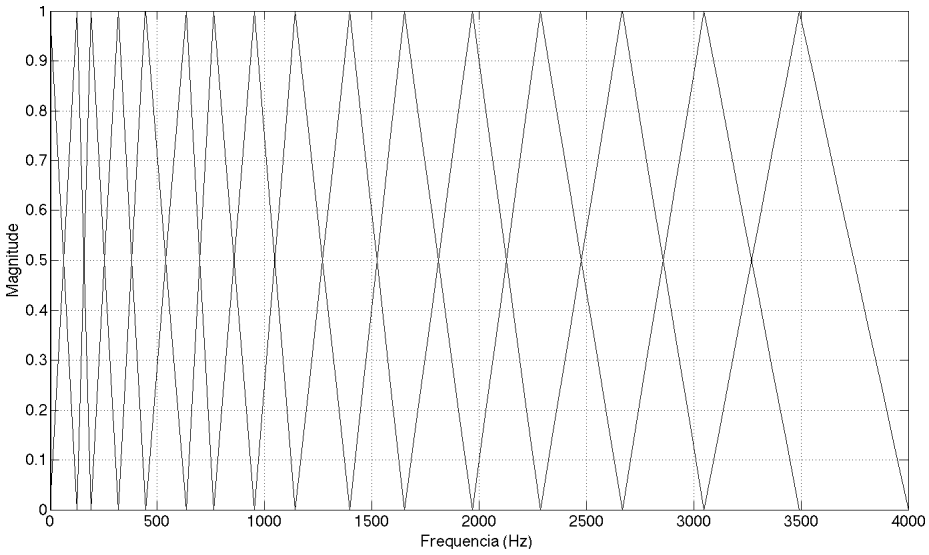

Figura 2.7: Banco de filtros triangulares para a conversão dos coeficientes no domínio da freqüência para a escala Mel.

Dessa forma, os N_w coeficientes de Fourier são transformado em M coeficientes na escala mel $z_w(\Omega)$. Sobre os logarítmos das energias dos filtros de bandas críticas $z_w(\Omega)$, é calculada a transformada de co-seno discreta Oppenheim (1999) gerando os coeficientes Mel-cepstrais (MFCCs) $c_w(m)$, $0 \leq m \leq M-1$.

A figura 2.8 mostra os coeficientes mel-cepstrais da décima janela do sinal de voz apresentado na figura 2.2.

2.2.5 Coeficientes Δ-Mel-cepstrais

Além dos coeficientes Mel-cepstrais em si, os sistemas atuais usam estimativas de suas derivadas temporais como componentes adicionais do vetor de atributos, colocando-as num patamar de igual relevância ao dos coeficientes em si (Juang e Furui , 2000; Furui , 1986). O objetivo desta inclusão é acrescentar alguma informação temporal sobre a variação dos MFCCs.

As componentes de primeira derivada dos coeficientes MFCC, conhecidas como Δ-Cepstrum ou Δ-MFCCs, representam a velocidade com que o espectro mel-cepstral varia e são facilmente computáveis através da diferença entre os coeficientes do segmento atual e um, ou dois, segmentos precedentes e consequentes. Assim, as componentes Δ-MFCC do coeficiente m no segmento w, são definidas por Furui (1986):

$$\Delta_w(m) = \frac{\sum_{\delta=1}^{2} \delta \left[c_{m+\delta}(m) - c_{m-\delta}(m)\right]}{2\sum_{\delta=1}^{2} \delta^2} \tag{2.12}$$

É de grande valia utilizar também as componentes de aceleração do espectro mel-cepstral, por isso aplica-se a equação acima novamente aos recém calculados coeficientes Δ-MFCCs, gerando coeficientes ΔΔ-MFCCs e adicionando-os ao vetor de atributos.

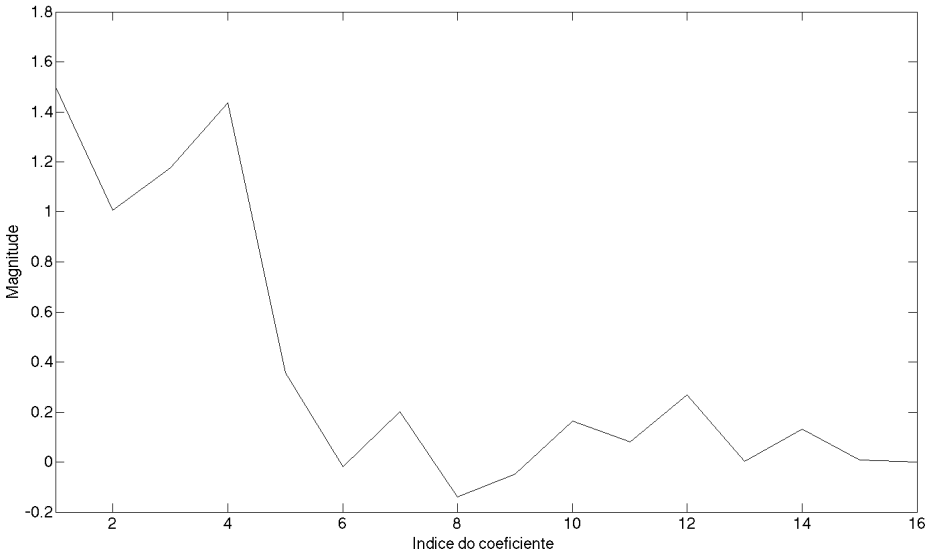

Figura 2.8: Coeficientes mel-cepstrais do sinal de voz janelado apresentado na figura 2.5.

2.3 Quantização Vetorial

Neste trabalho optou-se por utilizar um classificador discreto para o reconhecimento da fala, dessa forma o vetor de características utilizado como entrada do modelo deve ter suas amostras pertencentes a um alfabeto com quantidade de símbolos finita. Por isso, os coeficientes Mel-cepstrias calculados no item anterior não podem ser utilizados diretamente como vetores de características do sinal de voz. Assim, esses coeficientes devem ser quantizados vetorialmente.

A quantização vetorial, também chamada de quantização de blocos, atua em vetores de N dimensões que se encontram em um determinado espaço vetorial delimitado por células ou clusters. Essa limitação em células é feita a partir das magnitudes das distâncias euclidianas entre os vetores. Cada célula tem um vetor de código(*code-vector*) de M dimensões, também chamado de centróide, e cada centróide é representado por um código numérico de 1 a L, onde L é o número máximo de células que subdivide o espaço vetorial.

Um dos algoritmos mais utilizados para calcular os L centróides M-dimensionais, ou seja, para poder dividir o espaço vetorial em uma maneira ótima, é o algoritmo LBG (Linde et al. , 1980) .

2.3.1 Algoritmo LBG

O algoritmo LBG é uma extensão do algoritmo K-means (MacQueen , 1965). O codebook inicial é formado por apenas dois centróides correspondentes a duas células. Em seguida, o algoritmo executa uma divisão de cada um dos centróides, gerando um codebook de quatro centróides, que deverão ser atualizados, e assim por diante. O algoritmo implementado baseia-se em 3 passos principais:

1. *Incialização:* É escolhido um método para gerar os $B = 2$ centróides, podendo ser um método randômico.

2. *Recursão:*

   (a) Classificar cada um dos vetores de treinamento em alguma célula usando a regra de *Nearest Neighbor*, ou seja, o vetor que for mais próximo de um centróide, em termos de distância, pertencerá àquela célula correspondente.

   (b) Computar a distorção média global:

   $$D = \frac{1}{T} \sum_{n=1}^{T} d\left[x_n, y_n\right] \tag{2.13}$$

   onde y_n é o vetor código representante de cada vetor x_n, e T é o número de vetores de treinamento.

   (c) Atualizar os centróides como a média dos vetores de cada célula.

   (d) Determinar a diferença entre a distorção média global, obtida na iteração atual com a obtida na iteração anterior e compará-la a um limiar pré-estabelecido.

3. *Finalização:* Se B for menor que L, acrescentar novos B centróides ao *codebook*, duplicando o número de centróides e o número de células, y_n e retornar ao passo 2. Caso contrário, termine. A divisão do *codebook* se faz com um fator γ, chamado de fator de perturbação, que determina uma pequena alteração nos centróides de todos os agrupamentos a serem divididos, de acordo com:

   $$y_{n+B} = (1 - \gamma) y_n \tag{2.14}$$

   onde B é o número de centróides da iteração $0 \leq n \leq B$.

Com a quantização vetorial, cada vetor de atributos (contendo os coeficientes MFCC, Δ-MFCC e $\Delta\Delta$-MFCC) é representado por um número de 1 a L, dessa forma, cria-se um vetor de características com esses números, de W dimensões (quantidade de janelas). A figura 2.9 apresenta os vetores de códigos para as duas primeiras dimensões do vetor de características extraídos do sinal de voz apresentado na figura 2.2.

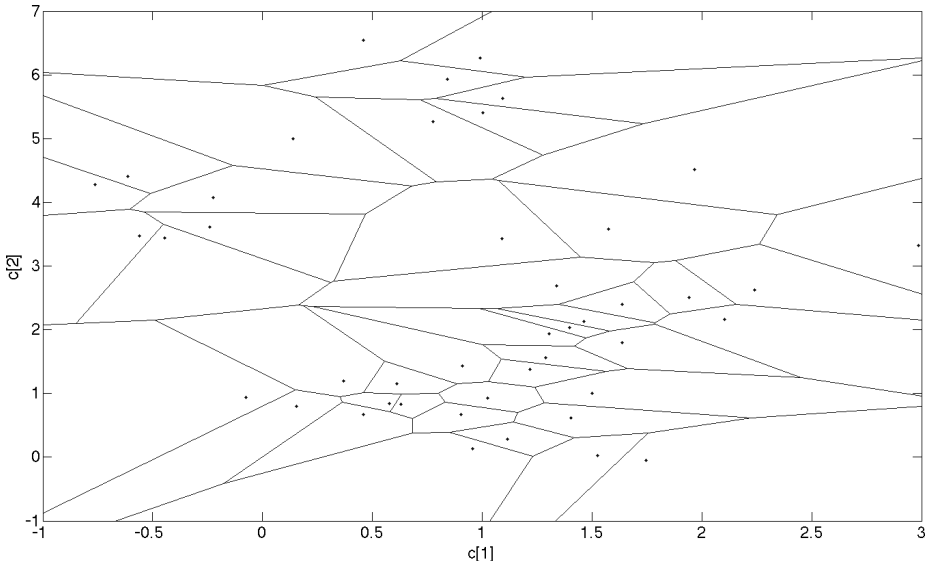

Figura 2.9: Vetores de códigos gerados para as duas primeiras dimensões dos coeificentes mel-cepstrais do sinal da figura 2.2

Capítulo 3

O modelo escondido de Markov - HMM

A questão de identificar um locutor ou reconhecer um comando possui soluções que utilizam diferentes modelagens. Uma abordagem possível está descrita no artigo de Kim e Yang (2006), que extrai características do sinal sonoro recebido com funções que geram os coeficientes Mel-Cepstrais e usam o modelo de Markov escondido (HMM, do inglês *Hidden Markov Model*) para identificar o locutor.

O HMM é uma modelagem matemática probabilística que caracteriza exemplos de dados em tempo discreto. Ele se tornou um dos métodos mais poderosos para reconhecimento e identificação de sinais sonoros. O princípio de funcionamento de um HMM já foi utilizado com sucesso em projetos relacionados a sinais sonoros, como reconhecimento de discurso automático, síntese de discurso, dentre outros.

Os conceitos que serão abordados nesse capítulo são os necessários para implementar a modelagem probabilística de reconhecimento, o modelo escondido de Markov.

Este capítulo possui duas principais referências: O artigo de Rabiner (1989) e o livro de Huang, Acero e Hon (2001).

3.1 Cadeia de Markov

A cadeia de Markov modela uma classe de processos aleatórios que não usam informação passada. Por exemplo, para um conjunto de eventos aleatórios, sabe-se que ele é uma cadeia de Markov de primeira ordem se for possível calcular a probabilidade de que ocorra um determinado evento considerando apenas o evento imediatamente anterior, mesmo que existam mais eventos que precedem esses dois em questão. Assim, temos:

$$P(X_i|X_{i-1}) \tag{3.1}$$

A equação 3.1 é o resultado da premissa de Markov: a probabilidade de ocorrer uma determinada saída dado que a saída anterior ocorreu é igual à probabilidade dessa mesma saída ocorrer considerando que todas as saídas anteriores ocorreram:

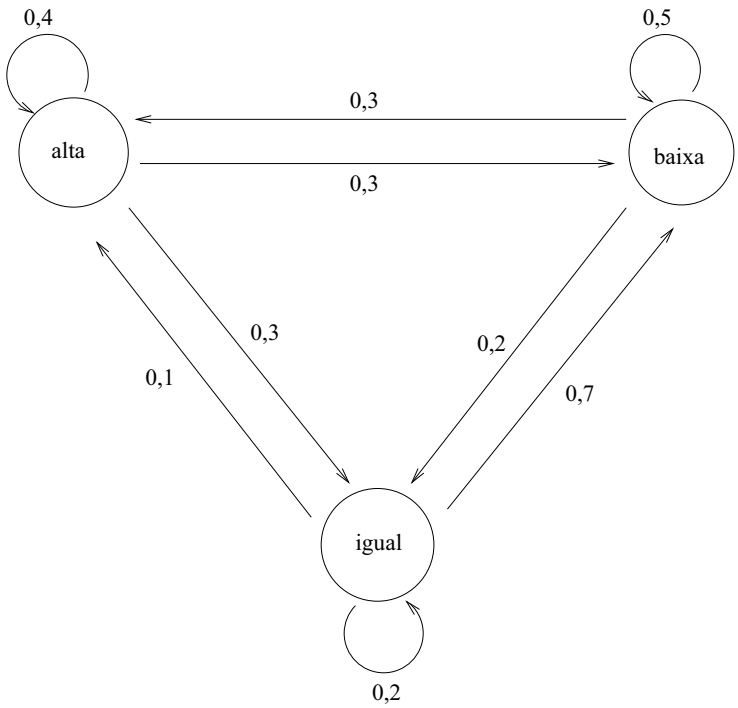

Figura 3.1: Diagrama de estados finitos modelando o comportamento de uma bolsa de valores

$P(X_i|X_{i-1}) = P(X_i|X_1^{i-1})$, onde $P(X_i|X_1^{i-1})$ é a probabilidade do evento X_i ocorrer dado que todos os eventos anteriores foram observados.

Um exemplo que se encaixa com a premissa de Markov é o relacionamento entre o modelo e um diagrama de estados finitos, figura 3.1. Ou seja, associa-se cada evento X_i a um estado do diagrama. E cada probabilidade $P(X_i|X_{i-1})$ a uma transição entre estados que dependerá apenas do estado atual do diagrama. A figura 3.1 possui um diagrama de estados finitos que modela de forma simplificada o comportamento de uma bolsa de valores. Os estados dão a informação de como está o mercado - em alta, em baixa ou estável - e estão mapeados da seguinte forma:

- Estado 1: alta;
- Estado 2: baixa;
- Estado 3: estável.

Com os estados identificados por números e sendo N o total, define-se as transições por:

$$\begin{aligned} a_{ij} = P(j|i) = P(s_t|s_{t-1}) \\ 1 \leq i \leq N \\ 1 \leq j \leq N \end{aligned} \tag{3.2}$$

E a matriz de probabilidades de transição como:

$$A = [a_{ij}] \tag{3.3}$$

Para o diagrama 3.1, os elementos da matriz A seriam:

- $a_{11} = 0,4$: probabilidade de estar no estado 1 e continuar nele;
- $a_{12} = 0,3$: probabilidade de sair do estado 1 para o estado 2;
- $a_{13} = 0,3$: probabilidade de sair do estado 1 para o estado 3;
- $a_{21} = 0,3$: probabilidade de sair do estado 2 para o estado 1;
- $a_{22} = 0,5$: probabilidade de estar no estado 2 e continuar nele;
- $a_{23} = 0,2$: probabilidade de sair do estado 2 para o estado 3;
- $a_{31} = 0,1$: probabilidade de sair do estado 3 para o estado 1;
- $a_{32} = 0,7$: probabilidade de sair do estado 3 para o estado 2;
- $a_{33} = 0,2$: probabilidade de estar no estado 3 e continuar nele;

que formam a seguinte matriz:

$$A = \begin{bmatrix} 0,4 & 0,3 & 0,3 \\ 0,3 & 0,5 & 0,2 \\ 0,1 & 0,7 & 0,2 \end{bmatrix}$$

Além disso, define-se a matriz de probabilidades iniciais, que diz qual a chance que cada estado tem de ser o primeiro da seqüência:

$$\begin{gathered} \pi = [\pi_i] \\ 1 \leq i \leq N \end{gathered} \tag{3.4}$$

Somente observando o diagrama de estados finitos não é possível extrair as informações sobre a matriz π, mas um exemplo seria:

$$\pi = \begin{bmatrix} 0,5 \\ 0,2 \\ 0,3 \end{bmatrix}$$

3.2 Definição do modelo escondido de Markov - HMM

Uma extensão da cadeia de Markov é o HMM, que introduz o não-determinismo ao modelo ao não deixar explícita a seqüência de estados quando determinadas saídas são observadas. Comparando com o exemplo dado na seção 3.1 (figura 3.1), em um HMM não sabe-se ao certo qual seria a seqüência de estados do mercado observando apenas um conjunto de saídas (alta, baixa ou estável). A probabilidade de uma saída ocorrer nesse modelo depende do estado atual em que ele se encontra. Porém, sem saber exatamente o estado é impossível determinar

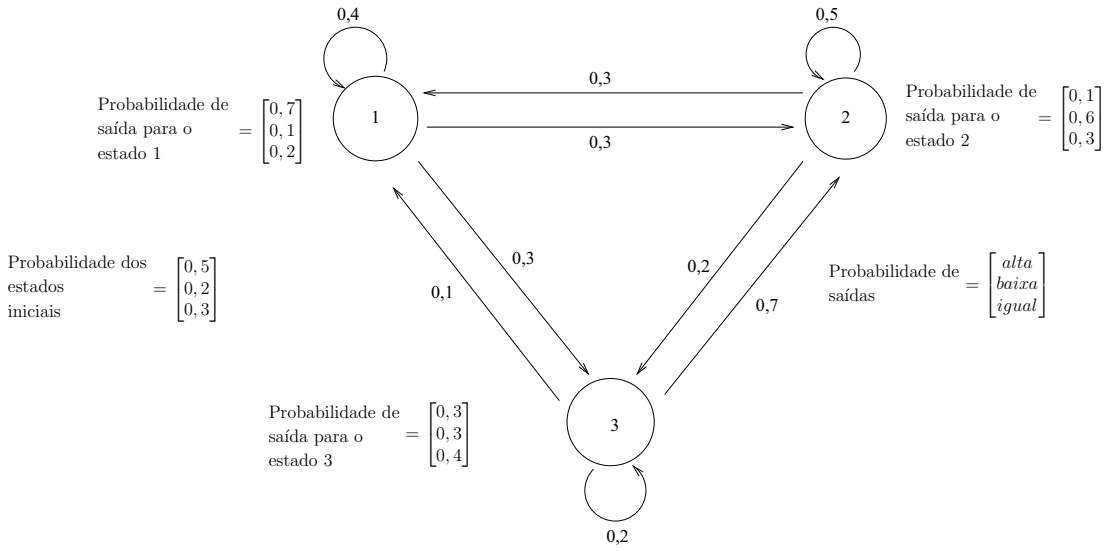

Figura 3.2: Diagrama de estados finitos modelando o comportamento de uma bolsa de valores por HMM

a probabilidade de um determinado evento acontecer, fato que explica o HMM ser um modelo não-observável.

A extensão do exemplo dessa cadeia de Markov para um HMM resulta em um diagrama similar ao diagrama 3.2. Nela, para cada estado temos uma probabilidade diferente de um determinado evento ocorrer. O estado 1 representa um mercado com tendência de alta. O estado 2, um mercado com tendência de queda. E o estado 3 representa um mercado conservador. Como os dados observáveis são os resultados diários, ou seja, se a bolsa teve queda, alta ou se manteve estável, não é possível determinar qual era a tendência do mercado no tempo em que a saída foi gerada. Essa tendência é determinada justamente pelos dados não observáveis, ou seja, a seqüência de estados.

Para uma definição formal completa de um HMM, define-se $b_j(o_k)$:

$$b_j(o_k), 1 \leq i \leq N, 1 \leq i \leq T, \tag{3.5}$$

que é a probabilidade da saída o_k ocorrer dado que estamos no estado j. A seguir, os parâmetros de um HMM:

- $O = \{o1, o2, \ldots, oM\}$, representando o alfabeto da modelagem. Para o exemplo da bolsa de valores seriam os resultados de alta, baixa, ou estável;
- $\Omega = \{1, 2, ..., N\}$, representando o espaço de estados. Para o exemplo da bolsa de valores seriam as tendências de alta, de baixa ou de estabilidade do mercado;
- $A = \{a_{ij}\}$, representando a matriz de probabilidades de transição de estados;
- $B = \{b_j(o_k)\}$, representando a matriz de probabilidades de uma saída o_k ser gerada dado um estado j;

- $\pi = \pi_i$, representando a matriz de probabilidades de cada estado i ser o inicial;
- N, representando o número total de estados do HMM;
- M, representando o número total de saídas possíveis do HMM (tamanho do alfabeto).

De forma resumida, o modelo é definido por $\Phi = (A, B, \pi)$.

3.3 Os três problemas do modelo

Para um HMM de primeira ordem temos as duas premissas:

1. A premissa de Markov:
$$P(s_t|s_1^{t-1}) = P(s_t|s_{t-1}) \tag{3.6}$$
2. A premissa da independência da saída:
$$P(s_t|s_1^{t-1}) = P(s_t|s_{t-1}) \tag{3.7}$$

Usar essa modelagem em aplicações do mundo real significa solucionar os três problemas a seguir, onde $X = \{x_1, x_2, \ldots, x_T\}$ é um conjunto de observações seqüencial, $\Phi = (A, B, \pi)$ é o modelo HMM e $S = \{s_0, s_1, \ldots, s_T\}$ é a seqüencia de estados mais provável:

1. **O problema da avaliação**: dado um modelo Φ e uma seqüência de observações X, qual é a probabilidade $P(X|\Phi)$, ou seja, qual a probabilidade do modelo gerar o conjunto X observado. Com esse problema resolvido, é possível avaliar o quão perto um modelo está da seqüência de observações observada (problema de *match*);
2. **O problema da decodificação**: dado um modelo Φ e um conjunto de observações X, qual é a seqüência de estados S que maximiza o resultado de $P(X|\Phi)$ - onde essa seqüência não é observável. A solução desse problema afeta diretamente a solução do problema de avaliação;
3. **O problema do aprendizado**: dado um modelo Φ e um conjunto de observações X, qual é a maneira de ajustar os três parâmetros A, B e π do HMM com o objetivo de maximizar a probabilidade comum $P(X|\Phi)$. A solução desse problema fornece meios para estimar automaticamente os parâmetros do HMM a partir de uma ou mais seqüências de observações.

Cada problema descrito acima possui um algoritmo clássico como solução:

1. Algoritmo Forward, que está relacionado com o problema da avaliação;
2. Algoritmo de Viterbi, que está relacionado com o problema da decodificação;
3. Algoritmo de Baum-Welch ou algoritmo Forward-Backward, que está relacionado com o problema do aprendizado;

Os quatro algoritmos estão descritos nas seção seguinte.

3.4 Os algoritmos Forward, Viterbi, Baum-Welch e Forward-Backward

Essa seção usará o exemplo da figura 3.2, ou seja, o HMM possuirá os seguintes parâmetros:

- 3 estados = {tendência de alta(1), tendência de baixa(2), tendência de estabilidade(3)}
- 3 saídas possíveis (alfabeto) = {alta(1), baixa(2), estável(3)}
- $\pi = \begin{bmatrix} 0,5 \\ 0,2 \\ 0,3 \end{bmatrix}$
- $A = \begin{bmatrix} 0,4 & 0,3 & 0,3 \\ 0,3 & 0,5 & 0,2 \\ 0,1 & 0,7 & 0,2 \end{bmatrix}$
- $B = \begin{bmatrix} 0,7 & 0,1 & 0,2 \\ 0,1 & 0,6 & 0,3 \\ 0,3 & 0,3 & 0,4 \end{bmatrix}$
- 4 saídas observadas = {alta(X_1), baixa(X_2), estável(X_3), baixa(X_4)}

3.4.1 Algoritmo Forward

O objetivo do algoritmo é calcular $P(X|\Phi)$ a partir de uma entrada de dados X. Uma maneira intuitiva de calcular essa probabilidade seria:

$$P(X|\Phi) = \sum_S [P(S|\Phi)P(X|S,\Phi)] = \sum_S a_{s_0 s_1} b_{s_1}(X_1) a_{s_1 s_2} b_{s_2}(X_2) \dots a_{s_{T-1} s_T} b_{s_T}(X_T), \tag{3.8}$$

onde $a_{s_0 s_1}$ representa a probabilidade do estado s_1 ser o estado inicial (igual a π_1), $as_1 s_2$ representa a probabilidade de transição do estado s_1 para o estado s_2, e $b_{s_1}(X_1)$ representa a probabilidade da saída X_1 ocorrer dado que estamos no estado s_1.

Calcular diretamente essa equação demandaria um esforço computacional bem grande, da ordem de N^T (Oliveira e Morita (2002)). Porém, é possível armazenar os cálculos intermediários evitando executar o mesmo cálculo várias vezes. Por exemplo, os dois primeiros termos da segunda equação de 3.8, $a_{s_0 s_1} b_{s_1}(X_1)$ e $a_{s_1 s_2} b_{s_2}(X_2)$. Eles seriam calculados para todos os termos em que T fosse maior ou igual a dois. Se esse valor é armazenado, o esforço computacional é reduzido.

Usando as premissas da modelagem - equações 3.6 e 3.7 - verifica-se que o cálculo de $P(s_t|s_{t-1},\Phi) * P(X_t|s_t,\Phi)$ envolve apenas uma recursão em t, que pode ser representada por uma indução. Para apresentar um pseudo-código do algoritmo, define-se a probabilidade do modelo estar no estado i em um tempo t

tendo gerado uma observação parcial $X_1^t = \{X_1, X_2, \ldots, X_t\}$, nomeada de probabilidade forward, por:

$$\alpha_t(i) = P(X_1^t, s_t = i|\Phi) \tag{3.9}$$

Algoritmo Forward é (Huang, Acero e Hon (2001)):

1. Inicialização:
$$\alpha_1(i) = \pi_i b_i(X_1), 1 \leq i \leq N$$

2. Indução:
$$\alpha_t(j) = \sum_{i=1}^{N} \left[\alpha_{t-1}(i)a_{ij}\right] b_j(X_t), 2 \leq t \leq T, 1 \leq i \leq N$$

3. Finalização:
$$P(X|\Phi) = \sum_{i=1}^{N} \alpha_T(i)$$

Usando o HMM definido anteriormente e com as saídas observadas como exemplo para esse algoritmo, obtem-se:

1. Inicialização:
$\alpha_1(1) = \pi_1 b_1(X_1) = 0,5 \cdot 0,7 = 0,35$
$\alpha_1(2) = \pi_2 b_2(X_1) = 0,2 \cdot 0,1 = 0,02$
$\alpha_1(3) = \pi_2 b_3(X_1) = 0,3 \cdot 0,3 = 0,09$

2. Indução:
$$\begin{aligned}\alpha_2(1) &= (\alpha_1(1)a_{11} + \alpha_1(2)a_{21} + \alpha_1(3)a_{31})b_1(X_2)\\ &= (0,35 \cdot 0,4 + 0,02 \cdot 0,3 + 0,09 \cdot 0,1) \cdot 0,1\\ &= 0,0155\end{aligned}$$
$$\begin{aligned}\alpha_2(2) &= (\alpha_1(1)a_{12} + \alpha_1(2)a_{22} + \alpha_1(3)a_{32})b_2(X_2)\\ &= (0,35 \cdot 0,3 + 0,02 \cdot 0,5 + 0,09 \cdot 0,7) \cdot 0,6\\ &= 0,1068\end{aligned}$$
$$\begin{aligned}\alpha_2(3) &= (\alpha_1(1)a_{13} + \alpha_1(2)a_{23} + \alpha_1(3)a_{33})b_3(X_2)\\ &= (0,35 \cdot 0,3 + 0,02 \cdot 0,2 + 0,09 \cdot 0,2) \cdot 0,3\\ &= 0,0381\end{aligned}$$
$$\begin{aligned}\alpha_3(1) &= (\alpha_2(1)a_{11} + \alpha_2(2)a_{21} + \alpha_2(3)a_{31})b_1(X_3)\\ &= (0,0155 \cdot 0,4 + 0,1068 \cdot 0,3 + 0,0381 \cdot 0,1) \cdot 0,2\\ &= 0,00841\end{aligned}$$
$$\begin{aligned}\alpha_3(2) &= (\alpha_2(1)a_{12} + \alpha_2(2)a_{22} + \alpha_2(3)a_{32})b_2(X_3)\\ &= (0,0155 \cdot 0,3 + 0,1068 \cdot 0,5 + 0,0381 \cdot 0,7) \cdot 0,3\\ &= 0,025416\end{aligned}$$

$$\begin{aligned}\alpha_3(3) &= (\alpha_2(1)a_{13} + \alpha_2(2)a_{23} + \alpha_2(3)a_{33})b_3(X_3) \\ &= (0,0155 \cdot 0,3 + 0,1068 \cdot 0,2 + 0,0381 \cdot 0,2) \cdot 0,4 \\ &= 0,013452\end{aligned}$$

$$\begin{aligned}\alpha_4(1) &= (\alpha_3(1)a_{11} + \alpha_3(2)a_{21} + \alpha_3(3)a_{31})b_1(X_4) \\ &= (0,00841 \cdot 0,4 + 0,025416 \cdot 0,3 + 0,013452 \cdot 0,1) \cdot 0,1 \\ &= 0,0012334\end{aligned}$$

$$\begin{aligned}\alpha_4(2) &= (\alpha_3(1)a_{12} + \alpha_3(2)a_{22} + \alpha_3(3)a_{32})b_2(X_4) \\ &= (0,00841 \cdot 0,3 + 0,025416 \cdot 0,5 + 0,013452 \cdot 0,7) \cdot 0,6 \\ &= 0,01478844\end{aligned}$$

$$\begin{aligned}\alpha_4(3) &= (\alpha_3(1)a_{13} + \alpha_3(2)a_{23} + \alpha_3(3)a_{33})b_3(X_4) \\ &= (0,00841 \cdot 0,3 + 0,025416 \cdot 0,2 + 0,013452 \cdot 0,2) \cdot 0,3 \\ &= 0,00308898\end{aligned}$$

3. Finalização:

$$\begin{aligned}P(X|\Phi) &= \alpha_4(1) + \alpha_4(2) + \alpha_4(3) \\ &= 0,0012334 + 0,01478844 + 0,00308898 \\ &= 0,01911082\end{aligned}$$

Com esse exemplo prático, obtem-se que um modelo escondido de Markov definido com esses parâmetros tem uma probabilidade de aproximadamente 1,9% de gerar o conjunto de saídas X = {alta, baixa, estável, baixa}.

3.4.2 Algoritmo de Viterbi

O objetivo é conseguir uma seqüência de estados S de um determinado HMM que maximiza a probabilidade $P(X|S,\Phi)$ (que é parte do cálculo de $P(X|\Phi)$), e ela será nomeada de melhor seqüência. Como visto anteriormente, essa informação não é observável. Portanto, a saída do algoritmo não é a seqüência de estados que ocorreu de fato, mas uma que melhor se encaixa com uma entrada X específica. Técnicas de programação dinâmica são usadas e define-se a probabilidade da melhor seqüência de estados no tempo t gerar o conjunto de observações parcial X_i^t e terminar no estado i como:

$$V_t(i) = P(X_1^t, S_1^{t-1}, s_t = i|\Phi) \tag{3.10}$$

Novamente, existe uma recursão em t, de forma similar à vista no algoritmo Forward (3.4.1). O Viterbi obtém o melhor caminho parcial até o tempo t e lembra dele ao calcular o melhor caminho subseqüente. Para escrever o pseudocódigo a seguir, será usado um atributo chamado $B_t(j)$, que guardará o estado final correspondente a $V_t(j)$ e $S = (s_1, s_2, \ldots, s_T)$ será a melhor seqüência.

O algoritmo Viterbi é (Huang, Acero e Hon (2001)):

1. Inicialização:

$$V_1(i) = \pi_i \cdot b_i(X_1), \quad 1 \leq i \leq N$$

$$B_t(i) = 0, \quad 1 \leq i \leq N$$

2. Indução:

   $V_t(i) = \max_{1 \leq i \leq N} [V_{t-1}(i) \cdot a_{ij}] \cdot b_j(X_t), \quad 2 \leq t \leq T, \quad 1 \leq i \leq N$

   $B_t(i) = \arg\max_{1 \leq i \leq N} [V_{t-1}(i) \cdot a_{ij}], \quad 2 \leq t \leq T, \quad 1 \leq i \leq N$

3. Finalização:

   $s_T = \arg\max_{1 \leq i \leq N} [B_T(i)]$

   $s_t = B_{t+1}(s_{t+1}), \quad t = T-1, T-2, \ldots, 2, 1$

   $S = (s_1, s_2, \ldots, s_T)$

Usando o HMM definido anteriormente e com as saídas observadas usadas como exemplo para esse algoritmo, tem-se:

1. Inicialização:

   $V_1(1) = \pi_1 \cdot b_1(X_1) = 0,5 \cdot 0,7 = 0,35$
   $V_1(2) = \pi_2 \cdot b_2(X_1) = 0,2 \cdot 0,1 = 0,02$
   $V_1(3) = \pi_2 \cdot b_3(X_1) = 0,3 \cdot 0,3 = 0,09$

   $B_1(1) = 0$
   $B_1(2) = 0$
   $B_1(3) = 0$

2. Indução, onde $\hookrightarrow$ aponta o maior valor encontrado:

   $\hookrightarrow V_1(1) \cdot a_{11} = 0,35 \cdot 0,4 = 0,140$
   $V_1(2) \cdot a_{21} = 0,02 \cdot 0,3 = 0,006$
   $V_1(3) \cdot a_{31} = 0,09 \cdot 0,1 = 0,009$

   $\hookrightarrow V_1(1) \cdot a_{12} = 0,35 \cdot 0,3 = 0,105$
   $V_1(2) \cdot a_{22} = 0,02 \cdot 0,5 = 0,010$
   $V_1(3) \cdot a_{32} = 0,09 \cdot 0,7 = 0,063$

   $\hookrightarrow V_1(1) \cdot a_{13} = 0,35 \cdot 0,3 = 0,105$
   $V_1(2) \cdot a_{23} = 0,02 \cdot 0,2 = 0,004$
   $V_1(3) \cdot a_{33} = 0,09 \cdot 0,2 = 0,018$

   $V_2(1) = 0,140 \cdot 0,1 = 0,0140$
   $V_2(2) = 0,105 \cdot 0,6 = 0,0630$
   $V_2(3) = 0,105 \cdot 0,3 = 0,0315$

   $B_2(1) = 1$
   $B_2(2) = 1$
   $B_2(3) = 1$

   $V_2(1) \cdot a_{11} = 0,0140 \cdot 0,4 = 0,00560$
   $\hookrightarrow V_2(2) \cdot a_{21} = 0,0630 \cdot 0,3 = 0,01890$
   $V_2(3) \cdot a_{31} = 0,0315 \cdot 0,1 = 0,00315$

   $V_2(1) \cdot a_{12} = 0,0140 \cdot 0,3 = 0,00420$
   $\hookrightarrow V_2(2) \cdot a_{22} = 0,0630 \cdot 0,5 = 0,03150$
   $V_2(3) \cdot a_{32} = 0,0315 \cdot 0,7 = 0,02205$

$V_2(1) \cdot a_{13} = 0,0140 \cdot 0,3 = 0,0042$
$\hookrightarrow V_2(2) \cdot a_{23} = 0,0630 \cdot 0,2 = 0,0126$
$V_2(3) \cdot a_{33} = 0,0315 \cdot 0,2 = 0,0063$

$V_3(1) = 0,01890 \cdot 0,2 = 0,00378$
$V_3(2) = 0,03150 \cdot 0,3 = 0,00945$
$V_3(3) = 0,01260 \cdot 0,4 = 0,00504$

$B_3(1) = 2$
$B_3(2) = 2$
$B_3(3) = 2$

$V_3(1) \cdot a_{11} = 0,00378 \cdot 0,4 = 0,001512$
$\hookrightarrow V_3(2) \cdot a_{21} = 0,00945 \cdot 0,3 = 0,002835$
$V_3(3) \cdot a_{31} = 0,00504 \cdot 0,1 = 0,000504$

$V_3(1) \cdot a_{12} = 0,00378 \cdot 0,3 = 0,001134$
$\hookrightarrow V_3(2) \cdot a_{22} = 0,00945 \cdot 0,5 = 0,004725$
$V_3(3) \cdot a_{32} = 0,00504 \cdot 0,7 = 0,003528$

$V_3(1) \cdot a_{13} = 0,00378 \cdot 0,3 = 0,001134$
$\hookrightarrow V_3(2) \cdot a_{23} = 0,00945 \cdot 0,2 = 0,001890$
$V_3(3) \cdot a_{33} = 0,00504 \cdot 0,2 = 0,001008$

$V_4(1) = 0,002835 \cdot 0,1 = 0,0002835$
$V_4(2) = 0,004725 \cdot 0,6 = 0,0028350$
$V_4(3) = 0,001890 \cdot 0,3 = 0,0005670$

$B_4(1) = 2$
$B_4(2) = 2$
$B_4(3) = 2$

3. Finalização:

$s_4 = 2 \quad s_3 = B_4(2) = 2 \quad s_2 = B_3(2) = 2 \quad s_1 = B_2(2) = 1$

$S = (s_1, s_2, s_3, s_4) = (1, 2, 2, 2)$

Com esse exemplo, verifica-se que se forem observadas as probabilidades de alta, baixa e estabilidade em cada tendência de mercado (estados) e as respectivas probabilidades de transição entre tendências de mercado (matriz A), a melhor seqüência de estados para gerar um conjunto de saídas X =\{alta, baixa, estável, baixa\} é $S = (1, 2, 2, 2)$.

3.4.3 Algoritmo de Baum-Welch e o algoritmo *Forward-Backward*

O objetivo dos algoritmos é conseguir estimar os parâmetros do modelo $\Phi = (A, B, \pi)$ que maximiza a probabilidade comum $P(X|\Phi)$ para uma ou várias seqüências de saídas observadas. Dos três problemas enumerados na seção 3.2, esse é o mais complexo, pois não há método analítico conhecido que execute essa maximização (Huang, Acero e Hon (2001) e Rabiner (1989)). Então, esse

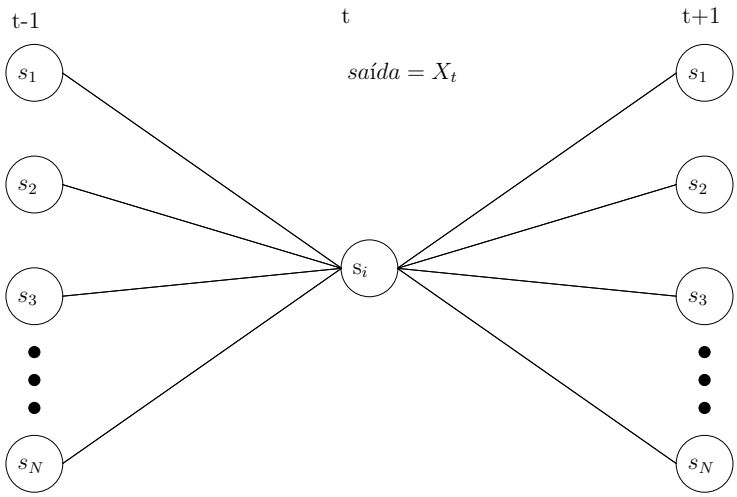

Figura 3.3: Relação entre α e β

algoritmo usa um método iterativo que pára quando alcança uma determinada convergência.

Assim como o algoritmo *Forward* , onde se define $\alpha_t(i)$, são definidos aqui as probabilidades $\beta_t(i)$ e $\gamma_t(i,j)$. $\beta_t(i)$, também chamada de probabilidade *Backward*, é a probabilidade do modelo gerar um conjuntos de saídas parcial $X_{t+1}^T = (X_{t+1}, X_{t+2}, \ldots, X_T)$, dado que o estado no tempo t é o estado i:

$$\beta_t(i) = P\left(X_{t+1}^T | s_t = i, \Phi\right) \tag{3.11}$$

A probabilidade *Backward* pode ser calculada por indução da seguinte maneira (Rabiner (1989)):

1. Inicialização:
$$\beta_T(i) = 1 \quad 1 \leq i \leq N$$

2. Indução:
$$\begin{array}{l} \beta_t(i) = \sum_{j=1}^{N} \left[a_{ij} \cdot b_{t+1}(X_{t+1}) \cdot \beta_{t+1}(j)\right] \\ t = T-1, T-2, \ldots, 1; \\ 1 \leq i \leq N \end{array}$$

A relação entre α e β é bem expressa pela figura 3.3

$\gamma_t(i,j)$ é a probabilidade da transição do estado i para o estado j ocorrer no tempo t, dado o modelo Φ e a seqüência de observações. A definição e a maneira de calcular usando α e β (Huang, Acero e Hon (2001)):

$$\begin{aligned} \gamma_t(i,j) &= P\left(s_{t-1} = i, s_t = j | X_1^T, \Phi\right) \\ &= \frac{P\left(s_{t-1}=i, s_t=j, X_1^T | \Phi\right)}{P(X_1^T | \Phi)} \\ &= \frac{\alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(X_t) \cdot \beta_t(j))}{\sum_{k=1}^{N} \alpha_T(k)} \end{aligned} \tag{3.12}$$

Usando as informações definidas, é possível reestimar os parâmetros da modelagem com as seguintes equações - onde $\overline{a}$ e $\overline{b}$ são os a_{ij}'s e $b_j(X_t)$'s reestimados - (Huang, Acero e Hon (2001) Rabiner (1989)):

$$\begin{aligned} \overline{a}_{ij} &= \frac{\frac{1}{P(X|\Phi)} \cdot \sum_{t=1}^{T} P(X, s_{t-1}=i, s_t=j|\Phi)}{\frac{1}{P(X|\Phi)} \cdot \sum_{t=1}^{T} P(X, s_{t-1}=i|\Phi)} \\ &= \frac{\sum_{t=1}^{T} \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t(i,k)} \end{aligned} \tag{3.13}$$

$$\begin{aligned} \overline{b}_j(k) &= \frac{\frac{1}{P(X|\Phi)} \cdot \sum_{t=1}^{T} P(X, s_t=j, |\Phi) \cdot \delta(X_t, o_k)}{\frac{1}{P(X|\Phi)} \cdot \sum_{t=1}^{T} P(X, s_t=j, |\Phi)} \\ &= \frac{\sum_{t \in X_t=o_k} \sum_{i=1}^{N} \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_t(i,j)} \end{aligned} \tag{3.14}$$

A reestimação desses parâmetros é feita até que $\sum_{i=1}^{N} \alpha_T(i)$ atinja uma convergência esperada. Supondo que o modelo será treinado com W seqüências de saídas, esse processo deve ser feito com cada uma, ou seja, W HMM's serão gerados. A partir das modelagens criadas, um único HMM deve ser feito, o que define uma iteração do algoritmo Baum-Welch. Isso deve ocorrer até que

$$\sum_{w=1}^{W} \sum_{i=1}^{N} \alpha_T^w(i),$$

onde $\alpha_T^w(i)$ corresponde ao $\alpha_T(i)$ do w-ésimo HMM gerado, atinja uma convergência esperada.

As equações de criação de um modelo Escondido de Markov a partir de vários HMM's são como a seguir:

$$\overline{a}_{ij} = \frac{\sum_{w=1}^{W} \sum_{t=1}^{T} \gamma_t^w(i,j)}{\sum_{w=1}^{W} \sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t^w(i,k)} \tag{3.15}$$

$$\overline{b}_j(k) = \frac{\sum_{w=1}^{W} \sum_{t \in X_t=o_k} \sum_{i=1}^{N} \gamma_t^w(i,j)}{\sum_{w=1}^{W} \sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_t^w(i,j)} \tag{3.16}$$

3.5 Fatores de escala

Quando se calcula a probabilidade *Forward* e *Backward*, especificadas na seção 3.4, elas se aproximam exponencialmente de zero, já que ambas são calculadas recursivamente e seus valores são sempre menores que 1. Para um T suficientemente grande, a grandeza de α e β ultrapassarão a precisão de qualquer máquina, mesmo usando precisão dupla de ponto flutuante. Assim, na prática, usando os algoritmos na forma em que foram descritos, ocorrerá *underflow*. Esse problema é solucionado usando fatores de escala para essas probabilidades. Por exemplo, $\alpha_t(i)$ e $\beta_t(i)$ podem ser multiplicados por (Huang, Acero e Hon (2001)): Rabiner (1989):

$$S_t = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)}, \tag{3.17}$$

$$S_t = \frac{1}{\sum_{i=1}^{N} \beta_t(i)}, \tag{3.18}$$

Onde S_t é o fator que evita *underflow* para o tempo t. Então, como as probabilidades em questão são calculadas de forma recursiva, para um suposto $\alpha_3(2)$, os fatores S_1, S_2 e S_3 estão aplicados. Logo, os fatores de escala para as probabilidades *Forward* e *Backward* em um tempo t são:

$$fator_{\alpha}(t) = \prod_{k=1}^{t} S_k \tag{3.19}$$

$$fator_{\beta}(t) = \prod_{k=t}^{T} S_k \tag{3.20}$$

Seja $\alpha_t^{'}(i)$, $\beta_t^{'}(i)$ e $\gamma_t^{'}(i,j)$ as probabilidades com seus respectivos fatores de escala. Dessa forma, tem-se (Huang, Acero e Hon (2001)):

$$\sum_{i=1}^{N} \alpha_T^{'}(i) = fator_{\alpha}(t) \cdot \sum_{i=1}^{N} \alpha_T(i) = fator_{\alpha}(t) \cdot P(X|\Phi) \tag{3.21}$$

$$\gamma_t^{'}(i,j) = \frac{fator_{\alpha}(t-1)\alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(X_t) \cdot \beta_t(j) \cdot fator_{\beta}(t)}{fator_{\alpha}(T) \sum_{i=1}^{N} \alpha_T(i)} = \gamma_t(i,j) \tag{3.22}$$

De acordo com a equação 3.22 conclui-se que nada precisa ser alterado para o cálculo do γ, pois os fatores de escala se cancelam. E, ainda, calcula-se $P(X|\Phi)$ com o algoritmo *Forward*, com base na equação 3.21, devemos fazer:

$$P(X|\Phi) = \frac{\sum_{i=1} N\alpha_T^{'}(i)}{fator_{\alpha}(T)}$$

Capítulo 4

Implementação do Sistema de Reconhecimento da Fala

Este capítulo detalha o software produzido neste trabalho, que é capaz de produzir modelos escondidos de Markov, representados por estrutura de dados pré-definidas, e com eles, reconhecer sinais sonoros em arquivo no formato WAVE. O programa foi implementado com os algoritmos e a teoria dos capítulos 2 e 3.

O diagrama 2.3 representa exatamente o modelo de sinal produzido no *software*.

Já o modelo probabilístico usado possui as seguintes caracterísiticas que podem ser alteradas no arquivo *defines.h*:

- 5 estados, onde cada estado representa um fonema da palavra que serviu como entrada do sistema;
- alfabeto de tamanho 64, 128 e 512;
- 37 coeficientes em cada bloco:
  - 12 coeficientes Mel-cepstrais;
  - 12 coeficientes Δ-Mel-cepstrais;
  - 12 coeficientes $\Delta\Delta$-Mel-cepstrais;
  - 1 coeficiente de energia total do bloco.

A hierarquia do sistema está representada na figura 4.1. Ela está dividida em dois grandes grupos: quatro módulos principais, responsáveis por ordenar a chamada de funções, e sete módulos secundários, que têm a tarefa de executar algoritmos e funções matemáticas específicas. A seguir, uma descrição de cada módulo.

Módulos principais:

- *telas.c*: o objetivo desse módulo é centralizar toda a escrita na tela. Tanto de informações de parâmetros usados e de estado de execução do programa quanto de resultados obtidos.

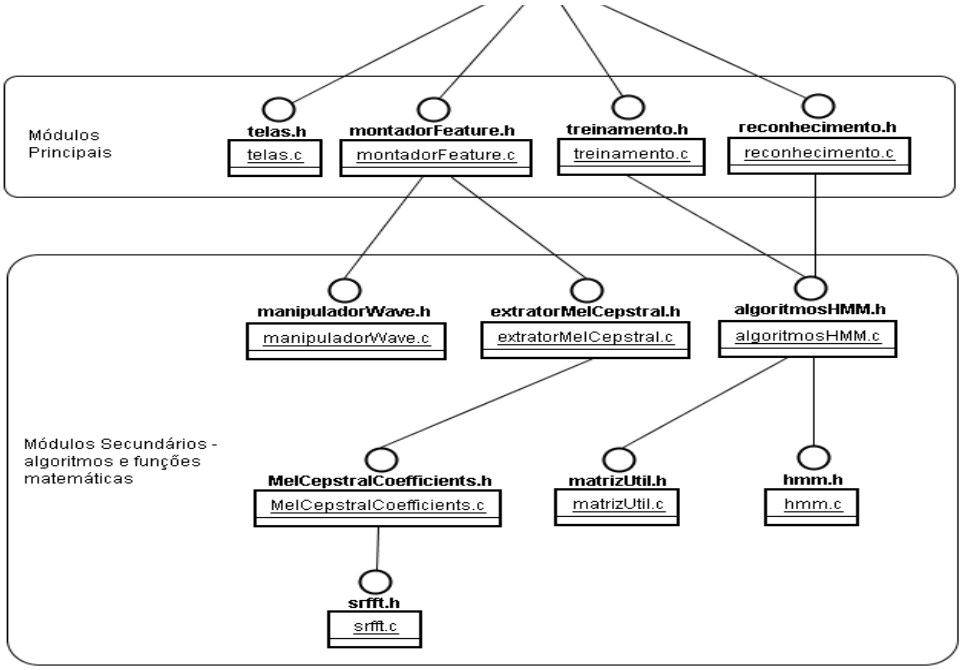

Figura 4.1: Hierarquia do software produzido

- *montadorFeature.c*: recebe uma amostra de voz em formato WAVE. Ele é responsável por chamar as funções necessárias para montar um conjunto de vetores de características do sinal sonoro que represente a amostra de voz.

- *treinamento.c*: recebe um conjunto de vetores de características. Ele chama as funções necessárias para criar um modelo escondido de Markov que represente esse conjunto.

- *reconhecimento.c*: recebe um conjunto de vetores de características e um ou mais HMM's. Ele chama as funções necessárias para a execução do algoritmo Forward e gera $P(X|\Phi)$ para cada modelagem. O modelo que gerar a maior probabilidade representará a palavra identificada.

Módulos secundários:

- *manipuladorWave.c*: como o próprio nome sugere, esse módulo é responsável por todas as manipulações do arquivo de áudio recebido. São elas: obtenção de uma *stream* a partir de um arquivo WAVE, retirada de cabeçalho, normalização das amostras, divisão em segmentos iguais.

- *extratorMelCepstral.c*: recebe um segmento do *stream* de áudio, denominada bloco. Ele é responsável por calcular os coeficientes Mel-cepstrais, Δ-Mel-cepstrais e $\Delta\Delta$-Mel-cepstrais correspondentes.

- *melCepstralCoefficients.c*: esse módulo possui as funções matemáticas necessárias para o cálculo dos coeficientes mel-cepstrais, exceto a FFT.

- *srfft.c*: módulo para o cálcula da FFT, extraído do livro (Malvar , 1992).

- *algoritmosHMM.c*: módulo onde os algoritmos Forward, Forward-Backward e Viterbi estão implementados.
- *matrizUtil.c*: módulo de auxílio aos algoritmos do módulo *algoritmosHMM.c* para a manipulação matemática de matrizes.
- *hmm.c*: esse módulo gerencia a memória para A estrutura de dados de um modelo e, além disso, define as probabilidades iniciais e executa a normalização que deve ocorrer nas matrizes A, B e π de um modelo Φ

O sistema de aprendizado/reconhecimento está dividido em três modos de execução: geração de alfabeto, aprendizado e reconhecimento. A seguir, as seções que os detalham.

4.1 Modo de geração de alfabeto

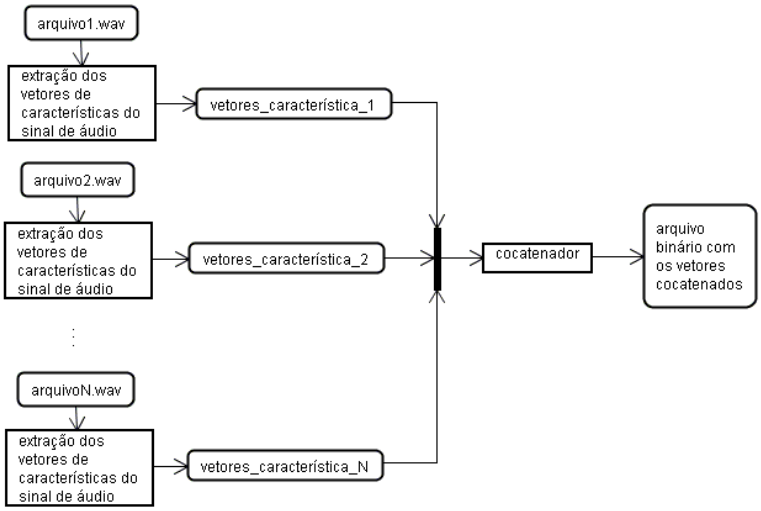

Figura 4.2: Diagrama de blocos - geração de arquivo binário com vetores de características de N sinais sonoros

Esse modo de execução apenas utiliza a funcionalidade do programa de extrair vetores de características dos sinais sonoros. Esses vetores são concatenados e seus elementos são gravados sequencialmente em ponto flutuante de precisão simples. Isso é feito para posterior quantização vetorial. Este processo está esquematizado no diagrama 4.2. Por exemplo, se comandos como 'liga' e 'desliga' serão identificados em um determinado sistema, a entrada do programa são arquivos de áudio com amostras dessas palavras para que vetores de características correspondentes sejam quantizados vetorialmente. A forma de uso do software é:

```
reconhecedor N arq_1 arq_2 ... arq_N
```

4.2 Modo de treinamento

O objetivo desse modo de execução é extrair as características de sinais sonoros - que são amostras similares a amostras que serão identificadas - e criar um HMM capaz de reconhecer sinais sonoros similares. O modelo, representado por uma estrutura de dados, é gravado em um arquivo binário. Essa modelagem é utilizada no modo de reconhecimento. Essas etapas estão esquematizadas em dois diagramas: 4.3 e 4.4.

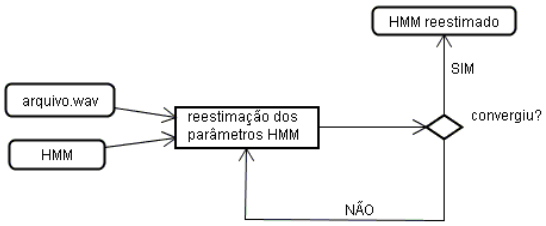

Figura 4.3: Diagrama de blocos - reestimação do modelo escondido de Markov

Por exemplo, se comandos como 'liga' e 'desliga' serão identificados em um determinado sistema, a entrada do programa são arquivos de áudio com amostras digitais dessas palavras. A saída tem modelagens, uma para cada palavra. A forma de uso do software é:

```
reconhecedor N arq_1 arq_2 ... arq_N alfabeto hmm_a_ser_gerado
```

4.3 Modo de reconhecimento

A forma de uso do programa nesse modo é:

```
reconhecedor arq_wave hmm alfabeto
```

O objetivo desse modo de execução é extrair as características de um sinal sonoro e obter a probabilidade dessas características serem geradas pelo modelo recebido como parâmetro. Por exemplo, se os comandos como 'liga' e 'desliga' serão identificados em um determinado sistema, o reconhecimento ocorrerá em três etapas - como no diagrama 4.5. As duas primeiras são utilizações do sistema de reconhecimento e a terceira é uma análise dos resultados:

1. `reconhecedor arquivo_a_reconhecer hmm_liga alfabeto`
   Dessa forma, obtém-se $P(X|\Phi_{liga})$
2. `reconhecedor arquivo_a_reconhecer hmm_desliga alfabeto`
   Dessa forma, obtém-se $P(X|\Phi_{desliga})$
3. As probabilidades são comparadas: o modelo com a maior probabilidade representa a palavra identificada.

O algoritmo Forward, coração da tarefa de reconhecimento da fala, pode ser resumido ao cálculo de α. Depois disso, apenas é preciso calcular $P(X|\Phi)$ da maneira descrita em 3.4.1.

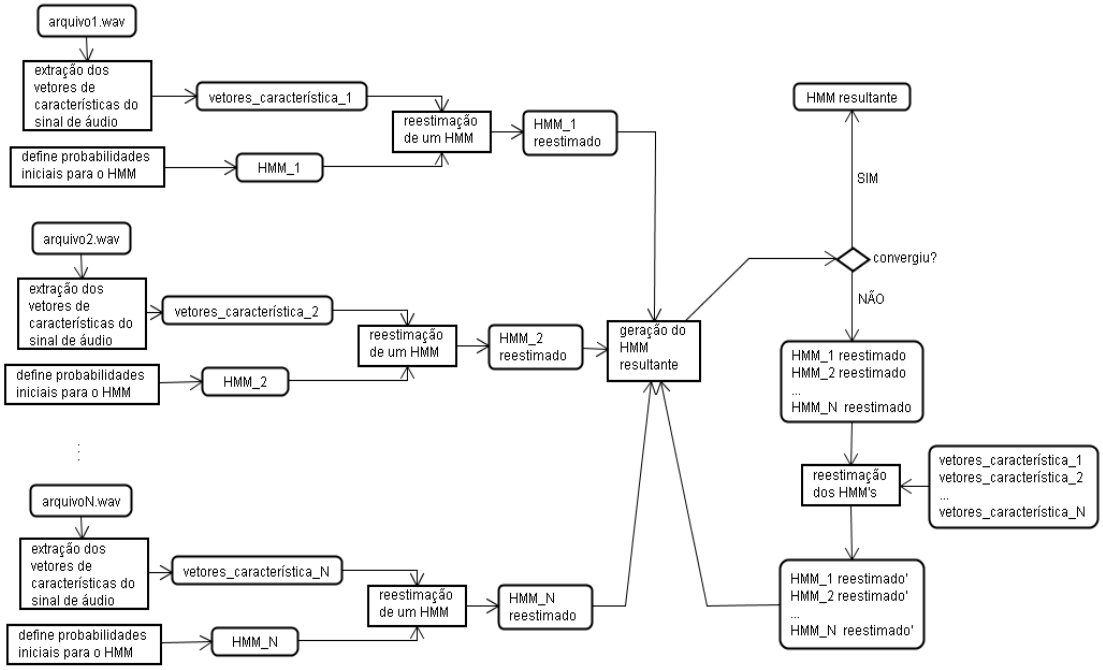

Figura 4.4: Diagrama de blocos - treinamento

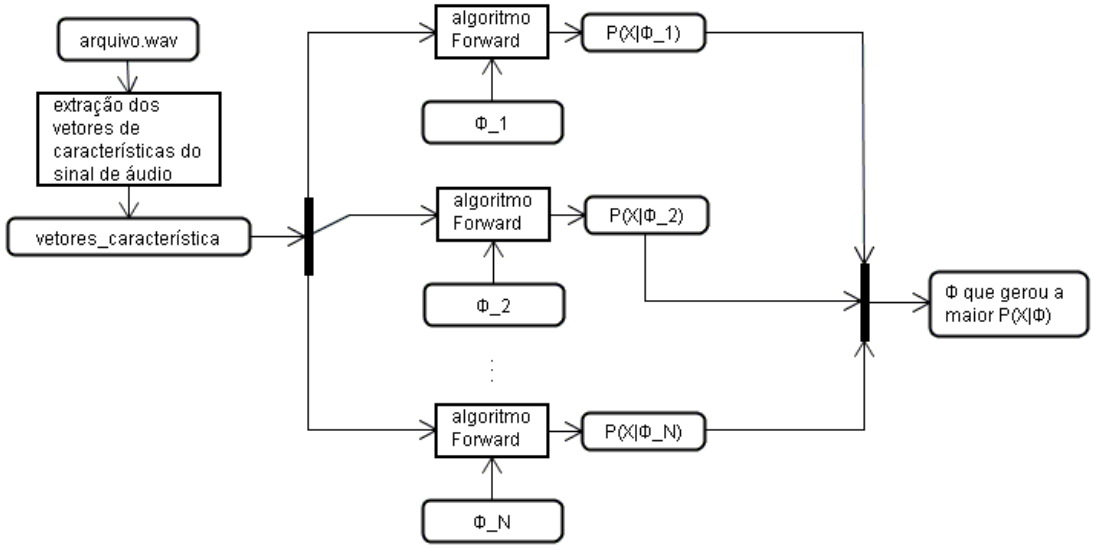

Figura 4.5: Diagrama de blocos - reconhecimento

Capítulo 5

Testes, resultados e análise

Os teste realizados neste trabalho utilizaram 150 comandos da voz, de apenas um locutor, gravados em arquivos no formato *wave*. Os comandos gravados correspondem a três palavras isoladas: *sim*, *não* e *talvez*. Ou seja, foram utilizadas 50 locuções da palavra *sim*; 50 da palavra *não* e 50 da palavra *talvez*. Essas locuções foram divididas em dois conjuntos, um para treinamento do sistema (40 locuções por palavra) e outro para validação (10 locuções por palavra).

Sabe-se que para gerar alfabetos que sejam generalistas, deve-se usar amostras diferenciadas. No caso de áudio, essa diferenciação se dá por locutores diferentes e por prosódias distintas. Amostras de uma mesma palavra que possuem velocidades de pronúncia distintas, posicionamento entre os silêncios presentes no início e no fim da amostra diferentes, entonação da pronúncia divergentes, dentre outras diferenças, são ditas amostras com prosódias distintas. A utilização de locuções com variabilidade prosódica tem grande impacto na qualidade do alfabeto gerado e na qualidade do HMM resultante após o treinamento. Ainda, é sabido que a retirada dos silêncios e de ruídos, a utilização de filtros de pré-ênfase, e a geração de alfabetos com uma grande gama de amostras são fatores decisivos para obtenção de bons resultados.

No *software* desenvolvido neste trabalho, por se tratar de um desenvolvimento inicial e um estudo preliminar , apenas um locutor foi usado. Pelo mesmo motivo, não foi gerado um grande número de amostras da mesma palavra com prosódias diferentes e nem foi implementada a retirada de silêncios e ruídos.

5.1 O procedimento de teste

Para o conjunto de treinamento, 120 locuções ao todo, foram construídos 9 alfabetos de vetores de características com o algoritmo LBG (2.3), varindo-se o seu tamanho (64, 128 e 256 vetores de código) e o tamanho da janela (512, 256 e 128 amostras). Esses parâmetros foram escolhidos por determinar grande impacto no modelo. O número de amostras por janela de processamento interfere na resolução tempo-freqüência e portanto tem impacto na extração de características dos sinais sonoros, como descrito no capítulo 2. O tamanho do alfabeto influencia diretamente no esforço computacional dos algoritmos e, como detalhado no capítulo 3, na representação do espaço de vetores de caracteríticas da quantização

vetorial (2.3). Portanto, foram utilizadas as seguintes configurações de teste:

1. Janela de 512 amostras e alfabeto de 256 vetores de códigos
2. Janela de 512 amostras e alfabeto de 128 vetores de códigos
3. Janela de 512 amostras e alfabeto de 64 vetores de códigos
4. Janela de 256 amostras e alfabeto de 256 vetores de códigos
5. Janela de 256 amostras e alfabeto de 128 vetores de códigos
6. Janela de 256 amostras e alfabeto de 64 vetores de códigos
7. Janela de 128 amostras e alfabeto de 256 vetores de códigos
8. Janela de 128 amostras e alfabeto de 128 vetores de códigos
9. Janela de 128 amostras e alfabeto de 64 vetores de códigos

Uma vez definidos os vetores de caracteríticas para os conjuntos de treinamento e validação, e para cada configuração de teste, foram construídos os modelos HMM (um para o *sim*, outro para *não* e outro para o *talvez*) com as 40 locuções de treinamento. Esses modelos foram testados com o conjunto de treinamento e de validação, 150 locuções ao todo, para observar a sua capacidade de generalização. As medidas de desempenho para os testes realizados foram:

1. Quantidade de acertos com locuções da palavra *talvez* e do conjunto de treinamento;
2. Quantidade de acertos com locuções da palavra *não* e do conjunto de treinamento;
3. Quantidade de acertos com locuções da palavra *sim* e do conjunto de treinamento;
4. Quantidade total de acertos do conjunto de treinamento (50 da palavra *sim*, 50 comandos da palavra *não* e 50 comandos da palavra *sim*);
5. Quantidade de comandos identificados como *talvez* e eram ou comandos da palavra *sim* da palavra *não*;
6. Quantidade de comandos identificados como *não* e eram ou comandos da palavra *sim* da palavra *talvez*;
7. Quantidade de comandos identificados como *sim* e eram ou comandos da palavra *talvez* da palavra *não*;
8. Quantidade de acertos com locuções da palavra *talvez* do conjunto de validação;
9. Quantidade de acertos com locuções da palavra *não* do conjunto de validação;

10. Quantidade de acertos com locuções da palavra *sim* do conjunto de validação;

11. Quantidade total de acertos do conjunto de validação.

5.2 Os resultados

Os resultados dos testes realizados são apresentados nas tabelas 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 e 5.9).

Tabela 5.1: Teste com janela de 512 amostras e alfabeto de 256 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	45	90,00%
	Acertos Não	42	84,00%
	Acertos Sim	43	86,00%
	Total de acertos	130	86,67%
	Falsos Talvez	6	6,00%
	Falsos Não	5	5,00%
	Falsos Sim	9	9,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	7	70,00%
	Acertos Não	4	40,00%
	Acertos Sim	5	50,00%
	Total de acertos	16	53,33%

Tabela 5.2: Teste com janela de 512 amostras e alfabeto de 128 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	44	88,00%
	Acertos Não	39	78,00%
	Acertos Sim	41	82,00%
	Total de acertos	124	82,67%
	Falsos Talvez	6	6,00%
	Falsos Não	6	6,00%
	Falsos Sim	14	14,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	7	70,00%
	Acertos Não	5	50,00%
	Acertos Sim	5	50,00%
	Total de acertos	17	56,67%

Tabela 5.3: Teste com janela de 512 amostras e alfabeto de 64 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	45	90,00%
	Acertos Não	33	66,00%
	Acertos Sim	40	80,00%
	Total de acertos	118	78,67%
	Falsos Talvez	8	8,00%
	Falsos Não	8	8,00%
	Falsos Sim	16	16,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	8	80,00%
	Acertos Não	3	60,00%
	Acertos Sim	6	30,00%
	Total de acertos	17	56,67%

Tabela 5.4: Teste com janela de 256 amostras e alfabeto de 256 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	44	88,00%
	Acertos Não	42	84,00%
	Acertos Sim	42	84,00%
	Total de acertos	128	85,33%
	Falsos Talvez	1	1,00%
	Falsos Não	10	10,00%
	Falsos Sim	11	11,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	5	50,00%
	Acertos Não	4	40,00%
	Acertos Sim	5	50,00%
	Total de acertos	14	46,67%

Tabela 5.5: Teste com janela de 256 amostras e alfabeto de 128 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	42	84,00%
	Acertos Não	37	74,00%
	Acertos Sim	36	72,00%
	Total de acertos	115	76,67%
	Falsos Talvez	5	5,00%
	Falsos Não	10	10,00%
	Falsos Sim	20	20,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	5	50,00%
	Acertos Não	5	50,00%
	Acertos Sim	5	50,00%
	Total de acertos	15	50,00%

Tabela 5.6: Teste com janela de 256 amostras e alfabeto de 64 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	43	86,00%
	Acertos Não	33	66,00%
	Acertos Sim	38	76,00%
	Total de acertos	130	86,67%
	Falsos Talvez	7	7,00%
	Falsos Não	7	7,00%
	Falsos Sim	22	22,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	6	60,00%
	Acertos Não	5	50,00%
	Acertos Sim	3	30,00%
	Total de acertos	14	46,67%

Tabela 5.7: Teste com janela de 128 amostras e alfabeto de 256 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	45	90,00%
	Acertos Não	44	88,00%
	Acertos Sim	41	82,00%
	Total de acertos	130	86,67%
	Falsos Talvez	2	2,00%
	Falsos Não	9	9,00%
	Falsos Sim	9	9,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	6	60,00%
	Acertos Não	6	60,00%
	Acertos Sim	5	50,00%
	Total de acertos	17	56,67%

Tabela 5.8: Teste com janela de 128 amostras e alfabeto de 128 vetores de código

	Medida de Desempenho	**Quantidade**	**Percentual**
Treinamento	Acertos Talvez	40	80,00%
	Acertos Não	44	88,00%
	Acertos Sim	35	70,00%
	Total de acertos	119	79,33%
	Falsos Talvez	3	3,00%
	Falsos Não	19	19,00%
	Falsos Sim	9	9,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	6	60,00%
	Acertos Não	6	60,00%
	Acertos Sim	4	40,00%
	Total de acertos	16	53,33%

Tabela 5.9: Teste com janela de 128 amostras e alfabeto de 64 vetores de código

	Medida de Desempenho	Quantidade	Percentual
Treinamento	Acertos Talvez	40	80,00%
	Acertos Não	39	78,00%
	Acertos Sim	37	74,00%
	Total de acertos	116	77,33%
	Falsos Talvez	4	4,00%
	Falsos Não	14	14,00%
	Falsos Sim	16	16,00%
Validação	**Medida de Desempenho**	**Quantidade**	**Percentual**
	Acertos Talvez	5	50,00%
	Acertos Não	4	50,00%
	Acertos Sim	5	40,00%
	Total de acertos	14	46,67%

5.3 Análise dos resultados

Analisando as tabelas da seção anterior, observa-se resultados promissores para o sistema desenvolvido, uma vez que a taxa de reconhecimento correto para o conjunto de treinamento é de 86,67% para uma configuração com janela de 128 amostras e um alfabeto de 256 vetores de código. Entretanto, para o conjunto de validação, as taxas de acerto com a mesma configuração foi de 60% para as palavra *talvez* e *não*, e 50% para a palavra *sim*, indicando que a variabilidade da prosódia nas locuções esteja atrapalhando o reconhecimento. Como o conjunto de validação é pequeno (10 locuções), seria possível especular que com o aumento do conjunto de validação os resultados poderiam melhorar, já que provavelmente surgiriam mais locuções com prosódias próximas das existentes nas locuções de treinamento. Entretanto, acredita-se que essa queda de desempenho deve-se, principalmente, à falta de um algoritmo de remoção de silêncio. Por exemplo, observa-se que no teste com janela de 512 amostras e alfabeto de 64 vetores de código 5.3, para a palavra *talvez* o sistema alcançou uma taxa de 80% de acerto, equanto para as palavras *sim* e *não*, os resultados foram de 30% e 60%, respectivamente. Ou seja, sem a remoção do silêncio, janelas mais longas favorecem a palavra mais longa (*talvez*), em detrimento das palavras curtas (*sim* e *não*). Além disso, nesse mesmo teste, a taxa de acerto para a palavra *não* só foi superior ao da palavra *sim* porque esta palavra apresenta um fonema nasal o que facilitaria a classificação mesmo com um alfabeto pequeno. Dessa forma, acredita-se que um algoritmo de remoção do silêncio tornaria o sistema mais independente da prosódia e os resultados para o conjunto de validação compatíveis com o conjunto de treinamento.

Outro fator que provavelmente tornou o sistema testado muito dependente da prosódia foi o seu treinamento com apenas um locutor. Acontece que, com apenas um locutor no treinamento, os modelos HMMs concentraram a classificação em caracteríticas estatíticas dos rítmos de locução e acentuação das locuções para o

conjunto de treinamento, o que limita sua capacidade de generalização.

Continuando a análise dos resultados, é possível especular qual seria a melhor configuração de tamanho da janela de análise e tamanho do alfabeto de vetores de código.

Para saber qual o tamanho do alfabeto mais adequado para o reconhecimento dos três comandos em questão, compara-se as quatro últimas linhas (8 - 11) das tabelas 5.1, 5.2 e 5.3. Conclui-se que, apesar da boa porcentagem de acerto para o conjunto de validação na tabela 5.3, os resultados individuais não são satisfatórios, já que o acerto do *sim* (linha 10) foi de 30%. Tal análise também pode ser feita para as três tabelas onde o tamanho da janela é de 256 amostras (tabelas 5.4, 5.5 e 5.6) e para as três outras (tabelas 5.7, 5.8 e 5.9), onde o número de amostras por bloco é de 128.

Dessa forma, descarta-se a utilização de alfabetos de tamanho 64. Isso era esperado, pois dividir a característica de uma parte de áudio em 64 partes não deve ser suficiente frente à grande diferenciação entre partes de ondas sonoras, como visto em 2.1. Porém, o teste é importante tanto para confirmação desse fato quanto para comparação.

Os dois outros tamanhos de alfabeto, 128 e 256, conseguiram resultados muito parecidos. Porém, as linhas 5, 6 e 7 dão uma dica importante: o alfabeto de tamanho 256 obteve um número menor de falsos positivos. No entanto, o tamanho reduzido do conjunto de validação - 10 para cada uma das três palavras - é um fator que dificulta essa análise. Portanto, a partir desses resultados, é difícil afirmar qual seria o mais adequado. Para descobrir qual o tamanho mais adequado para o reconhecimento dos três comandos em questão, compara-se as tabelas 5.1, 5.4 e 5.7, que têm os resultados com o alfabeto de tamanho 256 para as três janelas usadas: 512, 256 e 128. Nenhuma diferença significativa pode ser encontrada para esse tamanho de alfabeto. Porém, se a mesma análise for feita com as tabelas que contêm os dados para as três janelas com o alfabeto de 128 (tabelas 5.2, 5.5 e 5.8), vemos um resultado ruim na linha 4 (acertos dentre os 150 comandos) para os alfabetos de tamanho 64 e 128, o que sugere que a janela de 512 seja mais adequada - além disso, esse fato enfatiza a idéia de que o maior alfabeto fornece o melhor resultado. No entanto, novamente, a pequena quantidade de locuções de validação dificulta a análise. Portanto, existe um indício de que a janela de 512 fornece melhores resultados, mas a análise não é conclusiva.

Neste ponto, vale a pena justificar o tamanho reduzido do conjunto de validação. Recomenda-se que o conjunto de treinamento seja entre 4 e 5 vezes maior do que o conjunto de validação, dessa forma utilizar um conjunto de validação de 50 locuções por palavra significaria pré-processar e treinar o modelo com pelo menos 600 locuções. Considerando que a etapa de digitalização das locuções é feita de forma manual e que o processo de treinamento do sistema é custoso, e sendo este um estudo preliminar, decidiu-se limitar o conjunto de validação a 10 locuções por palavra.

A continuação desse projeto deve passar por uma fase de testes maior e mais conclusiva. Ou seja, um número maior de locuções, de diferentes locutores, devem ser utilizadas para a geração do alfabeto e treinamento do sistema de reconhecimento. Deve-se também, testar o sistema com um conjunto de validação maior. Por fim, especula-se que a utilização de um algoritmo de remoção do silêncio deve

melhorar consideravelmente o desempenho do sistema.

Capítulo 6

Conclusão

O objetivo deste trabalho foi desenvolver um sistema de reconhecimento de palavras isoladas. Para atingir este objetivo, foi desenvolvido um sistema que:

- Processa (por exemplo, normalizar) sinais de áudio no formato WAVE;
- Extrai vetores de atributos de sinais de voz, utilizando coeificentes Mel-cepstrais e seus derivados;
- Quantiza vetorialmente os vetores de atributos;
- Constrói modelos escondidos de Markov discretos;
- Reconhece palavras isoladas utilizandos os coeficientes Mel-cepstrais quantizados como entrada dos modelos de Markov.

O *software* desenvolvido tem um caráter bastante modularizado, assim, desde que seja especificado a taxa de amostragem, o tamanho do cabeçalho do arquivo WAVE de entretada e a duração da locução, é possível alterar parâmetros como: o número de estados do HMM, tamanho do vetor de características, tamanho da janela de análise e tamanho do alfabeto.

Apresentou-se uma solução com algoritmos clássicos. Eles possibilitaram o desenvolvimento de um reconhecedor de discurso que exige um esforço computacional razoavelmente pequeno.

Analisando os teste realizados, observou-se resultados promissores para o sistema desenvolvido, uma vez que a taxa de reconhecimento correto para o conjunto de treinamento foi de 86,67% para uma configuração com janela de 128 amostras e um alfabeto de 256 vetores de código. Entretanto, as taxas de acerto com a mesma configuração foi em torno de 60% para o conjunto de validação. Análise mais profundas indicam que um algoritmo de extração de silêncio poderia tornar os resultados para o conjunto de validação compatível com os resultados para o conjunto de treinamento.

Além disso, acredita-se que o treinamento do sistema com mais de um locutor pode melhorar o seu desempenho. E que para uma avaliação precisa de qual o melhor tamanho da janela de análise e do alfabeto de vetores de código, faz-se necessário um conjunto de validação com um número maior de locuções.

Dessa forma, conclui-se que o presente trabalho atingiu o seu objetivo, produzindo um sistema de reconhecimento de palavras isoladas, implementado de forma modular e que fornece resultados promissores em termos de percentual de acertos de reconhecimento.

Referências Bibliográficas

Davis S.B. & Mermelstein P., Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences, IEEE Trans. on ASSP, 28:357-366, 1980.

Duhamel P. & Vetterli M., Fast fourier transforms: A tutorial review and a state of the art, Signal Processing, 19:259-299, 1990.

Furui S., Speaker-independent isolated word recognition using dynamic features of speech spectrum, IEEE Trans. ASSP, 34:52-59, 1986.

Huang X. & Acero A. & Hon H., Spoken Language Processing - A Guide to Theory, Algorithm, and System Development, 1a ed, Raj Reddy, NJ: Prentice Hall, 2001.

Juang B.H. & Furui S., Automatic recognition and understanding of spoken language - a first step toward natural human-machine communication, Proceedings of the IEEE, 88(8):20, 2000.

Kim S. & Yang J., Speaker Identification System Using HMM and Mel Frequency Cepstral Coefficient, Pattern Recognition and Clustering, 1(1):1-11, 2006.

Linde Y., Buzo A. & Gray R.M., An algorithm for vector quantization design. IEEE Trans. Commun., 28, pp. 84-95, 1980.

MacQueen J., On convergence of k-means and partitions with minimum average variance, Ann. Math. Statist., 36:, 1965.

Malvar H.S., Signal Processing with Lapped Transforms, Artech House Publishers, 1992.

Oliveira L.E. & Morita M.E., Introdução aos Modelos Escondidos de Markov (HMM), PPGIA - Programa de Pós-Graduação em Informática Aplicada - PUC-PR, 1(1):1-16, 2002.

Oppenheim A.V., Schafer R.W. & Buck J.R., Discrete-Time Signal Processing, 2a ed, Upper Saddle River, NJ: Prentice Hall, 1999.

Rabiner L.R. & Juang B.H., An introduction to hidden markov models, IEEE ASSP Magazine, 3(1):4-16, 1986.

Rabiner L.R., A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proceedings of the IEEE, 77(2):1-30, 1989.

Schafer R.W. & Rabiner L.R., Digital Representations of Speech Signals, Proceedings of the IEEE, 67(1):1-19, 1975.

Apêndice A

Apêndice

A.1 Codificação do software implementado neste trabalho

Os códigos a seguir estão detalhados no capítulo 4 e esquematizados no diagrama 4.1

A.1.1 Código de main.c e defines.h

main.c:

```
//includes de bibliotecas C
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

//includes de bibliotecas internas
#include "extratorMelCepstral.h"
#include "hmm.h"
#include "emissor.h"
#include "MelCepstralCoefficients.h"
#include "manipuladorWave.h"
#include "telas.h"
#include "treinamento.h"
#include "motadorFeature.h"
#include "reconhecimento.h"

//include dos defines
#include "defines.h"

double ** carregaCodeBook(char * nomeArquivo){
    int i,j;
    double ** codebook;
    float valorLido;

    //alocação dinâmica para o codebook
    codebook = (double**) calloc(NUMERO_SAIDAS,sizeof(double));
    for(i=0; i<NUMERO_SAIDAS; i++) codebook[i] = (double*)calloc(TAMANHO_DA_OBSERVACAO,sizeof(double));

    //abertura do arquivo
    FILE * arqCodeBook;

    arqCodeBook = fopen(nomeArquivo,"rb");
    if(arqCodeBook==NULL) {
        printf("erro: Nao foi possivel abrir o codebook"); getchar();
        exit(1);
    }
```

```
    //leitura dos valors do codebook
    valorLido = 0.0;
    for(i=0; i<NUMERO_SAIDAS && !feof(arqCodeBook); i++){
        for(j=0; j<TAMANHO_DA_OBSERVACAO; j++){
            fread(&valorLido, sizeof(float), 1, arqCodeBook);
            codebook[i][j] = (double) valorLido;
            //printf("codebook[%d][%d]: %lf\n", i, j, codebook[i][j]);
        }

    }

    //fecha a stream do arquivo
    fclose(arqCodeBook);

    return codebook;
}

char* criarNomeArqTxt(char * arg){
    int i,tamArg;
    char * resposta;

    //verifica o tamanho do argumento recebido
    for(tamArg=0; arg[tamArg]!='\0'; tamArg++);

    //aloca memória para a string
    resposta = (char*)calloc(10+tamArg, sizeof(char));

    //escreve o nome
    resposta[0] = 'p';
    resposta[1] = 'r';
    resposta[2] = 'o';
    resposta[3] = 'b';
    resposta[4] = '_';
    for(i=0; i<tamArg; i++) resposta[i+5] = arg[i];
    resposta[5+tamArg] = '.';
    resposta[6+tamArg] = 't';
    resposta[7+tamArg] = 'x';
    resposta[8+tamArg] = 't';
    resposta[9+tamArg] = '\0';

    return resposta;
}

char* criarNomeArqBin(char * arg){
    int i,tamArg;
    char * resposta;

    //verifica o tamanho do argumento recebido
    for(tamArg=0; arg[tamArg]!='\0'; tamArg++);

    //aloca memória para a string
    resposta = (char*)calloc(10+tamArg, sizeof(char));

    //escreve o nome
    resposta[0] = 'p';
    resposta[1] = 'r';
    resposta[2] = 'o';
    resposta[3] = 'b';
    resposta[4] = '_';
    resposta[5] = 'b';
    resposta[6] = 'i';
    resposta[7] = 'n';
    resposta[8] = '_';
    for(i=0; i<tamArg; i++) resposta[i+9] = arg[i];
    resposta[9+tamArg] = '\0';

    return resposta;
}
```

```
//programa
int main(int argc, char ** argv){

    //declaração de todas as variáveis que serão usadas
    int i,j,k,numArq,numBlocos;
    double delta;
    double **codebook;
double ***feature__;
HMM **estruturaHMM;
HMM *hmmEstimado1, *hmmEstimado2;


    //desenha a tela inicial e, em caso de parametros invalidos, retorna zero
if(!(numArq = desenharTelaInicial(argc,argv[1]))) return 0;

//informa os parametros que estão sendo usados para o treinamento / reconhecimento
    if(!informaParametrosPassados(argv)) return 0; //se o retorno for zero, aconteceu um erro inesperado

    if(MODO_TREINAMENTO==1) {
        //alocação de memória para as features que serão criadas
        feature__ = (double***)calloc(numArq,sizeof(double));

        //alocação de memória para os hmms que serão gerados
        estruturaHMM = (HMM**)calloc(numArq,sizeof(HMM));

        //obtenção das features a partir dos arquivos passados como argumentos
        for(i=0; i<numArq; i++) feature__[i] = montarFeature(argv[i+2], &numBlocos);

        //normalização do último elemento de cada feature
        //for(i=0; i<numArq; i++) normalizarEnergiaDoBloco(feature[i], numBlocos);

        //carrega o codebook
        codebook = carregaCodeBook(argv[3+numArq]);


        //primeiro treinamento com todas as features
        for(i=0; i<numArq; i++) {
            estruturaHMM[i] = treinar1aVez(feature__[i], numBlocos, codebook);
        }

        //imprimirCodeBook(estruturaHMM[0], 36); getchar();

        //treinamentos a partir da segunda vez (com todas as features
        delta = 1.0;

        for(i=0; i<11 && delta>CONVERGENCIA_BAUM; i++){
            //gera uma estimação do hmmResultante
            hmmEstimado1 = (HMM*)geraHmmResultante(estruturaHMM, numArq, TAMANHO_DA_OBSERVACAO, numBlocos);

            //segundo e demais treinamentos
            for(j=0; j<numArq; j++) estruturaHMM[j] = treinar2aVezEmDiante(estruturaHMM[j], feature__[j], numBlocos);

            //gera outra estimação do hmmResultante
            hmmEstimado2 = (HMM*)geraHmmResultante(estruturaHMM, numArq, TAMANHO_DA_OBSERVACAO, numBlocos);

            //verifica se já aconteceu a convergência
            calculaDelta(&delta,hmmEstimado1,hmmEstimado2, feature__, numArq, numBlocos, TAMANHO_DA_OBSERVACAO);
        }


        printf("\n\ntreinamento para %s:\n",argv[2+numArq]);
        printf("numero iteracoes no algoritmo baum-welch...: %d\n",i);
        printf("numero da convergencia proposto.....: %lf = %e\n",CONVERGENCIA_BAUM,CONVERGENCIA_BAUM);
        printf("numero de convergencia atingido.....: %lf = %e\n\n\n\n",delta,delta); getchar();

        gravarHMMTreinado(hmmEstimado2, argv[2+numArq], NUMERO_SAIDAS, TAMANHO_DA_OBSERVACAO);
    }
    else if(MODO_RECONHECIMENTO==1){
        //alocação de memória para os hmms que foram passados como parametro
        estruturaHMM = (HMM**)calloc(numArq,sizeof(HMM));
```

```
//carregamento do codebook
codebook = carregaCodeBook(argv[3+numArq]);

//carregamento dos hmm's passados e inclusão do codebook
for(i=0; i<numArq; i++) {
    estruturaHMM[i] = (HMM*)carregarHMM(argv[i+2], NUMERO_ESTADOS, NUMERO_SAIDAS, TAMANHO_DA_OBSERVACAO, numBlo
}

//alocação de memória para as features que serão criadas
feature__ = (double***)calloc(1,sizeof(double));

//obtenção da feature a partir do arquivo passado como argumento
feature__[0] = montarFeature(argv[2+numArq], &numBlocos);

//roda o forward
i = reconhecer(numArq, estruturaHMM, feature__[0], numBlocos, codebook);

//escreve na tela o resultado
printf(" -> %s\t%s\n", argv[2+numArq],argv[2+i]);

//escreve no arquivo de resultados o mesmo que foi escrito na tela
FILE * arqRes;
arqRes = fopen("salvador.txt","a");
if(arqRes==NULL){
    printf("Erro na abertura do arquivo texto que conterah o resultado, que nao serah escrito");
    getchar();
}
else fprintf(arqRes,"%s\t%s\n", argv[2+numArq],argv[2+i]);

/*//carrega o hmm passado como parametro
HMM * estruturaHMM = (HMM*)carregarHMM(argv[2], NUMERO_ESTADOS, NUMERO_SAIDAS, TAMANHO_DA_OBSERVACAO, numBlocos

//carrega o codebook
//codebook = carregaCodeBook(argv[3]);



//põe o codebook na estruturaHMM
estruturaHMM->codebook = carregaCodeBook(argv[3]);

//alocação de memória para as features que serão criadas
feature__ = (double***)calloc(1,sizeof(double));

//obtenção da feature a partir do arquivo passado como argumento
feature__[0] = montarFeature(argv[1], &numBlocos);

//normalização do último elemento de cada feature
//normalizarEnergiaDoBloco(feature[0], numBlocos);

//probabilidade da palavra recebida ser a esperada pela modelagem
double probFinal;

//imprimirCodeBook(estruturaHMM, 36); getchar();

//roda o forward
forward(estruturaHMM, feature__[0], numBlocos, TAMANHO_DA_OBSERVACAO, &probFinal);

//escreve na tela o resultado
printf("probabilidade de ser a palavra:\n");
printf(" -> em notacao cientifica.: %e\n", probFinal);
printf(" -> sem notacao cientifica: %lf\n", probFinal);

//escreve em arquivo binário e texto o resultado
FILE * arqTxt, * arqBin;
arqTxt = fopen(criarNomeArqTxt(argv[2]),"a");
arqBin = fopen(criarNomeArqBin(argv[2]),"wb");
if(arqTxt==NULL) {
    printf("O reconhecimento ocorreu, mas nao foi possivel gravar o resultado em arquivo texto");
    getchar();
```

```
        }
        else {
            fprintf(arqTxt, "%e\t%s\n", probFinal, argv[1]);
            fclose(arqTxt);
        }
        if(arqBin==NULL) {
            printf("O reconhecimento ocorreu, mas nao foi possivel gravar o resultado em arquivo binario");
            getchar();
        }
        else {
            fwrite(&probFinal, sizeof(double), 1, arqBin);
            fclose(arqBin);
        }*/
    }
    else if(GERAR_CODEBOOK==1){
        //alocação de memória para as features que serão criadas
        feature__ = (double***)calloc(numArq,sizeof(double));

        //obtenção das features a partir dos arquivos passados como argumentos
        for(i=0; i<numArq; i++) feature__[i] = montarFeature(argv[i+2], &numBlocos);

        //normalização do último elemento de cada feature
        //for(i=0; i<numArq; i++) normalizarEnergiaDoBloco(feature[i], numBlocos);

        //arquivo que conterá o cbkFonte
        FILE * arqCbkFonte;
        arqCbkFonte = fopen(argv[2+numArq],"wb");
        float valor = 0.0;
        int contador = 0;
        for(k=0; k<numArq; k++){
            for(i=0; i<numBlocos; i++){
                for(j=0; j<TAMANHO_DA_OBSERVACAO; j++){
                    valor = (float)feature__[k][i][j];
                    fwrite(&valor, sizeof(float), 1, arqCbkFonte);
                    contador++;
                }
            }
        }
        fclose(arqCbkFonte);
        printf("numero de vetores escritos: %d", contador/TAMANHO_DA_OBSERVACAO);
        getchar();
    }

    printf("\n\n");
    printf("##############################################\n");
    printf("##                                          ##\n");
    printf("##                 FIM                      ##\n");
    printf("##                                          ##\n");
    printf("##############################################\n");
    //getchar();

    return 0;
}
```

defines.h:

```
#define NUMERO_DE_COEFICIENTES 12 //número de coeficientes mel cepstrais que serão calculados
#define TAMANHO_DA_OBSERVACAO 37 //número de coeficientes mel cepstrais que serão calculados
#define TAMANHO_CABECALHO_WAV 22 //tamanho do cabecalho do arquivo de áudio .WAV que foi capturado
#define NUM_AMOSTRAS_POR_BLOCO 128 //número de amostras de áudio que estarão em cada bloco
#define TAXA_AMOSTRAGEM 16000 //taxa em que o áudio foi amostrado durante a captação
#define TEMPO_COMANDO_VOZ_ms 2048 //tempo em que o dispositivo de captura de áudio ficou aberto
#define LIMITE_FREQ_INFERIOR 20.0 //limite da frequencia inferior usuada para calcular as fronteiras do cálculo
#define LIMITE_FREQ_SUPERIOR 8000.0 //limite da frequencia superior usuada para calcular as fronteiras do cálculo (meta
#define NUMERO_ESTADOS 5 //número de estados da modelagem HMM
#define NUMERO_SAIDAS 256 //número de saídas possíveis para o HMM
#define CONVERGENCIA_BAUM 0.40 //limite de convergência para o cálculo de reestimação dos parâmetros do hmm

//defines para setar como o programa será executado
#define MODO_TREINAMENTO 0
#define MODO_RECONHECIMENTO 1
```

```
#define GERAR_CODEBOOK 0
```

A.1.2 Código dos 4 módulos principais

Módulo principal 1 - telas.c:

```
#include <stdio.h>
#include "defines.h"

int criaDigito(char digito){
    switch(digito){
        case '0': return 0;
        case '1': return 1;
        case '2': return 2;
        case '3': return 3;
        case '4': return 4;
        case '5': return 5;
        case '6': return 6;
        case '7': return 7;
        case '8': return 8;
        case '9': return 9;
        default: return (-1);
    }
}

int verificarArgumento1(char * argumento1){

    int tamanhoPar,i,multiplicador;
    int * digito;
    int resposta;

    for(tamanhoPar=0; argumento1[tamanhoPar]!='\0'; tamanhoPar++);

    //alocação dos dígitos
    digito = (int*)calloc(tamanhoPar, sizeof(int));

    //for para obter os digitos
    for(i=0; i<tamanhoPar; i++){
        if((digito[i] = criaDigito(argumento1[i])) == -1) return -1;
    }



    //for para calcular o número
    multiplicador = 1;
    resposta = 0;
    for(i=tamanhoPar-1; i>=0; i--){
        resposta += digito[i]*multiplicador;

        multiplicador *= 10;
    }

    return resposta;


}

int desenharTelaInicial(int argc, char * argumento1){
    int numArg;
    if(MODO_TREINAMENTO==1 && MODO_RECONHECIMENTO==0 && GERAR_CODEBOOK==0){
        if(argc<2){
            printf("Numero de argumentos invalidos. \n");
            printf("Forma de uso no modo de treinamento:\n");
            printf("\n      programa numeroArqN arquivo1 arquivo2 ... arquivoN nomeHmm nomeCodeBook\n");
            getchar();
            return 0;
        }
```

```
    if((numArg = verificarArgumento1(argumento1)) != (-1)){
        if(argc!=(4+numArg)) {
            printf("Numero de argumentos invalidos. \n");
            printf("Forma de uso no modo de treinamento:\n");
            printf("\n      programa numeroArqN arquivo1 arquivo2 ... arquivoN nomeHmm nomeCodeBook\n");
            getchar();
            return 0;
        }
        else {
            printf("\n");
            printf("############################\n");
            printf("##                        ##\n");
            printf("##  MODO DE TREINAMENTO   ##\n");
            printf("##                        ##\n");
            printf("############################\n\n");
            return numArg;
        }
    }
    else{
        printf("Numero de argumentos invalidos. \n");
        printf("Forma de uso no modo de treinamento:\n");
        printf("\n      programa numeroArqN arquivo1 arquivo2 ... arquivoN nomeHmm nomeCodeBook\n");
        getchar();
        return 0;
    }
}
else if(MODO_TREINAMENTO==0 && MODO_RECONHECIMENTO==1 && GERAR_CODEBOOK==0){
    if(argc<2){
        printf("Numero de argumentos invalidos. \n");
        printf("Forma de uso no modo de reconhecimento, onde 'N' eh o nro de HMM's:\n");
        printf("\n      programa N hmm1 hmm2 ... hmmN arqWave codebook\n");
        getchar();
        return 0;
    }

    if((numArg = verificarArgumento1(argumento1)) != (-1)){
        if(argc!=(4+numArg)) {
            printf("Numero de argumentos invalidos. \n");
            printf("Forma de uso no modo de reconhecimento, onde 'N' eh o nro de HMM's:\n");
            printf("\n      programa N hmm1 hmm2 ... hmmN arqWave codebook\n");
            getchar();
            return 0;
        }
        else {
            printf("\n");
            printf("###############################\n");
            printf("##                           ##\n");
            printf("##  MODO DE RECONHECIMENTO   ##\n");
            printf("##                           ##\n");
            printf("###############################\n\n");
            return numArg;
        }
    }
    else{
        printf("Numero de argumentos invalidos. \n");
        printf("Forma de uso no modo de reconhecimento, onde 'N' eh o nro de HMM's:\n");
        printf("\n      programa N hmm1 hmm2 ... hmmN arqWave codebook\n");
        getchar();
        return 0;
    }
}
else if(MODO_TREINAMENTO==0 && MODO_RECONHECIMENTO==0 && GERAR_CODEBOOK==1) {
    if(argc<2){
        printf("Numero de argumentos invalidos. \n");
        printf("Forma de uso no modo de geracao de codebook:\n");
        printf("\n      programa numeroArqN arqWave1 arqWave2 ... arqWaveN nomeCodeBook\n");
        getchar();
        return 0;
    }
```

```
        if((numArg = verificarArgumento1(argumento1)) != (-1)){

            if(argc!=(3+numArg)) {
                printf("Numero de argumentos invalidos. \n");
                printf("Forma de uso no modo de geracao de codebook:\n");
                printf("\n      programa numeroArqN arqWave1 arqWave2 ... arqWaveN nomeCodeBook\n");
                getchar();
                return 0;
            }
            else {
                printf("\n");
                printf("####################################\n");
                printf("##                                ##\n");
                printf("##  MODO DE GERACAO DE CODEBOOK   ##\n");
                printf("##                                ##\n");
                printf("####################################\n\n");
                return numArg;
            }
        }
        else{
            printf("Numero de argumentos invalidos. \n");
            printf("Forma de uso no modo de geracao de codebook:\n");
            printf("\n      programa numeroArqN arqWave1 arqWave2 ... arqWaveN nomeCodeBook\n");
            getchar();
            return 0;
        }
    }
    else {
        printf("O arquivo defines.h deve conter:\n");
        printf("- Opcao 1:\n");
        printf("     #define MODO_TREINAMENTO 0\n");
        printf("     #define MODO_RECONHECIMENTO 1\n");
        printf("     #define GERAR_CODEBOOK 0\n");
        printf("- Opcao 2:\n");
        printf("     #define MODO_TREINAMENTO 1\n");
        printf("     #define MODO_RECONHECIMENTO 0\n");
        printf("     #define GERAR_CODEBOOK 0\n");
        printf("- Opcao 3:\n");
        printf("     #define MODO_TREINAMENTO 0\n");
        printf("     #define MODO_TREINAMENTO_1a_VEZ 0\n");
        printf("     #define MODO_RECONHECIMENTO 0\n");
        printf("     #define GERAR_CODEBOOK 1\n");
        printf("\nonde 1 indica SIM.\n");
        printf("QUALQUER OUTRA COMBINACAO IMPEDE A EXECUCAO DO PROGRAMA\n");
        getchar();
        return 0;
    }

}

int informaParametrosPassados(char ** argv){
    //prints mostrando os defines setados e os parametros passados
    printf("-------------------------------------------\n");
    printf("   tempo do comando........: %.3lf segundos\n",((double)TEMPO_COMANDO_VOZ_ms)/1000.0);
    printf("   taxa de amostragem......: %d KHz\n",TAXA_AMOSTRAGEM/1000);
    printf("   amostras por bloco......: %d\n",NUM_AMOSTRAS_POR_BLOCO);
    printf("   tamanho header wave.....: %d bytes\n",TAMANHO_CABECALHO_WAV*2);
    if(MODO_TREINAMENTO==1 && MODO_RECONHECIMENTO==0 && GERAR_CODEBOOK==0){
        printf("   numero de estados.......: %d\n",NUMERO_ESTADOS);
        printf("   numero de saidas........: %d\n",NUMERO_SAIDAS);
        printf("   nome do arquivo wave....: %s\n",argv[1]);
        printf("   convergencia baum.......: %e\n",CONVERGENCIA_BAUM);
        printf("   nome do codebook........: %s\n",argv[2]);
        printf("   nome HMM a ser gerado...: %s\n",argv[3]);
        printf("-------------------------------------------\n");
    }
    else if(MODO_TREINAMENTO==1 && MODO_RECONHECIMENTO==0 && GERAR_CODEBOOK==0){
        printf("   numero de estados.......: %d\n",NUMERO_ESTADOS);
        printf("   numero de saidas........: %d\n",NUMERO_SAIDAS);
        printf("   nome do arquivo wave....: %s\n",argv[1]);
```

```
        printf("   convergencia baum.......: %e\n",CONVERGENCIA_BAUM);
        printf("   nome HMM que sera usado.: %s\n",argv[2]);
        printf("   nome HMM a ser gerado...: %s\n",argv[3]);
        printf("-------------------------------------------\n");
    }
    else if(MODO_TREINAMENTO==0 && MODO_RECONHECIMENTO==1 && GERAR_CODEBOOK==0) {
        printf("   numero de estados.......: %d\n",NUMERO_ESTADOS);
        printf("   numero de saidas........: %d\n",NUMERO_SAIDAS);
        printf("   nome do arquivo wave....: %s\n",argv[1]);
        printf("   convergencia baum.......: %e\n",CONVERGENCIA_BAUM);
        printf("   nome HMM que sera usado.: %s\n",argv[2]);
        printf("-------------------------------------------\n");
    }
    else if(MODO_TREINAMENTO==0 && MODO_RECONHECIMENTO==0 && GERAR_CODEBOOK==1) {
        printf("   nome do arquivo wave....: %s\n",argv[1]);
        printf("   nome codebk a ser gerado: %s\n",argv[2]);
        printf("-------------------------------------------\n");
    }
    else {
        printf("ocorreu um erro inesperado. O programa sera finalizado");
        getchar();
        return 0;
    }

    return 1;
}
```

Módulo principal 2 - motadorFeature.c:

```
//includes de bibliotecas C
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

//includes de bibliotecas internas
#include "extratorMelCepstral.h"
#include "hmm.h"
#include "MelCepstralCoefficients.h"
#include "manipuladorWave.h"

//include dos defines
#include "defines.h"

double calculaEnergiaDoBloco(double * bloco){
    int i;
    double energiaBloco;

    energiaBloco = 0.0;
    for(i=0; i<NUM_AMOSTRAS_POR_BLOCO; i++) {
        energiaBloco += bloco[i] * bloco[i];
    }

    return energiaBloco;
}

double ** montarFeature(char * nomeArqWave, int *pNumBlocos){
    audio audiostream;
    double *stream, *streamsc, *streamn;
    double **blocos, **mfcc;
    int i, j, k, numBlocos, namostras;

    //obtenção da stream de audio passada como parâmetro
audiostream = obterStreamVozAPartirDoArquivoDeAudio(nomeArqWave,TAXA_AMOSTRAGEM,TEMPO_COMANDO_VOZ_ms,TAMANHO_CABECALHO_
stream = calloc(audiostream.nsamples, sizeof(double));
stream=audiostream.stream;
namostras=audiostream.nsamples-TAMANHO_CABECALHO_WAV;

//retira o cabeçalho do arquivo (apenas os dados servem para os cálculos dos coeficientes e etc)
streamsc = calloc(namostras,sizeof(double));
streamsc = retiraCabecalhoArquivoWav(stream, audiostream.nsamples, TAMANHO_CABECALHO_WAV);
free(stream);
```

```
//normaliza o áudio (objetivo aqui é nao diferenciar comandos dados com volume de voz alto e baixo)
streamn =  calloc(namostras, sizeof(double));
streamn = normalizaAudio(streamsc, namostras);

//PRÉ-ÊNFASE
streamn[0]=0;
for (i=1;i<namostras;i++) {
        streamn[i]=streamn[i]-0.95*streamn[i-1];
    }

    numBlocos = ceil(namostras/NUM_AMOSTRAS_POR_BLOCO); //ceil é a função "chão"
    *pNumBlocos = numBlocos;

    double s=0.5;
    numBlocos=(numBlocos/(1-s))-1;

//obtem a stream dividida em blocos (um vetor de vetores de double)
blocos = calloc(numBlocos, sizeof(double *));

for (i=0; i < numBlocos ; i++) {
blocos[i] = calloc(NUM_AMOSTRAS_POR_BLOCO, sizeof(double));
}
k=0;
for(i=0; i<numBlocos; i++){
for (j=0; j < NUM_AMOSTRAS_POR_BLOCO ; j++) {
blocos[i][j] = streamn[k];
k++;
}
k=k-((NUM_AMOSTRAS_POR_BLOCO*s)-1);
}
free(streamn);

    //alocação de espaço para os mfcc's que serão gerados (a alocação da segunda dimensão é feita na função de extraçã
    mfcc = (double**)calloc(numBlocos,sizeof(double));

   //iteração para o cálculo dos coeficientes mel cepstrais de cada bloco
   for(i=0; i<numBlocos; i++){
     extrairCoefientesMelCepstraisDeUmBloco(blocos[i], &(mfcc[i]), NUMERO_DE_COEFICIENTES, NUM_AMOSTRAS_POR_BLOCO, LIM
   }


    /*EXTRAÇÃO COEFICIENTES DELTA MEL-CEPSTRAIS*/
double **dmfcc;
dmfcc = (double**)calloc(numBlocos,sizeof(double));
for ( i=0; i < numBlocos; i++) {
dmfcc[i]=calloc(NUMERO_DE_COEFICIENTES,sizeof(double));
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
dmfcc[0][j]=0.1*(mfcc[1][j]+2*mfcc[2][j]);
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
dmfcc[1][j]=0.1*(-mfcc[0][j]+mfcc[2][j]+2*mfcc[3][j]);
}
for(i=2;i < numBlocos-2;i++){
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
dmfcc[i][j]=0.1*(-2*mfcc[i-2][j]-mfcc[i-1][j]+mfcc[i+1][j]+2*mfcc[i+2][j]);
}
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
dmfcc[numBlocos-2][j]=0.1*(-2*mfcc[numBlocos-4][j]-mfcc[numBlocos-3][j]+mfcc[numBlocos-1][j]);
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
dmfcc[numBlocos-1][j]=0.1*(-2*mfcc[numBlocos-3][j]-mfcc[numBlocos-2][j]);
}

/*EXTRAÇÃO COEFICIENTES DELTA DELTA MEL-CEPSTRAIS*/
double **ddmfcc;
ddmfcc = (double**)calloc(numBlocos,sizeof(double));
for ( i=0; i < numBlocos; i++) {
```

```
ddmfcc[i]=calloc(NUMERO_DE_COEFICIENTES,sizeof(double));
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
ddmfcc[0][j]=0.1*(dmfcc[1][j]+2*dmfcc[2][j]);
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
ddmfcc[1][j]=0.1*(-dmfcc[0][j]+dmfcc[2][j]+2*dmfcc[3][j]);
}
for(i=2;i < numBlocos-2;i++){
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
ddmfcc[i][j]=0.1*(-2*dmfcc[i-2][j]-dmfcc[i-1][j]+dmfcc[i+1][j]+2*dmfcc[i+2][j]);
}
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
ddmfcc[numBlocos-2][j]=0.1*(-2*dmfcc[numBlocos-4][j]-dmfcc[numBlocos-3][j]+dmfcc[numBlocos-1][j]);
}
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
ddmfcc[numBlocos-1][j]=0.1*(-2*dmfcc[numBlocos-3][j]-dmfcc[numBlocos-2][j]);
}

   /*VETOR DE CARACTERÍSTICAS*/
double **feature;
feature = (double**)calloc(numBlocos,sizeof(double));
for ( i=0; i < numBlocos; i++) {
feature[i]=calloc(TAMANHO_DA_OBSERVACAO,sizeof(double));
}
for (i=0; i < numBlocos; i++) {
for(j=0;j<NUMERO_DE_COEFICIENTES;j++) {
feature[i][j]=mfcc[i][j];
}
for(j=NUMERO_DE_COEFICIENTES;j<2*NUMERO_DE_COEFICIENTES;j++) {
feature[i][j]=dmfcc[i][j-NUMERO_DE_COEFICIENTES];
}
for(j=2*NUMERO_DE_COEFICIENTES;j<3*NUMERO_DE_COEFICIENTES;j++) {
feature[i][j]=ddmfcc[i][j-2*NUMERO_DE_COEFICIENTES];
}
}

//coloca a energia do bloco em feature
for(i=0; i<numBlocos; i++) feature[i][TAMANHO_DA_OBSERVACAO-1] = calculaEnergiaDoBloco(blocos[i]);

return feature;
}

int normalizarEnergiaDoBloco(double ** feature, int numeroDeObservacoes){
    int i;
    double soma;

    //cálculo da soma
    soma = 0.0;
    for(i=0; i<numeroDeObservacoes; i++) soma += feature[i][TAMANHO_DA_OBSERVACAO-1];

    //se a soma for zero, não se pode continuar (condição de existência da divisão)
    if(soma==0.0) return 0;

    //normalização
    for(i=0; i<numeroDeObservacoes; i++) feature[i][TAMANHO_DA_OBSERVACAO-1] /= soma;

    return 1;
}
```

Módulo principal 3 - treinamento.c:

```
#include <stdio.h>

#include "defines.h"
#include "hmm.h"
#include "emissor.h"
#include "matrizUtil.h"

HMM * treinar1aVez(double **feature, int numeroDeObservacoes, double **codebook){
```

```
    int t, *posicaoCbkDaObservacao;

    //alocacao da modelagem matemática
    HMM * estruturaHMM = (HMM*)criaEstruturaHMM(NUMERO_ESTADOS, NUMERO_SAIDAS, TAMANHO_DA_OBSERVACAO, numeroDeObservacc

    //alocação de memoria para a estrutura de dados posicaoCbkDaObservacao (relação entre a observação e a sua posição
    posicaoCbkDaObservacao = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //carrega o codebook na estrutura criada
    estruturaHMM->codebook = codebook;

    //define matrizes de probabilidades iniciais. Primeiramente com probabilidades iguais para todos os estados
    defineProbabilidadesIniciais(estruturaHMM);

    //calcula e coloca no HMM os elementos de posicaoCbkDaObservacao
    for(t=0; t<numeroDeObservacoes; t++) posicaoCbkDaObservacao[t] = definePosicaoSimbolo__(feature[t], estruturaHMM->c
    estruturaHMM->posicaoCbkDaObservacao = posicaoCbkDaObservacao;

    //treina modelagem de acordo com os coeficientes extraidos do sinal de áudio
    treinaModelagemHMM(estruturaHMM,feature,numeroDeObservacoes,TAMANHO_DA_OBSERVACAO,CONVERGENCIA_BAUM);

    return estruturaHMM;

}

HMM * treinar2aVezEmDiante(HMM * estruturaHMM, double **feature, int numeroDeObservacoes){
    //carrega o hmm passado como parametro
    //HMM * estruturaHMM = (HMM*)carregarHMM(nomeHmm, NUMERO_ESTADOS, NUMERO_SAIDAS, 3*NUMERO_DE_COEFICIENTES, numeroDe

    //imprimirHMM(estruturaHMM); getchar();
    //imprimirCodeBook(estruturaHMM, 3*NUMERO_DE_COEFICIENTES); getchar();

    //treina modelagem de acordo com os coeficientes extraidos do sinal de áudio
    treinaModelagemHMM(estruturaHMM,feature,numeroDeObservacoes,TAMANHO_DA_OBSERVACAO,CONVERGENCIA_BAUM);

    //imprimirHMM(estruturaHMM); getchar();

    return estruturaHMM;

}
```

Módulo principal 4 - reconhecimento.c:

```
#include <stdio.h>
#include "hmm.h"
#include "defines.h"
#include "algoritmosHMM.h"
#include "matrizUtil.h"

double obterNumAleatorioEntreZeroEEneMenosUm(int n){
    int i;
    int var[10];
    int numAleatorio;

    //obtém 10 números aleatórios
    for(i=0; i<10; i++) {
        var[i] = rand();
        if(var[i]<0.0) var[i]*=(-1.0);
    }

    //multiplica todos os numeros aleatorios obtidos
    numAleatorio = 0.0;
    for(i=0; i<10; i++) numAleatorio += var[i];

    //o número aleatório entre 0 e 2 está pronto
    return numAleatorio%(n);
}



int reconhecer(int numHMM, HMM ** estruturaHMM, double ** observacoes, int numeroDeObservacoes, double ** codebook){
```

```
    int i,t, indiceDaMaiorProb, *posicaoCbkDaObservacao, qtdeCorretorUsado;
    char * resultado;
    double * prob;

    //alocação de memória para as probabilidades de cada hmm
    prob = (double*)calloc(numHMM,sizeof(double));

    //alocação de memoria para a estrutura de dados posicaoCbkDaObservacao (relação entre a observação e a sua posição
    posicaoCbkDaObservacao = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //calcula os elementos de posicaoCbkDaObservacao
    for(t=0; t<numeroDeObservacoes; t++) posicaoCbkDaObservacao[t] = definePosicaoSimbolo__(observacoes[t], codebook, c

    //calculo das probabilidades de 1 até numHMM
    for(i=0; i<numHMM; i++) {
        estruturaHMM[i]->posicaoCbkDaObservacao = posicaoCbkDaObservacao;
        forward(estruturaHMM[i], observacoes, numeroDeObservacoes, TAMANHO_DA_OBSERVACAO, &(prob[i]));
    }
    //for(i=0; i<numHMM; i++) printf("----> prob[%d] = %e\n",i,prob[i]);
    //getchar();
    //verificação de qual probabilidade é a maior (o que possui o menor fatorT
    indiceDaMaiorProb = obterNumAleatorioEntreZeroEEneMenosUm(numHMM);
    for(i=0; i<numHMM; i++){
        if(prob[indiceDaMaiorProb]<prob[i]) indiceDaMaiorProb = i;
    }

    return indiceDaMaiorProb;
}
```

A.1.3 Código dos 7 módulos secundários

Módulo secundário 1 - manipuladorWave.c:

```
#include <stdio.h>
#include "manipuladorWave.h"

audio obterStreamVozAPartirDoArquivoDeAudio(char * nomeArquivo, int taxaAmostragem, int tempoComandoVez_ms, int tamCabe

unsigned char dado1, dado2;
unsigned int dadoInt;
double dadoDouble;
audio audiostream;
    int tamanhoStream;

//aloca a memória necessária
tamanhoStream = (taxaAmostragem*tempoComandoVez_ms/1000)+tamCabecalhoWave;
audiostream.stream = calloc(tamanhoStream,sizeof(double));

//leitura do arquivo de áudio
FILE * arqAudio;
if ((arqAudio = fopen(nomeArquivo,"rb"))==NULL) {
printf("### Erro ao abrir arquivo WAV: %s ###\n",nomeArquivo);
getchar();
exit(1);
}
audiostream.nsamples=0;
while (fread (&dado1, sizeof(char), 1, arqAudio) && audiostream.nsamples<tamanhoStream) {
fread (&dado2, sizeof(char), 1, arqAudio);
dadoInt = 0x0 & 0x0;
dadoInt = dadoInt | dado1;
dadoInt = dadoInt << 8;
dadoInt = dadoInt | dado2;

//faz o cast para o double
dadoDouble = 0x0 & 0x0;
dadoDouble = (double)dadoInt;

//coloca o dado no conteúdo que será retornado
```

```
audiostream.stream[audiostream.nsamples] = dadoDouble;
audiostream.nsamples++;
}

fclose(arqAudio);
   return audiostream;
}

double * retiraCabecalhoArquivoWav(double *streamVoz, int nsamples, int tamanhoCabecalhoWav){
int i;
double * streamVozsc;
streamVozsc = calloc(nsamples, sizeof(double));
for (i=tamanhoCabecalhoWav; i < nsamples; i++) {
streamVozsc[i] = streamVoz[i] ;
}
        return streamVozsc;
}

double * normalizaAudio(double * streamVoz, int numeroDeAmostrasNoComando){

    double maiorAmostra = 0.0;
    int i;

    //for para encontrar a maior amostra existente na stream de voz
    for(i=0; i<numeroDeAmostrasNoComando; i++){
        if(streamVoz[i] > maiorAmostra) maiorAmostra = streamVoz[i];
    }

    //for para normalizar todas as amostras em relação a maior amostra obtida
    if(maiorAmostra > 0.0){
        for(i=0; i<numeroDeAmostrasNoComando; i++){
            streamVoz[i] = streamVoz[i] / maiorAmostra;
        }
    }


    return streamVoz;
}
```

Módulo secundário 2 - extratorMelCepstral.c:

```
#include <stdio.h>
#define NUMERO_DE_FILTROS_DIGITAIS 20


double calculaSK(int k, double **streamVoz){
    double resultado = 1.0;
    return resultado;
}

/**
 * Função: extrairCoefientesMelCepstrais (char *streamVoz, double *mfcc)
 *
 * Parâmetros:
 *             1. char *streamVoz
 *                    representa a stream de bits do sinal sonoro
 *             2. char *mfcc
 *                    representa o ponteiro para o local de memória que os coeficientes mel-cepstrais serão guardados
 *
 * Regra de negócio (lógica):
 *             essa função calcula o os coeficientes mel-cepstrais de acordo com a seguinte fórmula matemática:
 *                               |K                                                  |
 *             c(n) = somatorio  |    log|S(k)|  * cos(  (n*Pi/K) * (k-0.5)  )  |
 *                               |k=1                                                |
 */
int extrairCoefientesMelCepstraisDeUmBloco (double *bloco, double **mfcc, int numeroCoeficientesMelCepstrais, int numer

    //aplica a transformada com janelamento de hamming
    double * XFFT = (double*)winFFT(bloco, numeroAmostrasPorBloco);

    //aplica os limites para calculo dos filtros
```

```
    double * f = (double*)computeBoundaries(numeroCoeficientesMelCepstrais, limiteFreqInferior, limiteFreqSuperior, nur

    //aplica os filtros
    double ** Hm = (double**)HmBank(f, numeroAmostrasPorBloco, numeroCoeficientesMelCepstrais);

    //começa cálculo dos coeficientes mel cepstrais
    double * S = (double*)MelWrapping (XFFT, Hm, numeroCoeficientesMelCepstrais, numeroAmostrasPorBloco);

    //finaliza cálculo dos coeficientes mel cepstrais (transformada inversa)
    *mfcc = (double*)MelCepstralCoefficients (S, numeroCoeficientesMelCepstrais, numeroCoeficientesMelCepstrais);

    return 0;
}
```

Módulo secundário 3 - MelCepstralCoefficients.c:

```
/*-------------------------------------------------------------------*
 *    Toolbox para claculo de parametros mel-cepstrais
 *
 *    Autor:  Pedro de Azevedo Berger
 *    Data:   03 de setembro de 2008
 *
 * Variáveis :
 *
 * nF - Número de filtros
 * fLow - Limite inferior de frequencia
 * fHigh - Limite superior de frequencia
 * nFFT - Numero de pontos da FFT
 * Fs - Frequencia de amostragem em Hz
 * Hm - matriz de filtros mel-cepstrais
 * XFFT - FFT janelada do audio
 * S - Espectro Mel
 * c - coeficientes mfcc
 *-------------------------------------------------------------------*/


#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "srfft.h"

#define PI 3.1415926536


double BMel( double f){
    double b;
    b = 1125 * log(1 + (f/700));

    return b;
}

double invBMel( double b) {
    double f;
    f = 700 * (exp(b/1125)-1);

    return f;
}

double* computeBoundaries(int nF, double fLow, double fHigh, int nFFT, double Fs) {
    double *f;
    int m=0;
    nFFT=nFFT/2;
    f = calloc(nF+1,sizeof(double));

    for (m=0; m < nF+1; m++) {
     f[m]= (invBMel(BMel(fLow) + (m *((BMel(fHigh)-BMel(fLow))/(nF + 1)))))*(nFFT/Fs);
    }

    return f;
}
```

```
double ** HmBank(double * f, int nFFT, int nF) {

    double **Hm;
    int m,k;

    Hm = (double **) calloc(nF,sizeof(double));
    nFFT=nFFT/2;
    for(k=0;k<nF;k++){
     Hm[k] = (double *) calloc(nFFT, sizeof(double));
    }
    for  (m=1; m < nF+1; m++) {
     for (k=0; k < nFFT; k++) {
     if (k < f[m-1]) {
     Hm[m-1][k] = 0;
     }
     else if (k <= f[m]) {
     Hm[m-1][k] = (2*(k - f[m-1]))/((f[m+1] - f[m-1]) * (f[m]-f[m-1]));
     }
     else {
     Hm[m-1][k] = (2*(f[m+1]-k))/((f[m+1]- f[m-1])*(f[m+1]-f[m]));
     }
     }
    }

    return Hm;
}


double * MelWrapping (double * XFFT, double ** Hm, int nF, int nFFT) {

    double * S;
    int m,k;
    nFFT=nFFT/2;
    //printf("\t%d\n\t%d\n\n",nF,sizeof(double));
    S = (double*)calloc(nF,sizeof(double));

    for ( m=0; m < nF; m++ ) {
     for ( k=0 ; k< nFFT; k++ ) {
     S[m] = S[m]+ (XFFT[k]*Hm[m][k])*(XFFT[k]*Hm[m][k]);
     }
     S[m] = log(S[m]);
    }

    return S;
}

double * MelCepstralCoefficients (double *S, int nF, int nMel) {

    int k,m;
    double *c;

    c = calloc (nMel, sizeof(double));

    for ( k=0 ; k < nMel ; k++ ) {
     for ( m=0 ; m < nF ; m++ ) {
     c[k]=c[k]+(S[m]*cos(PI*k*0.5*(m-1)/nMel));
     }
    }

    return c;
}

double win_hanning(int n, int nFFT){

    double w;

    w = 0.5 - 0.5*cos(n*2.0*PI/(nFFT-1));

    return w;
}
```

```
double * winFFT(double *stream, int nFFT) {

    int i;
    float *xr, *xi;
    double *XFFT;

    xr = calloc(nFFT, sizeof(float));
    xi = calloc(nFFT, sizeof(float));
    XFFT =  calloc(nFFT, sizeof(double));

    for ( i=0 ; i < nFFT ; i++) {
     xr[i]=stream[i]*win_hanning(i,nFFT);
    }

    srfft(xr,xi,log2(nFFT));

    for ( i=0 ; i < nFFT ; i++) {

     XFFT[i] = (double) sqrt(xr[i]*xr[i] + xi[i]*xi[i]);

    }

    return XFFT;
}
```

Módulo secundário 4 - srfft.c:

```
/*---------------------------------------------------------------------*
 *    SRFFT.C  -  Split-Radix Fast Fourier Transform                   *
 *                                                                     *
 *    This is a decimation-in-frequency version, using the steps       *
 *    of the algorithm as described in Section 2.5.                    *
 *                                                                     *
 *    Author:  Henrique S. Malvar.                                     *
 *    Date:    October 8, 1991.                                        *
 *                                                                     *
 *    Usage:   srfft(xr, xi, logm);   --  for direct DFT               *
 *             srifft(xr, xi, logm);  --  for inverse DFT              *
 *                                                                     *
 *    Arguments:  xr (float)  - input and output vector, length M,     *
 *                              real part.                             *
 *                xi (float)  - imaginary part.                        *
 *                logm (int)  - log (base 2) of vector length M,       *
 *                              e.g., for M = 256 -> logm = 8.         *
 *---------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
//#include <alloc.h>
#include <math.h>

#define  MAXLOGM     11    /* max FFT length = 2^MAXLOGM */
#define  TWOPI       6.28318530717958647692
#define  SQHALF      0.70710678118654524401

/*---------------------------------------------------------------------*
 *    Error exit for program abortion.                                 *
 *---------------------------------------------------------------------*/

static   void  error_exit(void)
{
   exit(1);
}

/*---------------------------------------------------------------------*
 *    Data unshuffling according to bit-reversed indexing.             *
 *                                                                     *
 *    Bit reversal is done using Evan's algorithm (Ref: D. M. W.       *
 *    Evans, "An improved digit-reversal permutation algorithm ...",   *
 *    IEEE Trans. ASSP, Aug. 1987, pp. 1120-1125).                     *
```

```
 *--------------------------------------------------------------------*/

static   int   brseed[256];   /* Evans' seed table */
static   int   brsflg;        /* flag for table building */

void BR_permute(float *x, int logm)
{
   int      i, j, imax, lg2, n;
   int      off, fj, gno, *brp;
   float    tmp, *xp, *xq;

   lg2 = logm >> 1;
   n = 1 << lg2;
   if (logm & 1) lg2++;

   /* Create seed table if not yet built */
   if (brsflg != logm) {
      brsflg = logm;
      brseed[0] = 0;
      brseed[1] = 1;
      for (j = 2; j <= lg2; j++) {
         imax = 1 << (j - 1);
         for (i = 0; i < imax; i++) {
            brseed[i] <<= 1;
            brseed[i + imax] = brseed[i] + 1;
         }
      }
   }

   /* Unshuffling loop */
   for (off = 1; off < n; off++) {
      fj = n * brseed[off]; i = off; j = fj;
      tmp = x[i]; x[i] = x[j]; x[j] = tmp;
      xp = &x[i];
      brp = &brseed[1];
      for (gno = 1; gno < brseed[off]; gno++) {
         xp += n;
         j = fj + *brp++;
         xq = x + j;
         tmp = *xp; *xp = *xq; *xq = tmp;
      }
   }
}

/*--------------------------------------------------------------------*
 *    Recursive part of the SRFFT algorithm.                          *
 *--------------------------------------------------------------------*/

void srrec(float *xr, float *xi, int logm)
{
   static   int      m, m2, m4, m8, nel, n;
   static   float    *xr1, *xr2, *xi1, *xi2;
   static   float    *cn, *spcn, *smcn, *c3n, *spc3n, *smc3n;
   static   float    tmp1, tmp2, ang, c, s;
   static   float    *tab[MAXLOGM];

   /* Check range of logm */
   if ((logm < 0) || (logm > MAXLOGM)) {
      printf("Error : SRFFT : logm = %d is out of bounds [%d, %d]\n",
         logm, 0, MAXLOGM);
      error_exit();
   }

   /* Compute trivial cases */
   if (logm < 3) {
      if (logm == 2) {  /* length m = 4 */
         xr2  = xr + 2;
         xi2  = xi + 2;
         tmp1 = *xr + *xr2;
         *xr2 = *xr - *xr2;
```

```
      *xr  = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi  = tmp1;
      xr1  = xr + 1;
      xi1  = xi + 1;
      xr2++;
      xi2++;
      tmp1 = *xr1 + *xr2;
      *xr2 = *xr1 - *xr2;
      *xr1 = tmp1;
      tmp1 = *xi1 + *xi2;
      *xi2 = *xi1 - *xi2;
      *xi1 = tmp1;
      xr2  = xr + 1;
      xi2  = xi + 1;
      tmp1 = *xr + *xr2;
      *xr2 = *xr - *xr2;
      *xr  = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi  = tmp1;
      xr1  = xr + 2;
      xi1  = xi + 2;
      xr2  = xr + 3;
      xi2  = xi + 3;
      tmp1 = *xr1 + *xi2;
      tmp2 = *xi1 + *xr2;
      *xi1 = *xi1 - *xr2;
      *xr2 = *xr1 - *xi2;
      *xr1 = tmp1;
      *xi2 = tmp2;
      return;
   }
   else if (logm == 1) {   /* length m = 2 */
      xr2  = xr + 1;
      xi2  = xi + 1;
      tmp1 = *xr + *xr2;
      *xr2 = *xr - *xr2;
      *xr  = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi  = tmp1;
      return;
   }
   else if (logm == 0) return;   /* length m = 1 */
}

/* Compute a few constants */
m = 1 << logm; m2 = m / 2; m4 = m2 / 2; m8 = m4 /2;

/* Build tables of butterfly coefficients, if necessary */
if ((logm >= 4) && (tab[logm-4] == NULL)) {

   /* Allocate memory for tables */
   nel = m4 - 2;
   if ((tab[logm-4] = (float *) calloc(6 * nel, sizeof(float))) == NULL) {
      error_exit();
   }

   /* Initialize pointers */
   cn  = tab[logm-4]; spcn  = cn + nel;  smcn  = spcn + nel;
   c3n = smcn + nel;  spc3n = c3n + nel; smc3n = spc3n + nel;

   /* Compute tables */
   for (n = 1; n < m4; n++) {
      if (n == m8) continue;
      ang = n * TWOPI / m;
      c = cos(ang); s = sin(ang);
      *cn++ = c; *spcn++ = - (s + c); *smcn++ = s - c;
```

```
      ang = 3 * n * TWOPI / m;
      c = cos(ang); s = sin(ang);
      *c3n++ = c; *spc3n++ = - (s + c); *smc3n++ = s - c;
   }
}

/* Step 1 */
xr1 = xr; xr2 = xr1 + m2;
xi1 = xi; xi2 = xi1 + m2;
for (n = 0; n < m2; n++) {
   tmp1 = *xr1 + *xr2;
   *xr2 = *xr1 - *xr2;
   *xr1 = tmp1;
   tmp2 = *xi1 + *xi2;
   *xi2 = *xi1 - *xi2;
   *xi1 = tmp2;
   xr1++; xr2++; xi1++; xi2++;
}

/* Step 2 */
xr1 = xr + m2; xr2 = xr1 + m4;
xi1 = xi + m2; xi2 = xi1 + m4;
for (n = 0; n < m4; n++) {
   tmp1 = *xr1 + *xi2;
   tmp2 = *xi1 + *xr2;
   *xi1 = *xi1 - *xr2;
   *xr2 = *xr1 - *xi2;
   *xr1 = tmp1;
   *xi2 = tmp2;
   xr1++; xr2++; xi1++; xi2++;
}

/* Steps 3 & 4 */
xr1 = xr + m2; xr2 = xr1 + m4;
xi1 = xi + m2; xi2 = xi1 + m4;
if (logm >= 4) {
   nel = m4 - 2;
   cn  = tab[logm-4]; spcn  = cn + nel;  smcn  = spcn + nel;
   c3n = smcn + nel;  spc3n = c3n + nel; smc3n = spc3n + nel;
}
xr1++; xr2++; xi1++; xi2++;
for (n = 1; n < m4; n++) {
   if (n == m8) {
      tmp1 =  SQHALF * (*xr1 + *xi1);
      *xi1 =  SQHALF * (*xi1 - *xr1);
      *xr1 =  tmp1;
      tmp2 =  SQHALF * (*xi2 - *xr2);
      *xi2 = -SQHALF * (*xr2 + *xi2);
      *xr2 =  tmp2;
   } else {
      tmp2 = *cn++ * (*xr1 + *xi1);
      tmp1 = *spcn++ * *xr1 + tmp2;
      *xr1 = *smcn++ * *xi1 + tmp2;
      *xi1 = tmp1;
      tmp2 = *c3n++ * (*xr2 + *xi2);
      tmp1 = *spc3n++ * *xr2 + tmp2;
      *xr2 = *smc3n++ * *xi2 + tmp2;
      *xi2 = tmp1;
   }
   xr1++; xr2++; xi1++; xi2++;
}

/* Call ssrec again with half DFT length */
srrec(xr, xi, logm-1);

/* Call ssrec again twice with one quarter DFT length.
   Constants have to be recomputed, because they are static! */
m = 1 << logm; m2 = m / 2;
srrec(xr + m2, xi + m2, logm-2);
m = 1 << logm; m4 = 3 * (m / 4);
```

```
   srrec(xr + m4, xi + m4, logm-2);
}

/*---------------------------------------------------------------------*
 *    Direct transform                                                 *
 *---------------------------------------------------------------------*/

void  srfft(float *xr, float *xi, int logm)
{
   /* Call recursive routine */
   srrec(xr, xi, logm);

   /* Output array unshuffling using bit-reversed indices */
   if (logm > 1) {
      BR_permute(xr, logm);
      BR_permute(xi, logm);
   }
}

/*---------------------------------------------------------------------*
 *    Inverse transform.  Uses Duhamel's trick (Ref: P. Duhamel        *
 *    et. al., "On computing the inverse DFT", IEEE Trans. ASSP,       *
 *    Feb. 1988, pp. 285-286).                                         *
 *---------------------------------------------------------------------*/

void  srifft(float *xr, float *xi, int logm)
{
   int      i, m;
   float    fac, *xrp, *xip;

   /* Call direct FFT, swapping real & imaginary addresses */
   srfft(xi, xr, logm);

   /* Normalization */
   m = 1 << logm;
   fac = 1.0 / m;
   xrp = xr; xip = xi;
   for (i = 0; i < m; i++) {
      *xrp++ *= fac;
      *xip++ *= fac;
   }
}
```

Módulo secundário 5 - algoritmosHMM.c:

```
#include <math.h>

#include "hmm.h"
#include "emissor.h"
#include "matrizUtil.h"

int * posicaoCbkDaObservacao;

int * viterbi(HMM *estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDaObservacao){
    int i,j,t,k,numeroEstados,segundoIndiceB,argMax;
    int * sequenciaEstados;
    double maximo, maiorV, maiorB;
    double **V, **B;

    numeroEstados = estruturaHMM->N;

    //alocação de memoria para a sequencia de estados
    sequenciaEstados = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //alocação dinâmica para V e B:
    V = (double**)calloc(numeroDeObservacoes,sizeof(double));
    for(i=0; i<numeroDeObservacoes; i++) V[i] = (double*)calloc(numeroEstados,sizeof(double));
    B = (double**)calloc(numeroDeObservacoes,sizeof(double));
    for(i=0; i<numeroDeObservacoes; i++) B[i] = (double*)calloc(numeroEstados,sizeof(double));

    //Inicialização de V e B
```

```
    for(i=0; i<numeroEstados; i++){
        segundoIndiceB = posicaoCbkDaObservacao[0];
        V[0][i] = estruturaHMM->pi[i] * estruturaHMM->b[i][segundoIndiceB];
    }
    for(i=0; i<numeroEstados; i++) B[0][i] = 0;

    //Indução
    for(t=1; t<numeroDeObservacoes; t++){
        for(j=0; j<numeroEstados; j++){
            V[t][j] = 0.0;
            argMax = 0;
            for(i=0; i<numeroEstados; i++){
                maximo = V[t-1][i] * estruturaHMM->a[i][j];
                if(maximo > V[t][j]) {
                    V[t][j] = maximo;
                    argMax = i;
                }
            }
            segundoIndiceB = posicaoCbkDaObservacao[t];
            V[t][j] *= estruturaHMM->b[j][segundoIndiceB];
            B[t][j] = argMax;
        }
    }

    //Finalização
    maiorB = 0.0;
    sequenciaEstados[numeroDeObservacoes-1] = 0;
    for (i=0; i<numeroEstados; i++) {
        if(B[numeroDeObservacoes-1][i] > maiorB){
            maiorB = B[numeroDeObservacoes-1][i];
            sequenciaEstados[numeroDeObservacoes-1] = i;
        }
    }

    //Back-tracking
    for(t=numeroDeObservacoes-2; t>=0; t--){
        sequenciaEstados[t] = B[t+1][sequenciaEstados[t+1]];
    }

    return sequenciaEstados;
}

double calculaProbabilidade_X_S_dado_FI(HMM * hmm, double **observacoes, int numeroDeObservacoes, int tamanhoDaObservac
    int t;
    int * sequenciaEstados;
    double resposta;

    //roda o viterbi
    sequenciaEstados = viterbi(hmm, observacoes, numeroDeObservacoes, tamanhoDaObservacao);

    //inicialização da resposta
    resposta = hmm->pi[sequenciaEstados[0]] * hmm->b[sequenciaEstados[0]][posicaoCbkDaObservacao[0]];

    //continuação do cálculo da resposta
    for(t=1; t<numeroDeObservacoes; t++){
       resposta *= hmm->a[sequenciaEstados[t-1]][sequenciaEstados[t]] * hmm->b[sequenciaEstados[t]][posicaoCbkDaObserva
    }

    return resposta;
}

double calculaDeltaProbConv(double valor1, double valor2){
    double resposta, maior, menor;

    //varifica qual é o maior e qual o menor
    if(valor1>=valor2) {
        maior = valor1;
        menor = valor2;
    }
    else{
```

```
        maior = valor2;
        menor = valor1;
    }

    //calcula a resposta
    resposta = menor/maior;


    return (1-resposta);
}

double obtemFatorContraUnderFlow(double num){
    int expoente,i;
    double fator;

    fator = 1.0;
    expoente = obtemExpoente(num);
    if(expoente<0) expoente *= (-1);
    else return fator;
    for(i=0; i<expoente; i++) fator *= 10.0;

    return fator;
}

double obtemFatorContraUnderFlow2(double num){
    int expoente,i;
    double fator;

    fator = 1.0;
    expoente = obtemExpoente(num);
    if(expoente<0) expoente *= (-1);
    else return fator;

    for(i=0; i<expoente; i++) fator *= 10.0;

    //printf("fator: %lf = %e\n",fator,fator);
    //printf("expoente: %d\n",expoente);
    //getchar();

    return fator;
}

int ocorreUnderFlow(double num1, double num2){
    int exp1, exp2, soma;

    exp1 = obtemExpoente(num1);
    exp2 = obtemExpoente(num2);
    soma = exp1+exp2;
    if(soma<0){
        soma *= (-1);
        if(soma >= 325) return 1;
        else return 0;
    }
    else return 0;

}

double ** normalizaLinha(double **matriz, int numLinhas, int numColunas){
    int t,i;
    double ** matrizNormalizada;
    double * fatoresDeNormalizacao;
    double soma;

    //alocação dinâmica para matrizNormalizada
    matrizNormalizada = (double**)calloc(numLinhas,sizeof(double));
    for(i=0; i<numLinhas; i++) matrizNormalizada[i] = (double*)calloc(numColunas,sizeof(double));

    //alocação dinâmica para fatoresDeNormalizacao
    fatoresDeNormalizacao = (double*)calloc(numLinhas,sizeof(double));
```

```
    //calcula todos os fatores de normalização (um para cada t)
    for(t=0; t<numLinhas; t++){
        soma = 0.0;
        for(i=0; i<numColunas; i++){
            soma += matriz[t][i];
        }
        fatoresDeNormalizacao[t] = soma;
    }

    //aplica os fatores de normalização
    for(t=0; t<numLinhas; t++){
        for(i=0; i<numColunas; i++){
            if(fatoresDeNormalizacao[t]>0.0) matrizNormalizada[t][i] =  matriz[t][i] / fatoresDeNormalizacao[t];
        }
    }

    return matrizNormalizada;

}

double obtemFatorNormalizacaoUltimoAlfa(double * ultimoAlfa, int numEstados){
    int i;
    double soma;
    soma = 0.0;
    for(i=0; i<numEstados; i++){
        soma += ultimoAlfa[i];
    }
    return soma;
}

double aplicaProdutoEscalarDaDiferenca(double * observacao, double * vetorCodeBook, int tamanhoVetor) {
    int i;
    double resultado;
    double * diferenca;

    diferenca = vetorA_menos_vetorB(tamanhoVetor, observacao, vetorCodeBook);
    resultado = 0.0;

    for(i=0; i<tamanhoVetor; i++) {
        resultado += diferenca[i] * diferenca[i];
    }

    if(resultado>0.0) return resultado;
    else return -resultado;
}

int definePosicaoSimbolo(double * observacao, double ** codebook, int tamanhoAlfabeto, int tamanhoDaObservacao){
    int i,posicao;
    double produtoEscalar, produtoEscalarAux ;

    produtoEscalar = 9999999999999999999999.0;
    posicao = 0;

    for(i=0; i<tamanhoAlfabeto; i++){
        produtoEscalarAux = aplicaProdutoEscalarDaDiferenca(observacao, codebook[i], tamanhoDaObservacao);
        if(produtoEscalarAux < produtoEscalar) {
            produtoEscalar = produtoEscalarAux;
            posicao = i;
        }
    }
    return posicao;
}


double ** calcularAlfa(int contador, HMM* estruturaHMM, double ** observacoes ,int numeroDeObservacoes, int tamanhoDaO
    double ** alfa, **alfaNormalizado, alfaAux, fatorContraUnderFlow;
    int x,y,i,j,t,w,numeroEstados, numeroSaidas,segundoIndiceB,expSomat,expB;
    double somatorio;

    //variáveis para melhor compreensão do código
```

```
numeroEstados = estruturaHMM->N;
numeroSaidas = numeroDeObservacoes;

//alocação dinâmica para a matriz de tamanho numeroSaidas X numeroEstados
alfa = (double**)calloc(numeroSaidas,sizeof(double));
for(i=0; i<numeroSaidas; i++) alfa[i] = (double*)calloc(numeroEstados,sizeof(double));

//passo 1: Inicialização
for(i=0; i<numeroEstados; i++) {
    segundoIndiceB = posicaoCbkDaObservacao[0];
    alfa[0][i] = estruturaHMM->pi[i] * estruturaHMM->b[i][segundoIndiceB];

    /*if(contador==2){
    printf("pi[%d] = %lf = %e\n",i,estruturaHMM->pi[i],estruturaHMM->pi[i]);
    printf("b[%d][%d] = %lf = %e\n",i,0,estruturaHMM->b[i][0],estruturaHMM->b[i][0]);
    printf("alfa[%d][%d] = %lf = %e\n\n",0,i,alfa[0][i],alfa[0][i]);
    getchar();
    }*/
}

//passo 2: Indução
for(t=1; t<numeroSaidas; t++){

    //normaliza os alfas anteriores
    for(w=0; w<numeroEstados; w++) {
        alfa[t-1][w] *= obtemFatorContraUnderFlow(alfa[t-1][w]);
    }

    //passo de indução propriamente dito
    for(j=0; j<numeroEstados; j++){
        somatorio = 0.0;
        for(i=0; i<numeroEstados; i++){
            //alfaAux = alfa[t-1][i];
            somatorio = somatorio + (alfa[t-1][i] * estruturaHMM->a[i][j]);
            /*if(t==42){
                printf("somatorio: %lf = %e\n",somatorio,somatorio);
                printf("alfa[%d][%d]: %lf = %e\n",t-1,i,alfa[t-1][i],alfa[t-1][i]);
                printf("estruturaHMM->a[%d][%d]: %lf = %e\n",i,j,estruturaHMM->a[i][j],estruturaHMM->a[i][j]);
                getchar();
            }*/

        }

        /*if(t==42){
            for(x=0;x<numeroEstados;x++) {
                for(y=0;y<numeroEstados;y++) {
                    printf("a[%d][%d] = %lf = %e\n",x,y,estruturaHMM->a[x][y], estruturaHMM->a[x][y]);
                }
            }
            printf("\n\n");
            for(x=0;x<numeroEstados;x++) printf("alfa[%d][%d] = %lf = %e\n",t-2,x,alfa[t-2][x], alfa[t-2][x]);
            printf("\n");
            for(x=0;x<numeroEstados;x++) printf("alfa[%d][%d] = %lf = %e\n",t-1,x,alfa[t-1][x], alfa[t-1][x]);
            printf("\n");
            for(x=0;x<numeroEstados;x++) printf("alfa[%d][%d] = %lf = %e\n",t-0,x,alfa[t-0][x], alfa[t-0][x]);
            printf("\n");
            getchar();
        }*/


        segundoIndiceB = posicaoCbkDaObservacao[t];
        alfa[t][j] = somatorio * estruturaHMM->b[j][segundoIndiceB];
        //printf("alfa[%d][%d] = %lf = %e\n",t,j,alfa[t][j],alfa[t][j]);
        /*if(t==43){
                printf("somatorio: %lf = %e\n",somatorio,somatorio);
                printf("alfa[%d][%d]: %lf = %e\n",t,j,alfa[t][j],alfa[t][j]);
                printf("b[%d][%d]: %lf = %e\n",j,segundoIndiceB,estruturaHMM->b[j][segundoIndiceB],estruturaHMM->b[
                getchar();
        }*/
    }
```

```
    }

    //passo 3: Finalização
double prob = 0.0;
    double somaLinhaAlfa;
for (t=0; t<numeroSaidas; t++){
        somaLinhaAlfa = 0.0;
    for (i=0; i<numeroEstados; i++){
            somaLinhaAlfa += alfa[t][i];
        }
prob += log(somaLinhaAlfa);
    }
    if(prob<0.0) prob = -prob;
    *probabilidadeLog = prob;


    *fatorNormalizacaoUltimoAlfa = obtemFatorNormalizacaoUltimoAlfa(alfa[numeroDeObservacoes-1],estruturaHMM->N);
    alfaNormalizado = (double**)normalizaLinha(alfa, numeroDeObservacoes, numeroEstados);
    free(alfa);

    return alfaNormalizado;
}

double ** calcularAlfaSemNormalizacao(int contador, HMM* estruturaHMM, double ** observacoes ,int numeroDeObservacoes,
    double ** alfa, alfaAux;
    int i,j,t,w,numeroEstados, numeroSaidas,segundoIndiceB,expSomat,expB;
    double somatorio;

    //variáveis para melhor compreensão do código
    numeroEstados = estruturaHMM->N;
    numeroSaidas = numeroDeObservacoes;

    //alocação dinâmica para a matriz de tamanho numeroSaidas X numeroEstados
    alfa = (double**)calloc(numeroSaidas,sizeof(double));
    for(i=0; i<numeroSaidas; i++) alfa[i] = (double*)calloc(numeroEstados,sizeof(double));

    //passo 1: Inicialização
    for(i=0; i<numeroEstados; i++) {
        segundoIndiceB = posicaoCbkDaObservacao[0];
        alfa[0][i] = estruturaHMM->pi[i] * estruturaHMM->b[i][segundoIndiceB];
        alfa[0][i] *= 1e300;
    }

    //passo 2: Indução
    for(t=1; t<numeroSaidas; t++){

        //passo de indução propriamente dito
        for(j=0; j<numeroEstados; j++){
            somatorio = 0.0;
            for(i=0; i<numeroEstados; i++){
                somatorio = somatorio + (alfa[t-1][i] * estruturaHMM->a[i][j]);
            }
            segundoIndiceB = posicaoCbkDaObservacao[t];
            alfa[t][j] = somatorio * estruturaHMM->b[j][segundoIndiceB];
        }
    }

    return alfa;
}

double ** calcularBeta(int contador, HMM* estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDaOb
    double ** beta, **betaNormalizado, fatorContraUnderFlow;

    int w,i,j,t,numeroEstados, numeroSaidas,segundoIndiceB;

    //variáveis para melhor compreensão do código
    numeroEstados = estruturaHMM->N;
    numeroSaidas = numeroDeObservacoes;
```

```
    //alocação dinâmica para a matriz de tamanho numeroSaidas X numeroEstados
    beta = (double**)calloc(numeroSaidas,sizeof(double));
    for(i=0; i<numeroSaidas; i++) beta[i] = (double*)calloc(numeroEstados,sizeof(double));

    //passo 1: Inicialização
    for(i=0; i<numeroEstados; i++) beta[numeroSaidas-1][i] = 1.0;///numeroEstados;

    //passo 2: Indução
    for(t=numeroSaidas-2; t>=0; t--){

        //normaliza os alfas anteriores
        for(w=0; w<numeroEstados; w++) beta[t+1][w] *= obtemFatorContraUnderFlow(beta[t+1][w]);

        for(i=0; i<numeroEstados; i++){
            beta[t][i] = 0.0;
            for(j=0; j<numeroEstados; j++){
                //segundoIndiceB = definePosicaoSimbolo(observacoes[t+1],estruturaHMM->codebook,estruturaHMM->M,tamanho
                segundoIndiceB = posicaoCbkDaObservacao[t+1];
                beta[t][i] = beta[t][i] + (estruturaHMM->a[i][j] * estruturaHMM->b[j][segundoIndiceB] * beta[t+1][j]);
            }
        }
    }

    betaNormalizado = (double**)normalizaLinha(beta, numeroDeObservacoes, numeroEstados);
    free(beta);
    return betaNormalizado;
}

double *** calcularGama(int contador, HMM* estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDa(

    double *** gama;
double **numerador,denominador;
    int ii,jj,i,j,k,t,numeroEstados, numeroSaidas,segundoIndiceB;



    //variáveis para melhor compreensão do código
    numeroEstados = estruturaHMM->N;
    numeroSaidas = numeroDeObservacoes;

    //alocação dinâmica para a matriz de tamanho numeroSaidas X numeroEstados
    gama = (double***)calloc(numeroSaidas,sizeof(double));
    for(i=0; i<numeroSaidas; i++) gama[i] = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroSaidas; i++) {
        for(j=0; j<numeroEstados; j++) gama[i][j] = (double*)calloc(numeroEstados,sizeof(double));
    }

    //alocação dos numeradores de gama
    numerador = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) numerador[i] = (double*)calloc(numeroEstados,sizeof(double));



    //calcula gama a partir da definição
for (t=0; t<numeroSaidas-1; t++) {
        denominador = 0.0;
        for (i=0; i<numeroEstados; i++) {
            for (j=0; j<numeroEstados; j++) {
                segundoIndiceB = posicaoCbkDaObservacao[t+1];
                gama[t][i][j] = alfa[t][i] * estruturaHMM->a[i][j] * estruturaHMM->b[j][segundoIndiceB] * beta[t+1][j];
                denominador += gama[t][i][j];

                /*for(ii=0; ii<numeroEstados; ii++) {
                    for(jj=0; jj<numeroEstados; jj++) {
                        denominador += alfa[t][ii] * estruturaHMM->a[ii][jj] * estruturaHMM->b[jj][segundoIndiceB] * be
                    }
                }

                if(denominador!=0.0) gama[t][i][j] = numerador / denominador;
                else gama[t][i][j] = 0.0;*/
```

```
            }
        }

        //for para calcular o gama considerando o denominador
        for (i=0; i<numeroEstados; i++) {
            for (j=0; j<numeroEstados; j++) {
                if(denominador!=0.0) gama[t][i][j] /= denominador;
                else gama[t][i][j] = 0.0;
            }
        }
    }

    //for para calcular os últimos elementos de gama
    denominador = 0.0;
    for (i=0; i<numeroEstados; i++) {
        for (j=0; j<numeroEstados; j++) {
            gama[t][i][j] = alfa[t][i];
            denominador += gama[t][i][j];




        }
    }
    //for para calcular o gama considerando o denominador
    for (i=0; i<numeroEstados; i++) {
        for (j=0; j<numeroEstados; j++) {
            if(denominador!=0.0) gama[t][i][j] /= denominador;
            else gama[t][i][j] = 0.0;
        }
    }


    return gama;
}

int forwardBackward(HMM * estruturaHMM, double **mfcc, int numeroDeObservacoes, int tamanhoDaObservacao, double converg

    int i,j,t,k,numeroSaidas,numeroEstados,contador;
    double **alfa, **beta, **gamaRabiner;
    double ***gama;
    double probLogForward,probLogForwardAntes,delta;
    double probConvergAntes,probConvergAtual,deltaProbConverg;

    numeroSaidas = estruturaHMM->M;
    numeroEstados = estruturaHMM->N;

    double **numeradorA;
    double **denominadorA;
    double **numeradorB;
    double **denominadorB;
    double *valorPi;

    //alocação de memoria para a estrutura de dados posicaoCbkDaObservacao (relação entre a observação e a sua posição
    //posicaoCbkDaObservacao = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //alocação de memória para o valorPi
    valorPi = (double*)calloc(numeroEstados,sizeof(double));

    //alocação de memória para o numerador A
    numeradorA = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) numeradorA[i] = (double*)calloc(numeroEstados,sizeof(double));

    //alocação de memória para o denominador A
    denominadorA = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) denominadorA[i] = (double*)calloc(numeroEstados,sizeof(double));

    //alocação de memória para o numerador B
    numeradorB = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) numeradorB[i] = (double*)calloc(numeroSaidas,sizeof(double));
```

```
//alocação de memória para o denominador B
denominadorB = (double**)calloc(numeroEstados,sizeof(double));
for(i=0; i<numeroEstados; i++) denominadorB[i] = (double*)calloc(numeroSaidas,sizeof(double));

//alocação de memória para o gamaRabiner
gamaRabiner = (double**)calloc(numeroDeObservacoes,sizeof(double));
for(i=0; i<numeroDeObservacoes; i++) gamaRabiner[i] = (double*)calloc(numeroEstados,sizeof(double));

//preenchimento da relação entre a observação e a sua posição no codebook (posicaoCbkDaObservacao)
//for(t=0; t<numeroDeObservacoes; t++) posicaoCbkDaObservacao[t] = definePosicaoSimbolo(mfcc[t], estruturaHMM->code
posicaoCbkDaObservacao = estruturaHMM->posicaoCbkDaObservacao;

contador = 0;
do{
    contador++;

    //#####################
    //## passo-E - prepara a iteração
    //#####################

    //cálculo do alfa, beta e gama
    alfa = (double**)calcularAlfa(contador, estruturaHMM,mfcc,numeroDeObservacoes,tamanhoDaObservacao,&probLogForwa
    beta = (double**)calcularBeta(contador, estruturaHMM,mfcc,numeroDeObservacoes,tamanhoDaObservacao);
    gama = (double***)calcularGama(contador,estruturaHMM,mfcc,numeroDeObservacoes,tamanhoDaObservacao,alfa,beta);
    probConvergAntes = calculaProbabilidade_X_S_dado_FI(estruturaHMM,mfcc,numeroDeObservacoes,tamanhoDaObservacao);

    //cálculo do gamaRabiner
    for(t=0; t<numeroDeObservacoes; t++){
        for(i=0; i<numeroEstados; i++){
            gamaRabiner[t][i] = 0.0;
            for(j=0; j<numeroEstados; j++) gamaRabiner[t][i] += gama[t][i][j];
        }
    }

    //getchar();
    //imprimirHMM(estruturaHMM); getchar();
    //imprimirAlfa(numeroDeObservacoes, numeroEstados, alfa); getchar();
    //imprimirBeta(numeroEstados, numeroDeObservacoes, beta); getchar();
    //imprimirGama(numeroEstados, numeroDeObservacoes, gama); getchar();
    //imprimirGamaRabiner(numeroEstados, numeroDeObservacoes, gamaRabiner); getchar();

    //################################################
    //## passo-M - reestimação da matrizes A e B
    //################################################
    //-----------------------------------------MATRIZ A-----------------------------------------------------

    //NUMERADOR - matriz A
    for (i=0; i<numeroEstados; i++) {
        for(j=0; j<numeroEstados; j++) {
            numeradorA[i][j] = 0.0;
            for(t=0; t<numeroDeObservacoes-1; t++) {
                numeradorA[i][j] += gama[t][i][j];
            }
        }
    }

    //DENOMINADOR - matriz A
    for (i=0; i<numeroEstados; i++) {
        for(j=0; j<numeroEstados; j++) {
            denominadorA[i][j] = 0.0;
            for(t=0; t<numeroDeObservacoes-1; t++) {
                denominadorA[i][j] += gamaRabiner[t][i];
            }
        }
    }

    //MONTAGEM - matriz A
    for (i=0; i<numeroEstados; i++) {
        for(j=0; j<numeroEstados; j++) {
```

```
                if(denominadorA[i][j]!=0.0)estruturaHMM->a[i][j] = numeradorA[i][j]/denominadorA[i][j];
                else estruturaHMM->a[i][j] = 0.0;
            }
        }

        /*printf("MATRIZ A reestimada na iteracao %d:\n",contador);
        for (i=0; i<estruturaHMM->N; i++) {
        for (j=0; j<estruturaHMM->N; j++) {
            printf("a[%d][%d] = %lf = %e\n",i,j,estruturaHMM->a[i][j],estruturaHMM->a[i][j]);
}
    }
    getchar();*/

        //-----------------------------------------MATRIZ B-----------------------------------------------------

        //NUMERADOR - matriz B
        for (j=0; j<numeroEstados; j++) {
            for(k=0; k<numeroSaidas; k++) {
                numeradorB[j][k] = 0.0;
                for(t=0; t<numeroDeObservacoes; t++) {
                    if(posicaoCbkDaObservacao[t]==k) numeradorB[j][k] += gamaRabiner[t][j];
                }
            }
        }

        //DENOMINADOR - matriz B
        for (j=0; j<numeroEstados; j++) {
            for(k=0; k<numeroSaidas; k++) {
                denominadorB[j][k] = 0.0;
                for(t=0; t<numeroDeObservacoes; t++) {
                    denominadorB[j][k] += gamaRabiner[t][j];
                }
            }
        }

        //MONTAGEM - matriz B
        for (j=0; j<numeroEstados; j++) {
            for(k=0; k<numeroSaidas; k++) {
                if(denominadorB[j][k]!=0.0) {
                    if(numeradorB[j][k]!=0.0) estruturaHMM->b[j][k] = numeradorB[j][k]/denominadorB[j][k];
                    else estruturaHMM->b[j][k] /= 10.0;
                }
                else estruturaHMM->b[j][k] = 0.0;
            }
        }

        /*printf("MATRIZ B reestimada na iteracao %d (antes da normalizacao):\n",contador);
        for (i=0; i<estruturaHMM->N; i++) {
        for (j=0; j<estruturaHMM->M; j++) {
            printf("b[%d][%d] = %lf = %e\n",i,j,estruturaHMM->b[i][j],estruturaHMM->b[i][j]);
            if((i+j)%60==59) getchar();
}
    }
    getchar();*/

        //normaliza a matriz B que acabou de ser montada
        normalizaB(estruturaHMM);
        /*printf("MATRIZ B reestimada na iteracao %d (DEPOIS da normalizacao):\n",contador);
        for (i=0; i<estruturaHMM->N; i++) {
        for (j=0; j<estruturaHMM->M; j++) {
            printf("b[%d][%d] = %lf = %e\n",i,j,estruturaHMM->b[i][j],estruturaHMM->b[i][j]);
            if((i+j)%60==59) getchar();
}
    }
    getchar();*/


    //-----------------------------------------VETOR PI-----------------------------------------------------

        //calculo do i's valorPi
```

```
        for (i=0; i<numeroEstados; i++) valorPi[i] = gamaRabiner[0][i];

        //montagem de Pi na estrutura
        for (i=0; i<numeroEstados; i++) estruturaHMM->pi[i] = valorPi[i];

        probConvergAtual = calculaProbabilidade_X_S_dado_FI(estruturaHMM,mfcc,numeroDeObservacoes,tamanhoDaObservacao);

        deltaProbConverg = calculaDeltaProbConv(probConvergAtual,probConvergAntes);

    }while(deltaProbConverg>convergenciaBaum && contador<20);
    //----fim do laço do-while----//

    //coloca o gama e as posições das observações na estruturaHMM
    estruturaHMM->gama = gama;
    estruturaHMM->posicaoCbkDaObservacao = posicaoCbkDaObservacao;

    //libera os ponteiros usados
    free(alfa);free(beta);

    printf("\nnumero de iteracoes no algoritmo forward-backward: %d\n",contador);
    return 0;
}

double forward(HMM * estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDaObservacao, double * pr
    //variaveis que serao utilizadas no codigo
    double probLogForward, fatorNormalizacaoUltimoAlfa;
    double** alfa;
    int t,i;

    //alocação de memoria para a estrutura de dados posicaoCbkDaObservacao (relação entre a observação e a sua posição
    /*posicaoCbkDaObservacao = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //preenchimento da relação entre a observação e a sua posição no codebook (posicaoCbkDaObservacao)
    for(t=0; t<numeroDeObservacoes; t++) {

        posicaoCbkDaObservacao[t] = definePosicaoSimbolo(observacoes[t], estruturaHMM->codebook, estruturaHMM->M, tamar
        //printf("t: %d\t pos: %d",t,posicaoCbkDaObservacao[t]);getchar();
    }*/

    posicaoCbkDaObservacao = estruturaHMM->posicaoCbkDaObservacao;

    //obtenção de alfa
    alfa = calcularAlfaSemNormalizacao(0,estruturaHMM,observacoes,numeroDeObservacoes,tamanhoDaObservacao,&probLogForwa
    //alfa = calcularAlfa(0,estruturaHMM,observacoes,numeroDeObservacoes,tamanhoDaObservacao,&probLogForward, &fatorNor

    //calculo da probalidade da observação recebida ser gerada pelo modelo HMM estruturaHMM
    *probFinal = 0.0;
    for (i=0; i<estruturaHMM->N; i++) {
        *probFinal += (alfa[numeroDeObservacoes-1][i]);// * fatorNormalizacaoUltimoAlfa);
        //printf("alfa[%d][%d]: %lf = %e\n",numeroDeObservacoes-1,i,alfa[numeroDeObservacoes-1][i],alfa[numeroDeObserva
    }
    //getchar();
    //*probFinal += alfa[numeroDeObservacoes-1][(estruturaHMM->N)-1];
    //*probFinal = probLogForward;
    return 0.0;
}

HMM * geraHmmResultante(HMM ** estruturaHMM, int numHMM, int numeroCoeficientesPorSaida, int numeroDeObservacoes){
    int i,j,m,t,k,numEstados,numSaidas;
    HMM * hmm;
    double ****gama;
    double denominador;

    numEstados = estruturaHMM[0]->N;
    numSaidas = estruturaHMM[0]->M;

    //alocação de memória para os gamas
    gama = (double****)calloc(numHMM,sizeof(double));

    //estrutura que conterá o HMM resposta
```

```
    hmm = criaEstruturaHMM(numEstados, numSaidas, numeroCoeficientesPorSaida, numeroDeObservacoes, "Bruno");

    //carrega os gamas dos hmm's
    for(i=0; i<numHMM; i++) gama[i] = estruturaHMM[i]->gama;

    //estima o a
    for(i=0; i<numEstados; i++){

        //cálculo do denominador
        denominador = 0.0;
        for(m=0; m<numHMM; m++){
            for(t=0; t<numeroDeObservacoes; t++){
                for(k=0; k<numEstados; k++) denominador += gama[m][t][i][k];
            }

        }

        for(j=0; j<numEstados; j++){
            //cálculo do numerador
            hmm->a[i][j] = 0.0;
            for(m=0; m<numHMM; m++){
                for(t=0; t<numeroDeObservacoes; t++) hmm->a[i][j] += gama[m][t][i][j];
            }
            //considera o denominador
            if(denominador!=0.0) hmm->a[i][j] /= denominador;
            else hmm->a[i][j] = 0.0;
        }
    }

    //estima o b
    for (j=0; j<numEstados; j++) {

        for(k=0; k<numSaidas; k++) {

            //cálculo do numerador e do denominador de cada b[j][k]
            hmm->b[j][k] = 0.0;
            denominador = 0.0;
            for(m=0; m<numHMM; m++) {
                for(t=0; t<numeroDeObservacoes; t++) {
                    for(i=0; i<numEstados; i++){
                        if(estruturaHMM[m]->posicaoCbkDaObservacao[t]==k) hmm->b[j][k] += gama[m][t][i][j];
                        denominador += gama[m][t][i][j];
                    }
                }
            }

            //considera o denominador
            if(denominador!=0.0) hmm->b[j][k] /= denominador;
            else hmm->b[j][k] = 0.0;
        }
    }
    normalizaB(hmm);

    //escreve o pi
    for(i=0; i<numEstados; i++) {
        if(i==0) hmm->pi[i] = 1.0;
        else hmm->pi[i] = 0.0;
    }

    return hmm;
}

int calculaDelta(double *pDelta, HMM *hmm1,HMM *hmm2, double *** feature, int numeroDeFeatures, int numeroDeObservacoes
    int i,j,k;
    double ***alfaHMM1,***alfaHMM2;
    double resultHMM1,resultHMM2,logHMM1,logHMM2,delta,probLog;

    //alocação de memória para os alfas do hmm1
    alfaHMM1 = (double***)calloc(numeroDeFeatures, sizeof(double));
```

```
    //alocação de memória para os alfas do hmm2
    alfaHMM2 = (double***)calloc(numeroDeFeatures, sizeof(double));

    //cálculo de todos os alfas para o hmm1
    for(i=0; i<numeroDeFeatures; i++) alfaHMM1[i] = calcularAlfaSemNormalizacao(0, hmm1, feature[i] , numeroDeObservacc

    //cálculo de todos os alfas para o hmm2
    for(i=0; i<numeroDeFeatures; i++) alfaHMM2[i] = calcularAlfaSemNormalizacao(0, hmm2, feature[i] , numeroDeObservacc

    //imprimirHMM(hmm1);getchar();
    //imprimirHMM(hmm2); getchar();

    //soma os alfas para calcular o resultado de cada hmm
    resultHMM1 = 0.0;
    resultHMM2 = 0.0;
    for(i=0; i<numeroDeFeatures; i++){
        //imprimirAlfa(numeroDeObservacoes, hmm1->N, alfaHMM1[i]); getchar();
        for(k=0; k<hmm1->N; k++) {
            resultHMM1 += alfaHMM1[i][numeroDeObservacoes-1][k];
            resultHMM2 += alfaHMM2[i][numeroDeObservacoes-1][k];
        }
    }

    //calcula o delta (siginifica o quanto longe está o número resultHMM2 de resultHMM1
    if(resultHMM1>=resultHMM2) delta = 1-(resultHMM2/resultHMM1);
    else delta = 1-(resultHMM1/resultHMM2);

    //passa o número calculado por referência
    *pDelta = delta;

    return 1;
}
```

Módulo secundário 6 - matrizUtil.c:

```
int obtemExpoente(double num){
    int expoente = 0;
    if(num>1.0){
        while(num>=10.0){
            num /= 10.0;
            expoente++;
        }
    }
    else if(num<=1.0 && num>0.0) {
        while(num<1.0){
            num *= 10.0;
            expoente--;
        }
    }
    return expoente;
}

double** matrizA_menos_matrizB(int numLinhas, int numCols, double **matrizA, double **matrizB){
    int i,j;
    double ** matrizResult;

    matrizResult = (double**) calloc(numLinhas, sizeof(double));
    for(i=0; i<numLinhas; i++) matrizResult[i] = (double*)calloc(numCols,sizeof(double));

    for(i=0;i<numLinhas;i++){
        for(j=0;j<numCols;j++){
            matrizResult[i][j] = matrizA[i][j] - matrizB[i][j];
        }
    }

    return matrizResult;
}

double** matrizA_mais_matrizB(int numLinhas, int numCols, double **matrizA, double **matrizB){
    int i,j;
    double ** matrizResult;
```

```
    matrizResult = (double**) calloc(numLinhas, sizeof(double));
    for(i=0; i<numLinhas; i++) matrizResult[i] = (double*)calloc(numCols,sizeof(double));

    for(i=0;i<numLinhas;i++){
        for(j=0;j<numCols;j++){
            matrizResult[i][j] = matrizA[i][j] + matrizB[i][j];
        }
    }

    return matrizResult;
}

double** matriz_divididoPor_escalar(int numLinhas, int numCols, double **matriz, double escalar){
    int i,j;
    double ** matrizResult;

    matrizResult = (double**) calloc(numLinhas, sizeof(double));
    for(i=0; i<numLinhas; i++) matrizResult[i] = (double*)calloc(numCols,sizeof(double));

    for(i=0;i<numLinhas;i++){
        for(j=0;j<numCols;j++){
            matrizResult[i][j] = matriz[i][j] / escalar;
        }
    }

    return matrizResult;
}

double** matriz_vezes_escalar(int numLinhas, int numCols, double **matriz, double escalar){
    int i,j;
    double ** matrizResult;

    matrizResult = (double**) calloc(numLinhas, sizeof(double));
    for(i=0; i<numLinhas; i++) matrizResult[i] = (double*)calloc(numCols,sizeof(double));

    for(i=0;i<numLinhas;i++){
        for(j=0;j<numCols;j++){
            matrizResult[i][j] = matriz[i][j] * escalar;
        }
    }

    return matrizResult;
}

double* vetorA_menos_vetorB(int numElementos, double *vetorA, double *vetorB){
    int i;
    double * vetorResult;

    vetorResult = (double*) calloc(numElementos, sizeof(double));

    for(i=0;i<numElementos;i++){
        vetorResult[i] = vetorA[i] - vetorB[i];
    }

    return vetorResult;
}

double* vetorA_mais_vetorB(int numElementos, double *vetorA, double *vetorB){
    int i;
    double * vetorResult;

    vetorResult = (double*) calloc(numElementos, sizeof(double));

    for(i=0;i<numElementos;i++){
        vetorResult[i] = vetorA[i] + vetorB[i];
    }

    return vetorResult;
}
```

```
double* vetor_divididoPor_escalar(int numElementos, double *vetor, double escalar){
    int i;
    double * vetorResult;

    vetorResult = (double*) calloc(numElementos, sizeof(double));

    for(i=0;i<numElementos;i++){
        vetorResult[i] = vetor[i] / escalar;
    }

    return vetorResult;
}

double* vetor_vezes_escalar(int numElementos, double *vetor, double escalar){
    int i;
    double * vetorResult;

    vetorResult = (double*) calloc(numElementos, sizeof(double));

    for(i=0;i<numElementos;i++){
        vetorResult[i] = vetor[i] * escalar;
    }

    return vetorResult;
}
double** vetor_vezes_vetorTransposto(int numElementos, double *vetor){
    int i,linha,coluna;
    double ** matrizResult;

    matrizResult = (double**) calloc(numElementos, sizeof(double));
    for(i=0; i<numElementos; i++) matrizResult[i] = (double*)calloc(numElementos,sizeof(double));

    for(linha=0;linha<numElementos;linha++){
        for(coluna=0;coluna<numElementos;coluna++){
            matrizResult[linha][coluna] = vetor[linha] * vetor[coluna];
        }
    }

    return matrizResult;
}

double aplicaProdutoEscalarDaDiferenca__(double * observacao, double * vetorCodeBook, int tamanhoVetor) {
    int i;
    double resultado;
    double * diferenca;

    diferenca = vetorA_menos_vetorB(tamanhoVetor, observacao, vetorCodeBook);
    resultado = 0.0;

    for(i=0; i<tamanhoVetor; i++) {
        resultado += diferenca[i] * diferenca[i];
    }

    if(resultado>0.0) return resultado;
    else return -resultado;
}

int definePosicaoSimbolo__(double * observacao, double ** codebook, int tamanhoAlfabeto, int tamanhoDaObservacao){
    int i,posicao;
    double produtoEscalar, produtoEscalarAux ;

    produtoEscalar = 9999999999999999999.0;
    posicao = 0;

    for(i=0; i<tamanhoAlfabeto; i++){
        produtoEscalarAux = aplicaProdutoEscalarDaDiferenca__(observacao, codebook[i], tamanhoDaObservacao);
        if(produtoEscalarAux < produtoEscalar) {
            produtoEscalar = produtoEscalarAux;
            posicao = i;
```

```
        }
    }
    return posicao;
}
```

Módulo secundário 7 - hmm.c:

```
#include <stdio.h>
#include <math.h>
#include <time.h>

#include "hmm.h"
#include "algoritmosHMM.h"
#include "emissor.h"


int normalizaA(HMM * estruturaHMM){
    int i,j;
    double * somatorio;

    //alocação dos somatórios que serão usados para a normalização
    somatorio = (double*)calloc(estruturaHMM->N,sizeof(double));

    //cálculo das somas dos elementos pertecentes a cada estado
    for(i=0; i<estruturaHMM->N; i++){
            somatorio[i] = 0.0;
            for(j=0; j<estruturaHMM->N; j++) somatorio[i] += estruturaHMM->a[i][j];
    }

    //faz a normalização (divide todos os elementos pela soma)
    for(i=0;i<estruturaHMM->N;i++){
        for(j=0;j<estruturaHMM->N;j++) {
            if(somatorio[i]!=0)estruturaHMM->a[i][j] /= somatorio[i];
        }
    }

    return 1;
}

int normalizaB(HMM * estruturaHMM){
    int i,j;
    double * somatorio;

    //alocação dos somatórios que serão usados para a normalização
    somatorio = (double*)calloc(estruturaHMM->M,sizeof(double));

    //cálculo das somas dos elementos pertecentes a cada estado
    for(i=0; i<estruturaHMM->N; i++){
            somatorio[i] = 0.0;
            for(j=0; j<estruturaHMM->M; j++) somatorio[i] += estruturaHMM->b[i][j];
    }

    //faz a normalização (divide todos os elementos pela soma)
    for(i=0;i<estruturaHMM->N;i++){
        for(j=0;j<estruturaHMM->M;j++) {
            if(somatorio[i]>0.0) estruturaHMM->b[i][j] /= somatorio[i];
        }
    }

    return 1;
}

double obterNumAleatorio(){
    int i;
    double var[10];
    double numAleatorio;

    //obtém 10 números aleatórios
    for(i=0; i<10; i++) {
        var[i] = rand();
        if(var[i]<0.0) var[i]*=(-1.0);
```

```
    }

    //multiplica todos os numeros aleatorios obtidos
    numAleatorio = 1.0;
    for(i=0; i<10; i++) numAleatorio *= var[i];

    //faz um mod 10 para a multiplicação obtida
    if(numAleatorio>1.0){
        while(numAleatorio>=10.0) numAleatorio /= 10.0;
    }
    else {
        while(numAleatorio<1.0 && numAleatorio>0.0) numAleatorio *= 10.0;
    }

    //o número aleatório entre 0 e 10 está pronto
    return numAleatorio;
}

int defineProbabilidadesIniciais(HMM * estruturaHMM){
    int i,j,numeroEstados,numeroSaidas;

    numeroEstados = estruturaHMM->N;
    numeroSaidas = estruturaHMM->M;

    //preenchimento da matriz A
    for(i=0;i<numeroEstados;i++){
        for(j=0;j<numeroEstados;j++) {
            while( (estruturaHMM->a[i][j]=obterNumAleatorio()) <= 0.0);
        }
    }
    //zerar os a's
    for(i=0;i<numeroEstados;i++){
        for(j=0;j<numeroEstados;j++) {
            if(i!=j && j!=(i+1) && j!=(i+2)) estruturaHMM->a[i][j] = 0.0;
        }
    }
    normalizaA(estruturaHMM);

    //preenchimento da matriz B
    for(i=0;i<numeroEstados;i++){
        for(j=0;j<numeroSaidas;j++){
            while( (estruturaHMM->b[i][j]=obterNumAleatorio()) <= 0.0);
        }
    }
    normalizaB(estruturaHMM);

    //preenchimento da matriz Pi
    estruturaHMM->pi[0] = 1.0;
    for(i=1;i<numeroEstados;i++) estruturaHMM->pi[i] = 0.0;

    return 0;
}

HMM* criaEstruturaHMM(int numeroEstados, int numeroSaidas, int numeroCoeficientesPorSaida, int numeroDeObservacoes, cha
    int i,j,k;

//aloca memória para a estrutra que será criada
HMM *estruturaHMM = (HMM*)calloc(1,sizeof(HMM));

    //campos que vêm dos parâmetros
    estruturaHMM->N = numeroEstados;
    estruturaHMM->M = numeroSaidas;
    estruturaHMM->palavraModelada = palavraModelada;

    //alocação dinâmica para a matriz de tamanho numeroEstados X numeroEstados
    estruturaHMM->a = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) estruturaHMM->a[i] = (double*)calloc(numeroEstados,sizeof(double));

    //alocação dinâmica para a matriz de tamanho numeroEstados X numeroSimbolos
    estruturaHMM->b = (double**)calloc(numeroEstados,sizeof(double));
```

```
    for(i=0; i<numeroEstados; i++) estruturaHMM->b[i] = (double*)calloc(numeroSaidas,sizeof(double));

    //alocação dinâmica para a matriz de tamanho numeroEstados
    estruturaHMM->pi = (double*)calloc(numeroEstados,sizeof(double));

    //alocação dinâmica para a matriz de tamanho numeroEstados X numeroEstados
    estruturaHMM->codebook = (double**)calloc(numeroSaidas,sizeof(double));
    for(i=0; i<numeroSaidas; i++) estruturaHMM->codebook[i] = (double*)calloc(numeroCoeficientesPorSaida,sizeof(double)

    //alocação para o gama (3 dimensões)
    estruturaHMM->gama = (double***)calloc(numeroDeObservacoes,sizeof(double));
for(i=0; i<numeroDeObservacoes; i++) estruturaHMM->gama[i] = (double**)calloc(numeroEstados,sizeof(double));
for(i=0; i<numeroDeObservacoes; i++) {
        for(j=0; j<numeroEstados; j++){
            estruturaHMM->gama[i][j] = (double*)calloc(numeroEstados,sizeof(double));
        }
    }

    //retorna estrutura
    return estruturaHMM;
}
int treinaModelagemHMM(HMM * estruturaHMM, double **mfcc, int numeroDeObservacoes, int tamanhoDaObservacao, double conv

    forwardBackward(estruturaHMM, mfcc, numeroDeObservacoes, tamanhoDaObservacao, convergenciaBaum);

    return 0;
}

int gravarHMMTreinado(HMM * hmm, char * nomeArquivo, int tamanhoCodeBook, int numeroCoeficientesPorSaida){
    int i,j,valorInt;
    double valorDouble;

    //abertura do arquivo
    FILE * arq;
    arq = fopen(nomeArquivo,"wb");
    if(arq==NULL) {
        printf("erro: Nao foi possivel criar o arquivo para escrever o HMM treinado"); getchar();
        exit(1);
    }

    //escrita de N e M
    valorInt = hmm->N;
    fwrite(&valorInt, sizeof(int), 1, arq);
    valorInt = hmm->M;
    fwrite(&valorInt, sizeof(int), 1, arq);

    //escrita de pi
    for (i=0; i<hmm->N; i++) {
        valorDouble = hmm->pi[i];
        fwrite(&valorDouble, sizeof(double), 1, arq);
}

    //escrita de a
     for (i=0; i<hmm->N; i++) {
        for (j=0; j<hmm->N; j++) {
            valorDouble = hmm->a[i][j];
            fwrite(&valorDouble, sizeof(double), 1, arq);
}
}

    //escrita de b
    for (i=0; i<hmm->N; i++) {
        for (j=0; j<hmm->M; j++) {
            valorDouble = hmm->b[i][j];
            fwrite(&valorDouble, sizeof(double), 1, arq);
}
}

    //escrita do codebook
    for (i=0; i<tamanhoCodeBook; i++) {
```

```
        for (j=0; j<numeroCoeficientesPorSaida; j++) {
            valorDouble = hmm->codebook[i][j];
            fwrite(&valorDouble, sizeof(double), 1, arq);
        }
    }

    //fecha a stream do arquivo
    fclose(arq);

    return 1;
}

HMM* criaEstruturaHMMCarregar(int numeroEstados, int numeroSaidas, int numeroCoeficientesPorSaida, int numeroDeObservac
    int i,j,k;

//aloca memória para a estrutra que será criada
HMM *estruturaHMM = (HMM*)calloc(1,sizeof(HMM));

    //campos que vêm dos parâmetros
    estruturaHMM->N = numeroEstados;
    estruturaHMM->M = numeroSaidas;
    estruturaHMM->palavraModelada = palavraModelada;

    //alocação dinâmica para a matriz de tamanho numeroEstados X numeroEstados
    estruturaHMM->a = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) estruturaHMM->a[i] = (double*)calloc(numeroEstados,sizeof(double));

    //alocação dinâmica para a matriz de tamanho numeroEstados X numeroSimbolos
    estruturaHMM->b = (double**)calloc(numeroEstados,sizeof(double));
    for(i=0; i<numeroEstados; i++) estruturaHMM->b[i] = (double*)calloc(numeroSaidas,sizeof(double));

    //alocação dinâmica para a matriz de tamanho numeroEstados
    estruturaHMM->pi = (double*)calloc(numeroEstados,sizeof(double));

    /*//alocação dinâmica para a matriz de tamanho numeroEstados X numeroEstados
    estruturaHMM->codebook = (double**)calloc(numeroSaidas,sizeof(double));
    for(i=0; i<numeroSaidas; i++) estruturaHMM->codebook[i] = (double*)calloc(numeroCoeficientesPorSaida,sizeof(double)

    //alocação para o gama (3 dimensões)
    estruturaHMM->gama = (double***)calloc(numeroDeObservacoes,sizeof(double));
for(i=0; i<numeroDeObservacoes; i++) estruturaHMM->gama[i] = (double**)calloc(numeroEstados,sizeof(double));
for(i=0; i<numeroDeObservacoes; i++) {
        for(j=0; j<numeroEstados; j++){
            estruturaHMM->gama[i][j] = (double*)calloc(numeroEstados,sizeof(double));
        }
    }
    */
    //retorna estrutura
    return estruturaHMM;
}

HMM* carregarHMM(char * nomeArquivo, int numeroEstados, int tamanhoCodeBook, int numeroCoeficientesPorSaida, int numero
    int i,j,valorInt;
    double valorDouble;
    double ** codebook;
    HMM *hmm;

    //abertura do arquivo
    FILE * arq;
    arq = fopen(nomeArquivo,"rb");

    if(arq==NULL) {
        printf("erro: Nao foi possivel abrir o arquivo de carregamento do HMM"); getchar();
        exit(1);
    }

    hmm = criaEstruturaHMMCarregar(numeroEstados, tamanhoCodeBook, numeroCoeficientesPorSaida, numeroDeObservacoes, pal
```

```
    //leitura de N e M
    fread(&(hmm->N), sizeof(int), 1, arq);
    //hmm->N = valorInt;
    fread(&(hmm->M), sizeof(int), 1, arq);
    //hmm->M = valorInt;

    //leitura de pi
    for (i=0; i<hmm->N; i++) {
        fread(&(hmm->pi[i]), sizeof(double), 1, arq);
        //printf("valor: %e\n",valorDouble); getchar();
        //hmm->pi[i] = valorDouble;
}

    //leitura de a
     for (i=0; i<hmm->N; i++) {
        for (j=0; j<hmm->N; j++) {
            fread(&(hmm->a[i][j]), sizeof(double), 1, arq);
            //hmm->a[i][j] = valorDouble;
}
}

    //leitura de b
    for (i=0; i<hmm->N; i++) {
        for (j=0; j<hmm->M; j++) {
            fread(&(hmm->b[i][j]), sizeof(double), 1, arq);
            //hmm->b[i][j] = valorDouble;
}
}
    //system("pause");
    //leitura do codebook
    //passo 1: alocacao da memoria necessaria
    /*codebook = (double**) calloc(tamanhoCodeBook,sizeof(double));
    for(i=0; i<tamanhoCodeBook; i++) codebook[i] = (double*)calloc(numeroCoeficientesPorSaida,sizeof(double));

    //leitura do codebook
    //passo 2: leitura do arquivo HMM para o codebook alocado
    for (i=0; i<tamanhoCodeBook; i++) {
        for (j=0; j<numeroCoeficientesPorSaida; j++) {
            fread(&(codebook[i][j]), sizeof(double), 1, arq);
            //codebook[i][j] = valorDouble;
        }
    }

    //leitura do codebook
    //passo 3: coloca o codebook lido na estrutura hmm e libera o ponteiro
    hmm->codebook = codebook;
    //free(codebook);
    */
    //fecha a stream do arquivo
    fclose(arq);

    return hmm;
}
```

A.1.4 Código dos cabeçalhos (arquivos .h) entre o arquivo main.c e os módulos principais

telas.h - correspondente ao módulo principal 1:

```
int desenharTelaInicial(int argc, char * argumento1);
int informaParametrosPassados(char ** argv);
```

motadorFeature.h - correspondente ao módulo principal 2:

```
double ** montarFeature();
```

treinamento.h - correspondente ao módulo principal 3:

```
#include <stdio.h>

#include "defines.h"
#include "hmm.h"
#include "emissor.h"
#include "matrizUtil.h"

HMM * treinar1aVez(double **feature, int numeroDeObservacoes, double **codebook){
    int t, *posicaoCbkDaObservacao;

    //alocacao da modelagem matemática
    HMM * estruturaHMM = (HMM*)criaEstruturaHMM(NUMERO_ESTADOS, NUMERO_SAIDAS, TAMANHO_DA_OBSERVACAO, numeroDeObservacc

    //alocação de memoria para a estrutura de dados posicaoCbkDaObservacao (relação entre a observação e a sua posição
    posicaoCbkDaObservacao = (int*)calloc(numeroDeObservacoes,sizeof(int));

    //carrega o codebook na estrutura criada
    estruturaHMM->codebook = codebook;

    //define matrizes de probabilidades iniciais. Primeiramente com probabilidades iguais para todos os estados
    defineProbabilidadesIniciais(estruturaHMM);

    //calcula e coloca no HMM os elementos de posicaoCbkDaObservacao
    for(t=0; t<numeroDeObservacoes; t++) posicaoCbkDaObservacao[t] = definePosicaoSimbolo__(feature[t], estruturaHMM->c
    estruturaHMM->posicaoCbkDaObservacao = posicaoCbkDaObservacao;

    //treina modelagem de acordo com os coeficientes extraidos do sinal de áudio
    treinaModelagemHMM(estruturaHMM,feature,numeroDeObservacoes,TAMANHO_DA_OBSERVACAO,CONVERGENCIA_BAUM);

    return estruturaHMM;

}

HMM * treinar2aVezEmDiante(HMM * estruturaHMM, double **feature, int numeroDeObservacoes){
    //carrega o hmm passado como parametro
    //HMM * estruturaHMM = (HMM*)carregarHMM(nomeHmm, NUMERO_ESTADOS, NUMERO_SAIDAS, 3*NUMERO_DE_COEFICIENTES, numeroDe

    //imprimirHMM(estruturaHMM); getchar();
    //imprimirCodeBook(estruturaHMM, 3*NUMERO_DE_COEFICIENTES); getchar();

    //treina modelagem de acordo com os coeficientes extraidos do sinal de áudio
    treinaModelagemHMM(estruturaHMM,feature,numeroDeObservacoes,TAMANHO_DA_OBSERVACAO,CONVERGENCIA_BAUM);

    //imprimirHMM(estruturaHMM); getchar();

    return estruturaHMM;

}
```

reconhecimento.h - correspondente ao módulo principal 4:

```
int reconhecer(int numHMM, HMM ** estruturaHMM, double ** observacoes, int numeroDeObservacoes, double ** codebook);
```

A.1.5 Código dos cabeçalhos (arquivos .h) entre os módulos principais e os módulos secundários

manipuladorWave.h:

```
typedef struct struct_audio {
int nsamples;
double * stream;
} audio;

audio obterStreamVozAPartirDoArquivoDeAudio(char * nomeArquivo, int taxaAmostragem, int tempoComandoVez_ms, int tamCabe
double * retiraCabecalhoArquivoWav(double *streamVoz, int nsamples, int tamanhoCabecalhoWav);
double * normalizaAudio(double * streamVoz, int numeroDeAmostrasNoComando);
```

extratorMelCepstral.h:

```
int extrairCoefientesMelCepstraisDeUmBloco (double *bloco, double **mfcc, int numeroCoeficientesMelCepstrais, int numer
```

algoritmosHMM.h:

```
int forwardBackward(HMM * estruturaHMM, double **mfcc, int numeroDeObservacoes, int tamanhoDaObservacao, double converg
double ** calcularAlfa(HMM* estruturaHMM, double ** observacoes ,int numeroDeObservacoes, int tamanhoDaObservacao,doubl
double ** calcularBeta(HMM* estruturaHMM, int* observacoes  ,int numeroDeObservacoes);
double viterbi(HMM *estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDaObservacao,int *sequenci
double forward(HMM * estruturaHMM, double ** observacoes, int numeroDeObservacoes, int tamanhoDaObservacao, double * pr
HMM * geraHmmResultante(HMM ** estruturaHMM, int numHMM, int numeroCoeficientesPorSaida, int numeroDeObservacoes);
int calculaDelta(double *pDelta, HMM *hmm1, HMM *hmm2, double *** feature, int numeroDeFeatures, int numeroDeObservacoe
```

A.1.6 Código dos cabeçalhos (arquivos .h) internos aos módulos secundários

MelCepstralCoefficients.h:

```
double * winFFT(double *stream, int nFFT);
double* computeBoundaries(int nF, double fLow, double fHigh, int nFFT, double Fs);
double ** HmBank(double * f, int nFFT, int nF);
double * MelWrapping (double * XFFT, double ** Hm, int nF, int nFFT);
double * MelCepstralCoefficients (double *S, int nF, int nMel);
```

srfft.h:

```
void  srfft(float *xr, float *xi, int logm);
```

matrizUtil.h:

```
double** matrizA_menos_matrizB(int numLinhas, int numCols, double **matrizA, double **matrizB);
double** matrizA_mais_matrizB(int numLinhas, int numCols, double **matrizA, double **matrizB);
double** matriz_divididoPor_escalar(int numLinhas, int numCols, double **matriz, double escalar);
double** matriz_vezes_escalar(int numLinhas, int numCols, double **matriz, double escalar);
double* vetorA_menos_vetorB(int numElementos, double *vetorA, double *vetorB);
double* vetorA_mais_vetorB(int numElementos, double *vetorA, double *vetorB);
double* vetor_divididoPor_escalar(int numElementos, double *vetor, double escalar);
double* vetor_vezes_escalar(int numElementos, double *vetor, double escalar);
double** vetor_vezes_vetorTransposto(int numElementos, double *vetor);
int obtemExpoente(double num);
double aplicaProdutoEscalarDaDiferenca__(double * observacao, double * vetorCodeBook, int tamanhoVetor);
int definePosicaoSimbolo__(double * observacao, double ** codebook, int tamanhoAlfabeto, int tamanhoDaObservacao);
```

hmm.h:

```
/**
 * Universidade de Brasília, UnB
 * Brasília, 20 de maio de 2008
 *
 * - Autor: Bruno de Assis Rolim
 *
 * - Arquivo: hmm.h
 *
 * - Referência Bibliográfica:
 *  Título: Spoken Language Processing
 *  Autores: Xuedong Huang, Alex Acero e Hsiao-Wuen Hon
 *  Capítulo usado: 8 (Hidden Markov Models)
 */

/* estrutura que representa o HMM */
typedef struct {
    char * palavraModelada;
int N; //número de estados do modelo;  Q={1,2,...,N}
int M;  //número de símbolos do alfabeto em questão; V={1,2,...,M}; tamanho do codebook
    double *pi; //tabela de dimensÃ£o N: pi[i] é a probabilidade do estado i ser o estado inicial
double **a; //tabela de dimensão N x N: a[i][j] é a probabilidade de ir do estado i (no tempo t) para o estado j (no te
double **b; //tabela de dimensÃ£o N x M. b[j][k] é a probabilidade do símbolo k ser observado quando o estado atual é
double **codebook;
double ***gama;
```

```
int * posicaoCbkDaObservacao;
} HMM;

int defineProbabilidadesIniciais(HMM * estruturaHMM);
HMM* criaEstruturaHMM(int numeroEstados, int numeroSaidas, int numeroCoeficientesPorSaida, int numeroDeObservacoes, cha
int treinaModelagemHMM(HMM * estruturaHMM, double **mfcc, int numeroDeObservacoes, int tamanhoDaObservacao, double conv
int gravarHMMTreinado(HMM * hmm, char * nomeArquivo, int tamanhoCodeBook, int numeroCoeficientesPorSaida);
HMM* carregarHMM(char * nomeArquivo, int numeroEstados, int tamanhoCodeBook, int numeroCoeficientesPorSaida, int numero
int normalizaB(HMM * estruturaHMM);
```