UNIVERSIDADE FEDERAL DO
RIO DE JANEIRO

ESCOLA POLITÉCNICA

DEPARTAMENTO DE ELETRÔNICA
E DE COMPUTAÇÃO

IMPLEMENTAÇÃO DE UM CODIFICADOR DE VOZ CELP EM TEMPO REAL

Autor:

Filipe Castello da Costa Beltrão Diniz

Orientador:

Prof. Sérgio Lima Netto, Ph.D.

Examinador:

Prof. Luiz Wagner Pereira Biscainho, D.Sc.

Examinador:

Prof. Sérgio Palma da Justa Medeiros, D.Sc.

DEL
Maio de 2003

Agradecimentos

Agradeço:

- Ao professor Sergio Lima Netto, pela orientação acadêmica, incentivos, idéias e conversas;
- Aos professores Luiz Wagner Pereira Biscainho e Sérgio Palma da Justa Medeiros por se disporem a compor a banca avaliadora;
- A todos os amigos do Laboratório de Processamento de Sinais, principalmente Rodrigo Meirelles pela companhia em tempos de DSP e pela contribuição nas gravações, João Baptista, por toda a ajuda e paciência no estudo de DSPs, Rodrigo Coura Torres, pela ajuda na programação, Ranniery da Silva Maia pelo auxílio prestado tão prontamente de tão longe, Bruno Oliveira por me orientar no começo do projeto, Rafael Teruszkin pela contribuição em relação às classes e Daniel Rojtenberg pelas idéias para o dicionário fixo;
- A todos os amigos da turma de Engenharia Eletrônica e de Computação da UFRJ, pela indispensável amizade;
- A todos aqueles que tiveram qualquer tipo de influência no projeto, mesmo que indireta;
- À Patrícia por estar a meu lado por todo o tempo da graduação, por acompanhar a evolução do projeto e por me ajudar a ensaiar a apresentação.

Índice

Resumo

Este trabalho descreve a implementação de um sistema de codificação de voz CELP em tempo real, isto é, de modo a fazer com que o sinal seja adquirido por um microfone conectado à placa de som do computador, e, ao mesmo tempo, codificado, decodificado e reproduzido nas caixas de som.

O software que implementa o sistema foi desenvolvido em Linux utilizando a Programação Orientada a Objeto em C++, visando a simplificar trabalhos de atualização e análise. Foram criadas classes que visam a aproveitar melhor os esforços de programação tanto deste trabalho quanto de outros. Alem disso, o funcionamento e a manipulação destas classes foram detalhadas em uma espécie de manual de uso.

A avaliação do sistema foi feita utilizando predominantemente a Razão Sinal / Ruído Segmentada Perceptiva devido ao fato de que esta forma de medida apresenta uma correlação razoável com formas subjetivas de medida. Essas medidas também foram feitas através de uma classe criada especificamente para este fim.

Por fim, chegou-se a um codificador em tempo real para Linux em *software*, obtendo-se uma Razão Sinal / Ruído média de 15.91 dB e máxima de 15.65 dB;

Capítulo 1

Introdução

1.1 Proposta do Trabalho

Este trabalho descreve a implementação de um sistema de codificação de voz CELP (em tempo real) e avalia o desempenho do mesmo. Por funcionamento em tempo real denota-se fazer com que o sinal seja adquirido por um microfone conectado à placa de som do computador, e, ao mesmo tempo, codificado, decodificado e reproduzido nas caixas de som. O *software* que implementa o sistema foi desenvolvido em Linux utilizando a Programação Orientada a Objeto em C++, visando a simplificar trabalhos de atualização e análise.

1.2 Princípios da Tecnologia Digital

Quando se fala em telecomunicações, refere-se à transmissão de informação à distância. O homem vem evoluindo nesse setor de forma exponencial nas últimas décadas, o que indica que há um crescimento igualmente rápido em relação às necessidades tanto das pessoas quanto dos meios de comunicação em si. A sociedade tem mostrado incrível interesse em ser capaz de conversar com qualquer um, de qualquer lugar, a qualquer hora. Para suprir tais requisitos, os dispositivos criados devem ser projetados de forma a acompanhar todo esse progresso. Assim, deve-se buscar as maneiras mais eficientes de fazer com que um meio de telecomunicação satisfaça todas as necessidades supramencionadas.

Com a era digital, os sinais, sejam estes voz, vídeo, áudio ou dados, podem ser tratados de forma muito mais eficaz e simplificada, por evitar dificuldades inerentes a projetos analógicos. Isto acontece, pois tal tecnologia permite o uso tanto de computadores como de algoritmos computacionais. Isto é uma grande vantagem, visto que há grande difusão de dispositivos que fazem esse tipo de papel, como os DSP's.

A representação digital de um sinal subentende três etapas: amostragem, quantização e codificação. A amostragem é o processo de considerar um sinal analógico apenas em momentos que são múltiplos inteiros de um certo tempo (denominado período de amostragem). Isso torna o sinal discreto no tempo, mas o sinal permanece contínuo na amplitude.

Para torná-lo integralmente discreto, isto é, no tempo e na amplitude, deve-se proceder à quantização. Esta é o processo em que a amplitude do sinal em um dado momento é associada a um nível discreto (bem definido e invariável). O número de níveis será dado pela *n*-ésima potência de 2, onde n é o número de bits disponíveis para representar o sinal. De modo geral, quanto maior for a quantidade de *bits* usada melhor será a qualidade do sinal digital reconstituído ao final da transmissão. Dessa forma, tal quantidade pode ser vista como a resolução do sinal.

A codificação (etapa estudada neste trabalho) é a forma através da qual os *bits* representarão o sinal já amostrado e quantizado. Dependendo do tipo de codificação, existe uma taxa de codificação diferente. Esta taxa denota a quantidade de *bits* necessária para representar o sinal em um determinado intervalo de tempo. Assim, pode-se ver, claramente, que há certos tipos de codificação que são mais eficientes do que outros. Quanto menor a taxa de codificação, maior será o número de sinais que poderão ser transmitidos dentro de uma banda de certa largura ou menor será o "esforço" gasto para transmitir um dado sinal.

Dentre os vários modos de se codificar um sinal de voz, o que se destaca em meio àqueles que possuem baixa taxa de codificação é o CELP (*Code Excited Linear Prediction*). Este faz uso de regressões lineares e de dicionários que contêm excitações que atuam como a base para a formação do sinal reconstruído. Desta forma, o sistema CELP permite uma qualidade muito boa para o sinal de voz reconstruído e demanda uma taxa de codificação relativamente baixa, sendo, por este motivo, largamente utilizado em vários padrões de telefonia digital.

Como foi dito acima, existem aplicações que possuem capacidade inerentemente baixa e onde a largura da banda é bem escassa. Em aplicações desse tipo, há uma necessidade básica e essencial de codificar a voz com taxas bem baixas, enquanto se mantém uma fidelidade ou qualidade de reprodução aceitáveis.

1.3 Organização da Dissertação

O Capítulo 2 fornece uma visão geral dos tipos de codificação de voz usados, bem como os princípios para uma codificação mais eficiente e as alternativas que os utilizam. Além disso, há uma explicação detalhada sobre o sistema CELP desenvolvido em [1] e aperfeiçoado em [2], enfocando principalmente seu funcionamento.

O Capítulo 3 apresenta informações sobre como foi realizada a implementação em tempo real. Além disso, são explicados o modo através do qual é feita a manipulação da placa de som e a metodologia de programação utilizada, detalhando toda a estrutura das classes implementadas em C++.

O capítulo 4 propõe como se deve dar o funcionamento do sistema CELP da forma através da qual foi desenvolvido neste trabalho, apresentando o algoritmo do sistema. Além disso, é realizada uma avaliação de desempenho do mesmo.

O Capítulo 5 mostra o resumo de toda a dissertação, com comentários e conclusões a respeito das contribuições resultantes deste trabalho e propostas para trabalhos futuros.

Capítulo 2

Sistemas de Codificação

Neste capítulo, é apresentada uma visão geral dos tipos de codificação de voz usados. Na Seção 2.2, há uma explicação sobre codificadores de forma de onda. Na Seção 2.3, há uma explicação sobre codificadores de fonte. Na Seção 2.4, há uma explicação sobre codificadores híbridos, enfatizando o funcionamento dos codificadores do tipo CELP. Na Seção 2.5, são propostas melhorias ao sistema CELP apresentado. Na Seção 2.6, é descrito o funcionamento do sistema CELP já com estas melhorias. Na Seção 2.7, apresentam-se as características do sistema e os ganhos por ele promovidos.

2.1 Tipos de Codificadores

Para codificar a voz humana, existem basicamente três tipos de codificadores:

- Codificadores de Forma de Onda
- Codificadores de Fonte ou Paramétricos
- Codificadores Híbridos

A diferença entre estes está na filosofia usada para realizar a codificação. Enquanto os codificadores de forma de onda enviam o sinal de voz propriamente dito (ou variantes deste), os demais manipulam este sinal e transmitem os parâmetros resultantes desta operação.

De modo geral, a qualidade do sinal de voz reconstruído e a taxa de codificação se relacionam da seguinte forma: os codificadores de forma de onda são aqueles que apresentam uma qualidade muito boa, mas com o custo de uma taxa de codificação muito alta. Em contra partida, os codificadores de fonte apresentam taxas bem reduzidas, o que faz com que apresentem uma qualidade precária.

Os codificadores híbridos unem as boas características desses dois tipos. O modo através do qual isto acontece será explicado mais adiante.

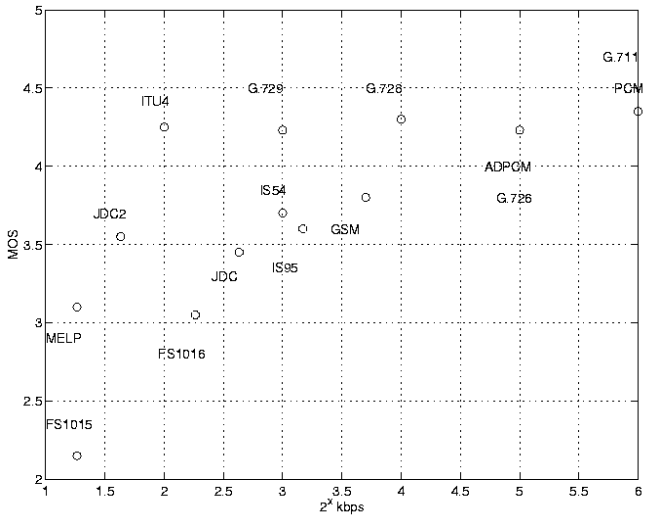

Figura 2.1 Gráfico contendo pontuações dos sistemas de codificação na escala MOS (*Mean Opinion Score*).

Isto pode ser resumido no gráfico da figura 2.1. Este é um gráfico de "qualidade subjetiva" que é baseado na escala MOS (do inglês *Mean Opinion Score* [1]). Neste gráfico, a cada taxa de codificação é associada uma nota (que pode ir de 1 a 5), que é dada por um público devidamente treinado. Neste gráfico, é possível distinguir vários padrões:

- PCM (utilizado no padrão G.711, por exemplo), com cerca de 4.3 na escala MOS: ótima qualidade, mas taxa de codificação muito alta;
- LPC (utilizado no padrão FS1015, por exemplo), com cerca de 2.1 na escala MOS: qualidade precária, mas taxa de codificação muito baixa;
- ADPCM (utilizado no padrão G.726, por exemplo), com cerca de 4.2 na escala MOS: ótima qualidade, mas taxa de codificação ainda muito alta;
- MELP, com cerca de 3.1 na escala MOS: qualidade mediana, mas taxa de codificação baixa;
- CELP (utilizado nos padrões G.729 e GSM, por exemplo), com cerca de 4 na escala MOS: boa qualidade e taxa de codificação relativamente baixa;

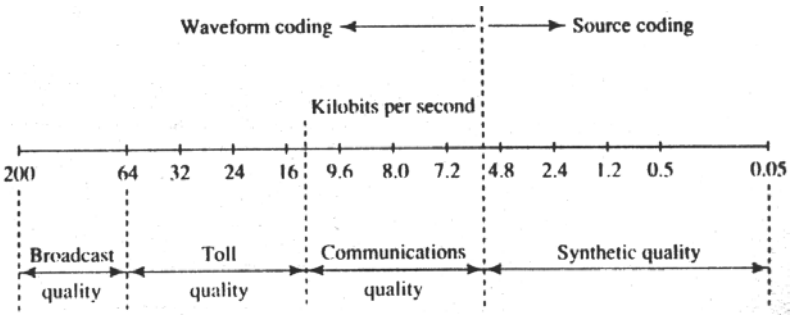

Figura 2.2 Uma interpretação "verbal" das notas contidas na escala MOS.

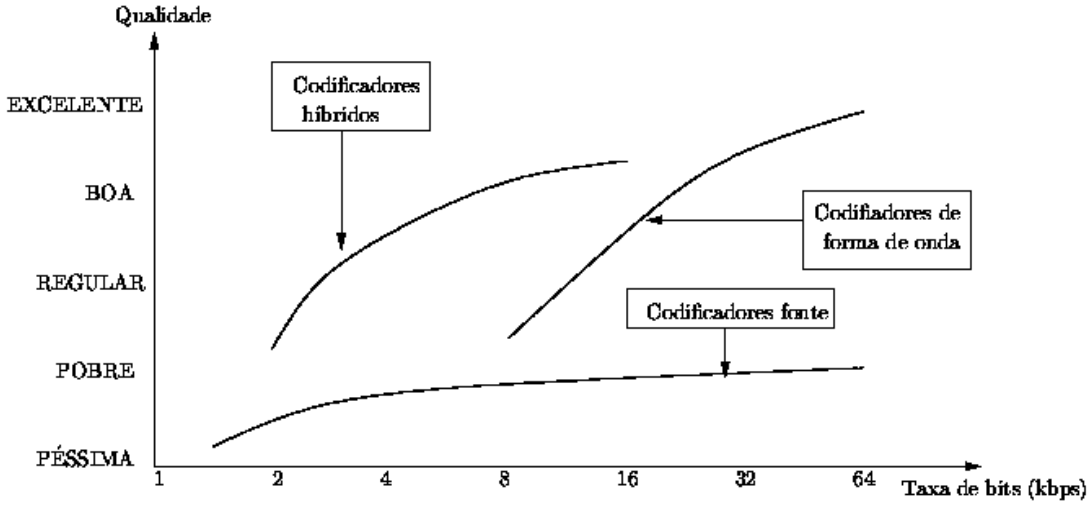

Figura 2.3 Os conjuntos dos pontos pertencentes a cada tipo de codificador.

Para melhor compreensão deste gráfico, pode-se recorrer à figura 2.2 [4], onde há uma associação entre a taxa de codificação e uma opinião expressa em palavras. É também possível ver no gráfico da figura 2.3 [1] os pontos que pertencem a cada tipo de codificador. Assim, é possível perceber que os codificadores de forma de onda tendem a apresentar uma melhor qualidade à medida que se aumenta a taxa de codificação (lembrando-se que este tipo de codificador necessita de, pelo menos, 8 kbits/s de taxa de codificação). Além disso, os codificadores de fonte, para taxas bastante baixas, seguem a mesma tendência. Contudo, quando se chega a cerca de 3 kbits/s, a melhora na qualidade já não é mais tão significativa. Os codificadores híbridos são capazes de apresentar uma

qualidade comparável com a dos codificadores de forma de onda e a taxa de codificação comparável com a dos codificadores de fonte.

2.2 Codificadores de Forma de Onda

Os codificadores que estão nessa primeira classificação utilizam, para a codificação, propriedades do sinal de caráter temporal ou até mesmo espectral. Mas não chegam ao mérito de analisar mais profundamente particularidades sobre aquilo que o sinal está representando. Tais codificadores só se preocupam com a necessidade de reconstruir a forma da onda a ser codificada. Por este motivo, qualquer tipo de sinal pode ser codificado usando tais métodos.

Entre estes, podem-se citar: PCM, ADPCM e o padrão MPEG. Os codificadores de forma de onda apresentam boa qualidade, mas, em compensação, taxas de codificação muito elevadas. Para taxas abaixo de 16 kbits/s, a melhor escolha é selecionar um outro tipo de codificador, como os de fonte, que serão vistos a seguir.

2.2.1 A Codificação PCM

PCM vem do inglês *Pulse Code Modulation***.** Esta é a forma mais básica de se codificar digitalmente um sinal qualquer. Seu objetivo é codificar cada amostra do sinal individualmente associando cada instante de tempo a um nível discreto de amplitude. O número de níveis é descrito como a *n*-ésima potência de dois, onde n é o número de bits reservado para a quantização.

Dessa forma, se foram reservados 8 bits para a quantização, isso significa que existem $2^8 = 64$ níveis discretos. Assim, cada amostra é representada apenas pela forma binária (considerando os 8 bits) da amplitude.

A codificação PCM é a que apresenta a melhor qualidade, tendo uma nota na escala MOS de cerca de 4.3. Entretanto, possui uma taxa de codificação extremamente alta.

Em telefonia digital fixa, utiliza-se uma taxa de amostragem de 8 kHz e uma resolução de 8 bits. Isso nos leva a uma taxa de codificação de 8000 Hz x 8 bits = 64 kbits/s. Para fins de transmissão, tanto numa rede de telefonia celular quanto via *web*, essa taxa é proibitivamente elevada, nos obrigando a buscar novas alternativas.

Esse tipo de codificação apresenta uma qualidade muito boa quando usado para tratar sinais de voz. Essa taxa é definida como uma referência para fins de comparação com outras taxas. Se, por exemplo, estamos falando de ADPCM para telefonia, cuja taxa de codificação é de 32 kbits/s, dizemos que apresenta uma compressão de 2:1.

2.3 Codificadores de Fonte ou Paramétricos

Os codificadores que estão nessa segunda classificação utilizam, para a codificação, peculiaridades sobre a fonte do sinal que está se representando. Isto é, se, no caso, o objetivo é codificar voz, será feito um estudo detalhado tanto do trato vocal como da natureza da voz humana propriamente dita. Fazem uso não de aspectos tão "visíveis" como aqueles usados pela PCM, mas sim de aspectos intrínsecos à voz, como será mostrado nessa seção. Tais codificadores, em oposição à classificação anterior, são, de certa forma, específicos para cada tipo de sinal.

2.3.1 A Codificação LPC

O codificador LPC (do inglês *Linear Predictive Coding*) é o principal representante dos codificadores de fonte ou paramétricos. Este codificador assume uma série de princípios em relação à voz humana.

A voz humana pode ser vista como a saída de um filtro digital, onde o mesmo é dado pelo trato vocal e a entrada é dada pelo ar que sai dos pulmões e passa pelas cordas vocais. O trato vocal é formado pela região da abertura das cordas vocais (glote), epiglote, orofaringe, garganta, língua, dentes e lábios.

Tendo em vista essas definições, pode-se classificar os sons vocais em dois grupos: sons sonoros e sons surdos. Os sons sonoros são aqueles em que as cordas vocais vibram, os quais são modelados como sendo formados por um trem de pulsos quase periódico. O período relativo a sua freqüência fundamental é geralmente chamado período de *pitch*, sendo este parâmetro muito importante para os codificadores atuais. Exemplos desses sons são os que formam os sons relativos às vogais ou a algum encontro vocálico.

Os sons surdos são aqueles em que as cordas vocais não vibram, os quais são modelados como sendo formados por ruído branco e apresentam, como principal característica, a alta taxa de cruzamentos por zero. Exemplos desses sons são os que formam os sons relativos a fonemas consonantais tais como o som de "ch" em "chá" ou o som de "s" em "sol".

Existe ainda outro tipo de som que seria uma mistura dessas duas classificações, como, por exemplo, os sons ditos plosivos ou bilabiais. Tais sons têm como excitação um simples pulso de ar. Como exemplos destes sons, podemos citar os sons das consoantes "p" e "b". Enquanto "p" é um som plosivo surdo, "b" é um som plosivo sonoro. Tal distinção pode ser confirmada se falarmos palavras que contenham essas letras.

A voz é um sinal altamente não periódico e não estacionário [4]. Contudo, se for considerado um intervalo extremamente pequeno de tempo (de 10 a 30 ms [4]), a voz pode ser considerada estacionária por partes. Para estes segmentos, pode-se, então, modelar o processo de geração da voz usando a representação dada na figura 2.4.

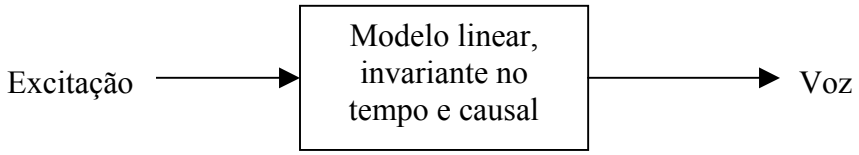

Figura 2.4 Modelando o trato vocal e o processo de geração da voz.

O sistema de codificação LPC faz uso da chamada “Análise LPC”. No modelo visto na figura 2.4, seria um filtro *all-pole*, ou seja, que contém apenas pólos,

determinado pela técnica de regressão linear, descrita mais adiante. Essa análise é feita para cada janela aplicada ao sinal de voz.

Nesse codificador, denominado *Vocoder LPC*, para se produzir um som sonoro, a excitação usada é um trem de pulsos cujo período é igual ao período de *pitch* da voz que serve de entrada para o sistema. Para um som surdo, a excitação é uma seqüência de ruído branco. Além disso, deve ser estimado o ganho para o modelo.

Desse modo, ao invés de se transmitirem todas as amostras quantizadas, transmitem-se os coeficientes calculados pela análise LPC, um *flag* para indicar se o som é sonoro ou surdo e o ganho do modelo. Se o som for sonoro, é ainda necessário transmitir o *pitch*. Assim, é possível transmitir uma quantidade muito menor de parâmetros, resultando em uma taxa de codificação acentuadamente menor. Considerando janelas de 20 ms, para o PCM com taxa de amostragem de 8 kHz, seriam necessários enviar 160 valores enquanto que, para o codificador LPC, seriam necessários, se assumirmos 10 coeficientes, apenas 13 valores.

Tudo isso é demonstrado no esquema contido na figura 2.5. Uma desvantagem clara deste sistema é o fato de a excitação ser exclusivamente sonora ou surda, impedindo uma espécie de meio termo, o que faz com que a voz resultante apresente um aspecto "robótico".

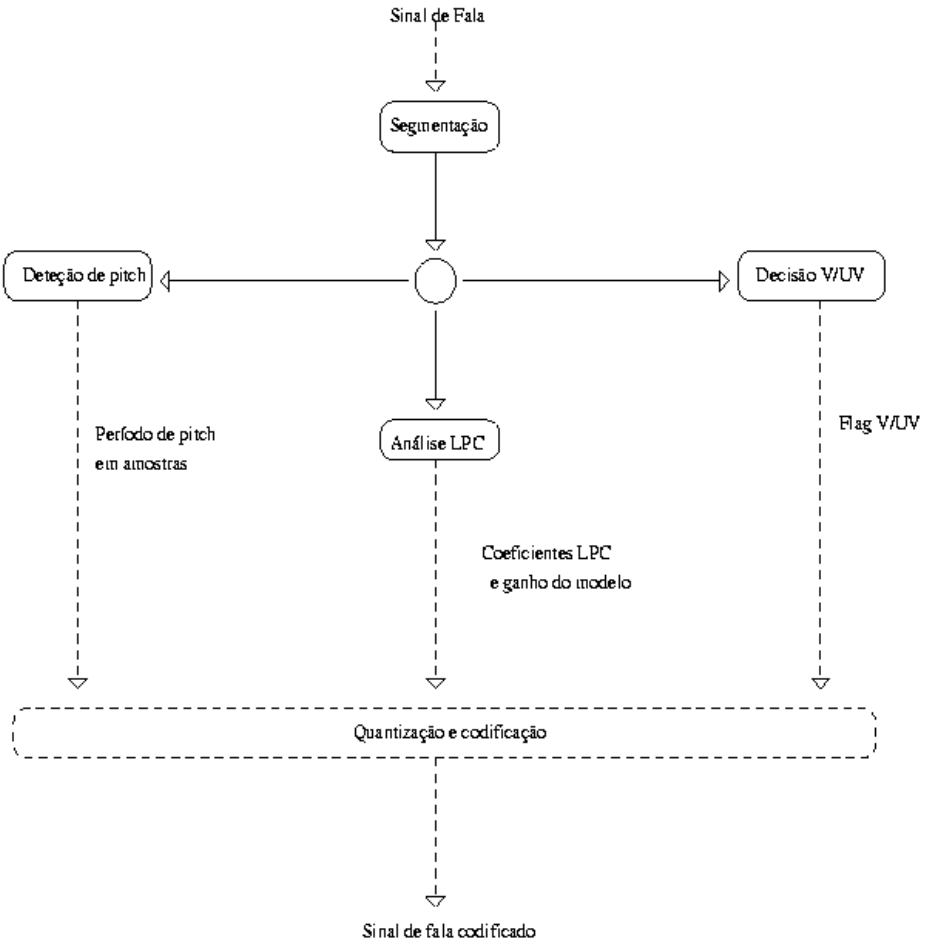

Figura 2.5 Esquema do codificador LPC.

2.4 Codificadores Híbridos - O Sistema CELP

Este tipo de codificador é a união de características dos dois tipos mencionados anteriormente, no sentido de que se mantém a parametrização LPC ao mesmo tempo que se determina a excitação pelo formato da onda. Através da determinação da melhor excitação com o auxílio de dicionários, é possível que se alcancem taxas de codificação bem reduzidas mantendo uma qualidade perfeitamente aceitável.

O sistema CELP, que vem do inglês *Code Excited Linear Prediction*, é um dos principais representantes deste tipo de codificador.

Esse sistema utiliza como base o sistema LPC, isto é, faz uso de todos os princípios apresentados na seção 2.3.1 para a redução do número de parâmetros a serem transmitidos. Entretanto, o CELP explora aquela que talvez seja a principal desvantagem do LPC, a questão da excitação utilizada.

O CELP utiliza dicionários para manipular essas excitações (isso será mais amplamente explicado nas próximas seções). Dessa maneira, é possível se obter um número bem mais amplo de excitações, o que torna a voz reconstituída ao final da transmissão bem mais natural que com o sistema LPC básico.

2.4.1 O Janelamento do Sinal de Voz

Como foi colocado acima, devido à natureza altamente não estacionária da voz humana, é preciso "janelar" o sinal para que apenas um intervalo bem reduzido seja considerado. Essa janela possui, para este trabalho, 20 ms, o que, a uma taxa de amostragem de 8 kHz, representa 160 amostras. O tipo de janela utilizado neste trabalho é a janela de *Hamming*, descrita por [3]:

$$w_H(n) = \begin{cases} \alpha + (1-\alpha)\cos\left(\dfrac{2\pi n}{M}\right) & \text{para } |n| \le \dfrac{M}{2} \\ 0 & \text{para } |n| > \dfrac{M}{2} \end{cases} \qquad (2.1)$$

onde α é igual a 0,54. Por razões a serem explicadas posteriormente, tais blocos de 20 ms são, então, divididos em 4 sub-blocos de 5 ms cada.

2.4.2 O Modelo do Trato Vocal

Através da análise LPC, são calculados, para cada bloco de 20 ms, os coeficientes do filtro digital que melhor modela o trato vocal do usuário no instante relativo àquele determinado bloco.

Isso é feito da seguinte forma [4]:

1. Calcula-se a matriz de auto-correlação do bloco do sinal de voz:

$$R_S = \begin{bmatrix} r_S(0) & r_S(1) & r_S(2) & \dots & r_S(M-1) \\ r_S(1) & r_S(0) & r_S(1) & \dots & r_S(M-2) \\ r_S(2) & r_S(1) & r_S(0) & \dots & r_S(M-3) \\ \vdots & \vdots & \vdots & & \vdots \\ r_S(M-1) & r_S(M-2) & r_S(M-3) & \dots & r_S(0) \end{bmatrix} \tag{2.2}$$

2. Monta-se a seguinte equação, onde â é o vetor dos coeficientes que se deseja calcular:

$$R_S \times \begin{bmatrix} \hat{a}(1) \\ \hat{a}(2) \\ \hat{a}(3) \\ \vdots \\ \hat{a}(M) \end{bmatrix} = \begin{bmatrix} r_S(1) \\ r_S(2) \\ r_S(3) \\ \vdots \\ r_S(M) \end{bmatrix} \tag{2.3}$$

A equação 2.3 pode ser descrita de forma mais simplificada por:

$$R_S \hat{a} = r_S \tag{2.4}$$

onde â é o vetor dos coeficientes de predição linear denotado por $[\hat{a}(1), \hat{a}(2), ..., \hat{a}(M)]^T$ e r_s é definida como $[r_S(1), r_S(2), ..., r_S(M)]^T$.

3. Esse sistema (representado pela equação 2.4) é, então resolvido pelo método recursivo de Levinson-Durbin, também descrito em [4].

O algoritmo recursivo de Levinson-Durbin aplicado à janela $n \in [m-N+1, m]$:

A inicialização: $l = 0$

$\xi^0(m)$ = energia total escalada relativa ao erro de um preditor de ordem 0

= energia média na janela do sinal de voz $f(n; m) = s(n)w(m-n)$

$= r_s(0; m)$

Recursão: Para $l = 1, 2, ..., M$, fazer o seguinte:

1. Calcular o l-ésimo coeficiente de reflexão:

$$k(l;m) = \frac{1}{\xi^{l-1}(m)} \left\{ r_s(l;m) - \sum_{i=1}^{l-1} \hat{a}^{l-1}(i;m) r_s(l-1;m) \right\} \tag{2.5}$$

2. Gerar o conjunto de ordem l de coeficientes preditores:

$$\hat{a}^{-1}(l;\, m) = k(l;\, m) \tag{2.6}$$

$$\hat{a}^{-1}(i;\, m) = \hat{a}^{l-1}(i;\, m) - k(l;\, m)\hat{a}^{l-1}(l\text{-}1;\, m),\ i=1,2,\ldots,\ l\text{-}1 \tag{2.7}$$

3. Calcular a energia do erro associado à solução de ordem *l*:

$$\xi^{1}(\mathrm{m}) = \xi^{1-1}(\mathrm{m})\left\{1 - \left[\mathrm{k}(\mathrm{l};\mathrm{m})\right]^{2}\right\} \tag{2.8}$$

4. Retornar ao passo 1 com *l* substituído por *l* + 1 se *l* < *M*.

2.4.3 Filtro de Síntese

Na predição linear, o filtro de síntese H(z) é da seguinte forma:

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 - \sum_{i=1}^{p} a_i\, z^{-i}} \tag{2.9}$$

A constante p representa a ordem do modelo LPC e denota a precisão com a qual os efeitos do trato vocal são modelados por H(z). Geralmente os codificadores CELP utilizam p=10, de forma a obter um compromisso entre qualidade e taxa de bits (ou de codificação), [1]. Os coeficientes $\{a_1, a_2, \ldots, a_p\}$ são os coeficientes de predição linear (CPLs ou em inglês *LPCs*), obtidos através da análise LPC.

O filtro H(z) representa os efeitos do trato vocal e é responsável pela introdução das correlações de curto termo.

2.4.4 Filtro de Ponderação ou Perceptivo

Foi feito um estudo, de acordo com [1], onde foi verificado que erros e ruídos em componentes de menor amplitude fazem mais diferença ao ouvido humano do que em componentes de maior amplitude. Como conseqüência, os componentes de menor amplitude são mais importantes no cálculo do erro contido no processo de análise por síntese.

A função do filtro perceptivo é enfatizar os componentes de menor amplitude e fazer o contrário com os componentes de maior amplitude. Este filtro é denotado aqui por W(z) e possui a seguinte forma:

$$W(z) = \frac{A(z)}{A\left(\frac{z}{\gamma}\right)} \tag{2.10}$$

onde γ é o fator de ponderação, que possui o valor típico de 0.8. Para ilustrar o efeito do filtro perceptivo, pode-se consultar a figura 2.6.

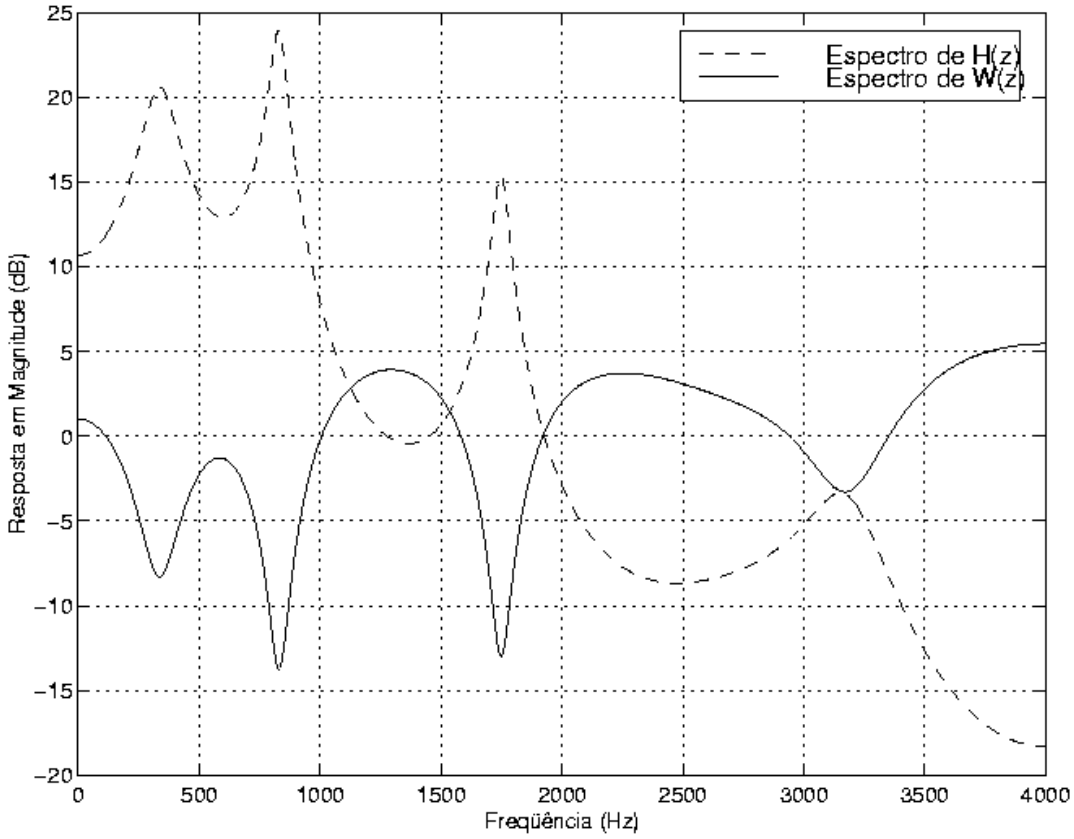

Figura 2.6 O efeito do filtro perceptivo.

Contudo, o que é realmente usado no sistema CELP deste trabalho é uma aglutinação do filtro de síntese com o filtro perceptivo. Isso é feito da seguinte forma:

$$H(z).W(z) = \frac{1}{A(z)} . \frac{A(z)}{A\left(\frac{z}{\gamma}\right)} = \frac{1}{A\left(\frac{z}{\gamma}\right)} \tag{2.11}$$

Usa-se esta versão do filtro de síntese modificado pelo filtro perceptivo para acelerar o processamento.

2.4.5 Dicionário Fixo

Um dicionário (ou, em inglês, *codebook*) é um conjunto de excitações. Este tipo de dicionário é, a princípio, da seguinte forma:

$$C_f = \{\{X_{f0}(n)\}, \{X_{f1}(n)\}, ..., \{X_{fK-1}(n)\}\} \tag{2.12}$$

onde cada termo é uma seqüência de um processo estocástico gaussiano de média zero (o que pode ser criado com o comando *randn* do Matlab). Esta representação também indica que o dicionário armazena K_f seqüências $X_{fI}(n)$, onde I é o índice da mesma. As amostras destas seqüências sofrem, então, um processo de *clipping* para simplificar os cálculos. Isto significa que as amostras que apresentarem amplitude com módulo menor que um certo valor serão zeradas. Este valor, segundo [2], é de 1,645.

Para preencher o dicionário fixo, utilizou-se o seguinte código de Matlab:

```
% FUNCTION FOR CREATING A FIXED CODEBOOK
function create = fc_create(size)
% FC_CREATE - Creates a binary file containing a fixed codebook which size is determined
%               in the argument. It is for use in the CELP codec.

       name = strcat('codebook', int2str(size), '.dat');

       n = 40;
       codebook = randn(1, size*n);

       for k = 1:size*n,
           if (abs(codebook(k))<1.645),
               codebook(k) = 0;
```

```
        end;
    end;


    fid = fopen(name, 'w');
    fwrite(fid, codebook, 'float');
    fs = fclose(fid);
```

Listagem 2.1 Geração do dicionário fixo.

2.4.6 Dicionário Adaptativo

Este tipo de dicionário é, a princípio, da seguinte forma:

$$C_a = \{\{X_{a0}(n)\}, \{X_{a1}(n)\}, ..., \{X_{aK_{a-1}}(n)\}\} \tag{2.13}$$

onde pode-se ver que o dicionário armazena K_a seqüências $X_{aI}(n)$, onde I é o índice da mesma. Tal conjunto de seqüências é constantemente atualizado, o que pode ser confirmado pela realimentação presente no diagrama de blocos da figura 2.6. Essa atualização é realizada com base na soma das melhores excitações de ambos os dicionários. Inicialmente, todas as amostras são zeradas.

2.4.7 Cálculo do Ganho dos Dicionários

Os ganhos dos dicionários, tanto do fixo como do adaptativo, devem ser calculados de acordo com a equação 2.14:

$$G = \frac{Corr_{Sinal-Alvo,Resposta-Dic}}{Corr_{Resposta-Dic,Resposta-Dic}} \tag{2.14}$$

onde $Corr_{Sinal\text{-}Alvo,\ Resposta\text{-}Dic}$ é a correlação entre o sinal-alvo e a resposta contida no dicionário em questão, enquanto que $Corr_{Resposta\text{-}Dic,\ Resposta\text{-}Dic}$ é a autocorrelação da resposta citada. O sinal alvo é o sinal de voz do qual deve ser feita a análise por síntese.

2.4.8 A Análise por Síntese

A excitação que deverá ser usada para reconstituir a voz do usuário na saída do sistema será determinada através de um processo denominado "Análise por Síntese". Esse processo é realizado para cada sub-bloco de 5 ms, pois a excitação varia mais rapidamente do que o trato vocal propriamente dito [1].

A análise por síntese se dá do seguinte modo:

1. Para cada sub-bloco de 5 ms, cada excitação contida no dicionário é submetida ao filtro calculado pela análise LPC, gerando uma resposta;
2. Essa resposta, então, é subtraída do sinal de voz presente na entrada do sistema;
3. A diferença resultante é, então, guardada;
4. A excitação que gerar a menor diferença será aquela escolhida para reconstruir o sinal de voz na saída do sistema.

O nome deste processo pode ser justificado pelo fato de ser necessário sintetizar o sinal de voz várias vezes para que se possa realizar a análise para determinar a melhor excitação.

2.4.9 O Sistema CELP Propriamente Dito

O sistema pode ser visualizado como um todo no diagrama de blocos da figura 2.7. Nesta figura, existem marcações para auxiliar a identificação de cada ponto que consta nesta descrição. Um detalhe importante é que este esquema relata o que ocorre após o janelamento, a análise LPC e a divisão em sub-blocos.

1. O sub-bloco do sinal de voz entra no sistema;
2. O sub-bloco do sinal de voz é filtrado pelo filtro de ponderação;
3. A resposta à entrada zero relativa ao sub-bloco anterior é subtraída da saída do filtro de ponderação, formando o denominado "sinal-alvo" (ou, em inglês, *target signal*). A resposta à entrada zero pode ser entendida como um

resquício da resposta do filtro de síntese que aparece na saída do filtro no momento em que a excitação não está mais presente;

4. O processo de análise por síntese tem início com a procura no dicionário adaptativo;
5. Tendo em mãos o índice da excitação do dicionário adaptativo, deve-se calcular o ganho relativo a esta excitação;
6. Multiplica-se esta excitação pelo ganho calculado;
7. Filtra-se a excitação atual (multiplicada pelo ganho calculado) pelo filtro de síntese alterado pelo filtro de ponderação, gerando-se o sinal estimado;
8. Subtrai-se o sinal estimado do sinal-alvo, gerando-se um sinal de erro;
9. A excitação que gerar o EMQ (erro médio quadrático) mínimo será a excitação ótima. Seu índice e o ganho relativo a esta serão guardados;
10. Atualiza-se o sinal-alvo, subtraindo do mesmo a melhor excitação do dicionário adaptativo (multiplicada pelo devido ganho) submetida ao filtro de síntese alterado pelo filtro de ponderação;
11. Realiza-se a busca no dicionário fixo já considerando o novo sinal-alvo (atualizado);
12. Tendo em mãos o índice da excitação do dicionário fixo, deve-se calcular o ganho relativo a esta excitação;
13. Multiplica-se esta excitação pelo ganho calculado;
14. Filtra-se a excitação atual (multiplicada pelo ganho calculado) pelo filtro de síntese alterado pelo filtro de ponderação, gerando-se o sinal estimado;
15. Subtrai-se o sinal estimado do sinal-alvo original, gerando-se um sinal de erro;
16. A excitação que gerar o EMQ mínimo será a excitação ótima. Seu índice e o ganho relativo a esta serão guardados;
17. Obtém-se a excitação completa somando-se as excitações de ambos os dicionários multiplicadas pelos respectivos ganhos;
18. Atualiza-se o dicionário adaptativo colocando-se ao fim do mesmo a resposta ótima completa.

Para a decodificação, utiliza-se um algoritmo semelhante, mas sem as buscas nos dicionários, visto que tais buscas já haviam sido feitas no processo de codificação. O único trabalho que o decodificador terá será de buscar a excitação completa, multiplicá-la pelo ganho, filtrá-la com o filtro de síntese e atualizar o dicionário adaptativo. Todos esses parâmetros (índices, ganhos e coeficientes) virão como produto do codificador.

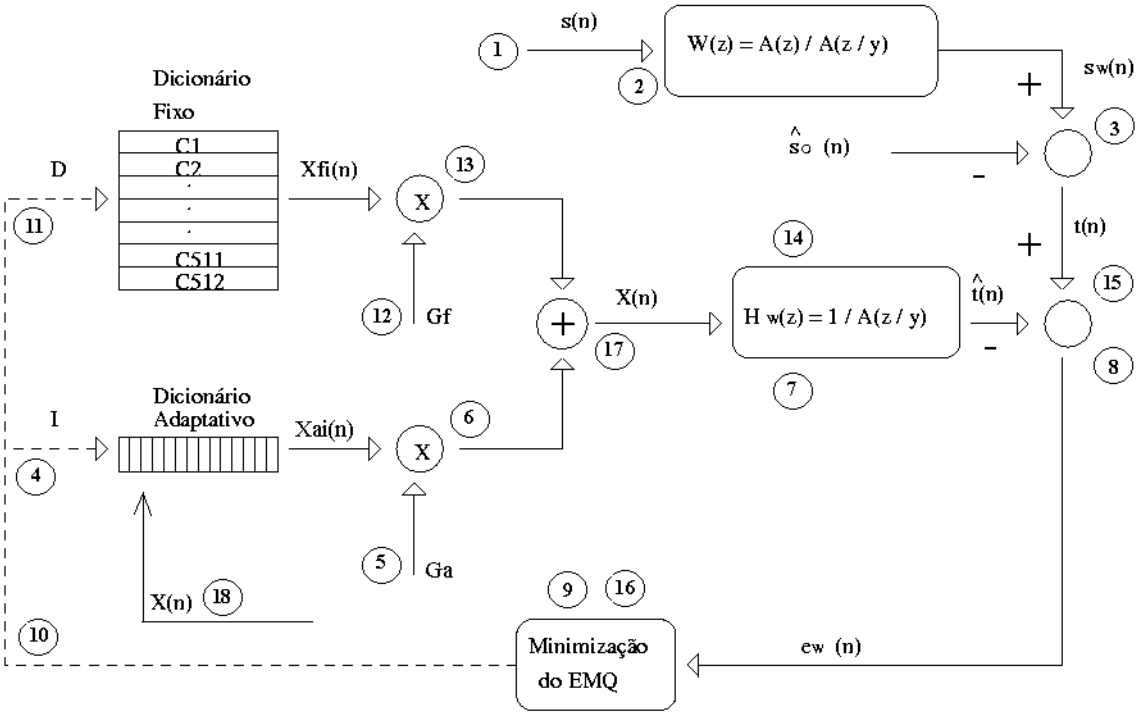

Figura 2.7 Esquema do sistema CELP

2.5 Melhorias para o Sistema CELP

2.5.1 Formas Alternativas para os Coeficientes do Filtro de Síntese

Os coeficientes LPC são extremamente sensíveis a erros de quantização. Por isso, serão usados, neste trabalho, os coeficientes LSF (do inglês, *Line Spectral Frequency*). Para o cálculo dos LSF, deve-se seguir a seguinte regra:

1. Calculam-se os polinômios P(z) (que é simétrico) e Q(z) (que é anti-simétrico) a partir de A(z), da seguinte forma:

$$P(z) = A(z) + z^{-p-1}A(z^{-1}) \tag{2.15}$$

$$Q(z) = A(z) - z^{-p-1}A(z^{-1}) \tag{2.16}$$

onde p é o número de coeficientes preditores.

2. Obtêm-se os polinômios $P_1(z)$, que é definido como sendo P(z) sem a raiz em –1, e $Q_1(z)$, que é definido como sendo Q(z) sem a raiz em +1:

$$P_1(z) = \frac{P(z)}{1+z^{-1}} \text{ e } Q_1(z) = \frac{Q(z)}{1-z^{-1}} \text{ para p par,}$$

$$P_1(z) = P(z) \text{ e } Q_1(z) = \frac{Q(z)}{1-z^{-2}} \text{ para p ímpar,} \tag{2.17}$$

3. Os polinômios $P_1(z)$ e $Q_1(z)$ são simétricos de ordem par. Como as raízes ocorrem em pares de números complexos conjugados, somente metade delas precisa ser determinada. Assim, para p par, p/2 raízes de $P_1(z)$ mais p/2 raízes de $Q_1(z)$, totalizando p, podem representar os polinômios P(z) e Q(z), e, conseqüentemente, o filtro 1/A(z), isto é, o filtro de síntese. Como as p raízes estão sobre o círculo unitário, somente os ângulos (ou freqüências) são necessários para representar A(z). Tais freqüências são as LSF.

Contudo, o que são transmitidas são as diferenças entre os coeficientes LSF, denominadas DLSF. Estas tendem a ter uma amplitude reduzida quando comparada com os LSF e, portanto, são melhores para se quantizar. Em [1], foi usada uma distribuição de *bits* para quantização como mostrado na tabela 2.1.

Quantizador	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**	**Total**
QDLSF-32	4	4	3	3	3	3	3	3	3	3	32

Tabela 2.1 Distribuição de bits para a quantização das QDLSF.

2.5.2 Interpolação dos Coeficientes

Os coeficientes do filtro de síntese são calculados para cada bloco, enquanto que as excitações são calculadas para cada sub-bloco (lembrando que existem 4 sub-blocos para cada bloco). Contudo, sabemos que o trato vocal se move de forma contínua assumindo infinitas posições, o que está muito longe do que foi descrito até aqui.

Para, então, suavizar a movimentação do trato vocal, o que se faz é realizar uma estimação dos coeficientes do filtro de síntese para cada sub-bloco. Isso é feito interpolando-se os coeficientes do bloco atual com os coeficientes do bloco anterior.

Essa interpolação é feita da seguinte forma:

$$w_i^n = (1 - q_n).w_i^a + q_n.w_i^c \tag{2.18}$$

onde w_i^n são os coeficientes do n-ésimo sub-bloco, w_i^a são os coeficientes do bloco anterior, w_i^c são os coeficientes do bloco corrente e $q_n = \{0{,}25;\ 0{,}5;\ 0{,}75;\ 1\}$.

2.5.3 Alterações

Para um processamento mais rápido, algumas modificações na estrutura do codificador podem ser feitas:

- Tanto na busca do dicionário fixo quanto do adaptativo, cada excitação tem seu ganho calculado, multiplicado e é finalmente filtrada pelo filtro de síntese alterado pelo filtro de ponderação. O que se pode fazer para agilizar o processo é filtrar de uma vez todo o dicionário e realizar a busca já com as amostras filtradas. No caso do dicionário fixo, por exemplo, só se teria que filtrar uma vez, visto que é fixo, ao invés de filtrar para todas as excitações de todos os sub-blocos, ganhando-se bastante em tempo de processamento.
- Os dicionários passam a ter uma estrutura de um grande vetor-linha. No fixo, a excitação passa a ser extraída da seguinte forma: as amostras da primeira excitação são, por exemplo, as de 0 a 39, as amostras da segunda excitação seriam as de 2 a 41, depois as de 4 a 43 e assim por diante, isto é, há um indexador do

dicionário que incrementado de 2 para cada excitação. No dicionário adaptativo esse indexador é incrementado de 1. Isso é fruto de resultados experimentais descritos em [4].

2.6 Funcionamento do Sistema CELP com as Melhorias Propostas

O sistema pode ser visualizado como um todo no diagrama de blocos da figura 2.8. Nesta figura, existem marcações para auxiliar a identificação de cada ponto que consta nesta descrição. Nesta figura há uma chave. Esta não existe realmente no programa desenvolvido. Seu uso é apenas justificado pela simplicidade que trará ao esquema. Um detalhe importante é que este esquema relata o que ocorre após o janelamento, a análise LPC e a divisão em sub-blocos.

1. O sub-bloco do sinal de voz entra no sistema;
2. O sub-bloco do sinal de voz é filtrado pelo filtro de ponderação;
3. A resposta à entrada zero relativa ao sub-bloco anterior é subtraída da saída do filtro de ponderação, formando o denominado “sinal-alvo” (ou, em inglês, *target signal*);
4. Convertem-se os coeficientes LPC, já calculados, para coeficientes LSF;
5. Interpolam-se os coeficientes LSF, usando-se os coeficientes do bloco atual e do bloco anterior. Dessa forma, tem-se uma estimativa de como está o trato vocal para cada sub-bloco, ao invés de para cada bloco. Em seguida, convertem-se, para cada sub-bloco, os coeficientes LSF interpolados de volta para coeficientes LPC, para que se possa gerar o filtro de síntese;
6. Convertem-se os coeficientes LSF relativos a cada bloco para coeficientes DLSF para que se possa transmiti-los;
7. O processo de análise por síntese tem início com a procura no dicionário adaptativo. Deve-se lembrar que o dicionário é integralmente submetido ao filtro de síntese modificado pelo filtro perceptivo para que se possa reduzir o tempo de processamento. Assim, ao invés de armazenar seqüências de excitações, os dicionários conterão respostas ao filtro de síntese;

8. Tendo em mãos o índice da seqüência do dicionário adaptativo, deve-se calcular o ganho relativo a esta resposta ao filtro de síntese;
9. Multiplica-se esta resposta pelo ganho calculado, gerando-se o sinal estimado. Devido à propriedade da linearidade [3], isso é equivalente a multiplicar a excitação pelo ganho e depois submetê-la ao filtro;
10. Coloca-se a chave A/F na posição A;
11. Subtrai-se o sinal estimado do sinal-alvo, gerando-se um sinal de erro;
12. A seqüência que gerar o EMQ mínimo será a seqüência ótima. Seu índice e o ganho relativo a esta serão guardados;
13. Atualiza-se o sinal-alvo, subtraindo-se do mesmo a melhor seqüência do dicionário adaptativo (multiplicada pelo devido ganho);
14. Realiza-se a busca no dicionário fixo, já considerando o novo sinal alvo (atualizado). Deve-se lembrar que o dicionário é integralmente submetido ao filtro de síntese modificado pelo filtro perceptivo para que se possa reduzir o tempo de processamento. Assim, ao invés de armazenar seqüências de excitações, os dicionários conterão respostas ao filtro de síntese;
15. Tendo em mãos o índice da seqüência do dicionário fixo, deve-se calcular o ganho relativo a esta resposta ao filtro de síntese;
16. Multiplica-se esta resposta pelo ganho calculado, gerando-se o sinal estimado. Devido à propriedade da linearidade [3], isso é equivalente a multiplicar a excitação pelo ganho e depois submetê-la ao filtro;
17. Coloca-se a chave A/F na posição F;
18. Subtrai-se o sinal estimado do sinal-alvo original, gerando-se um sinal de erro;
19. A seqüência que gerar o EMQ mínimo será a seqüência ótima. Seu índice e o ganho relativo a esta serão guardados;
20. Obtém-se a resposta completa, somando-se as respostas de ambos os dicionários multiplicadas pelos respectivos ganhos;
21. Atualiza-se o dicionário adaptativo, colocando-se ao fim do mesmo a resposta ótima completa.

Para a decodificação, utiliza-se um algoritmo semelhante, mas sem as buscas nos dicionários, visto que tais buscas já haviam sido feitas no processo de codificação. O único trabalho que o decodificador terá será de obter a excitação completa, multiplicá-la pelo ganho, submetê-la ao filtro de síntese suavizado pelo uso de interpolação e atualizar o dicionário adaptativo. Todos esses parâmetros (índices, ganhos e coeficientes) virão com produto do codificador.

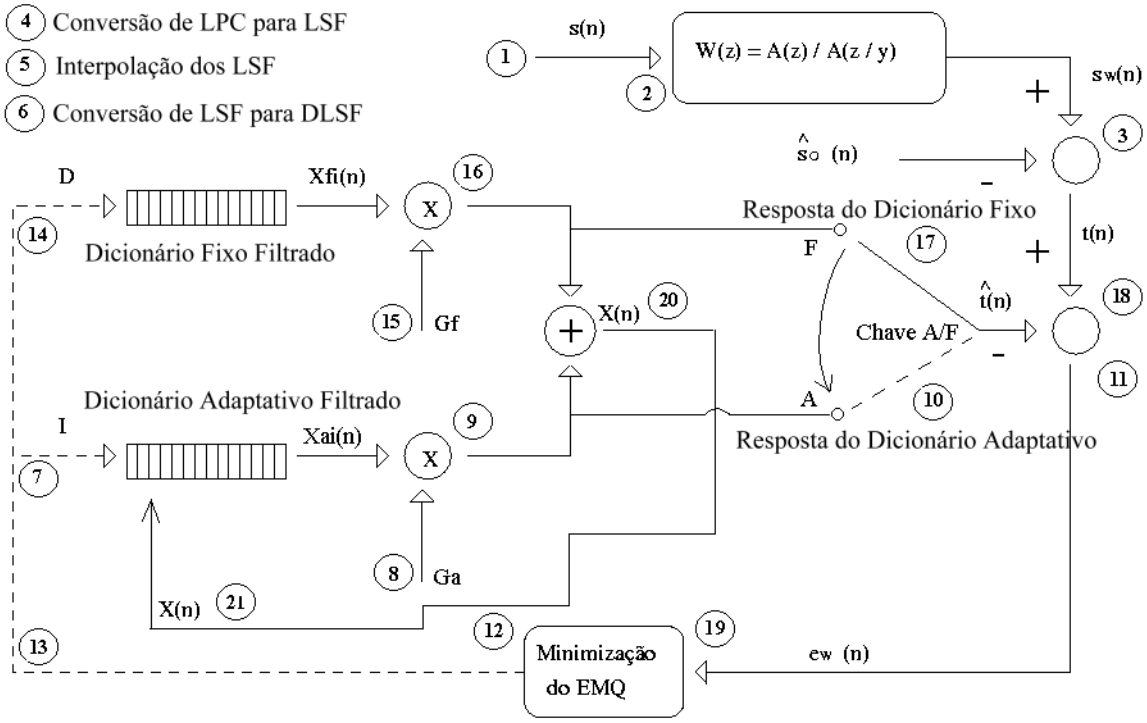

Figura 2.8 Esquema do sistema CELP.

2.7 Características e Ganhos Promovidos pelo Sistema CELP

No sistema CELP, o que é transmitido é apenas o índice da melhor excitação, o ganho do filtro e os coeficientes do filtro, ao invés de todas as amostras quantizadas, como no PCM. Por exemplo, em [2], utilizou-se a seguinte tabela de alocação de *bits*:

Parâmetro	Faixa	Número de Bits
Ganho do Dicionário Fixo (Gf)	-0,05 a 0,05	5 bits X 4
Índice do Dicionário Fixo (I)	0 a 511	9 bits X 4
Ganho do Dicionário Adaptativo (Ga)	0 a 2	4 bits X 4
Índice do Dicionário Adaptativo (L)	0 a 511	9 bits X 4
Coeficientes do Filtro de Síntese		32 bits
Total		**140 bits**

Tabela 2.2 Características do Sistema CELP.

Assim, é possível ver o ganho promovido pelo uso do sistema CELP. Se quando se usa PCM de 8 bits a 8 kHz, é necessária uma taxa de codificação de 64 kbits por segundo, usando-se o CELP, é necessária apenas uma taxa de 7 kbits por segundo.

2.8 Conclusão

Neste capítulo, foi apresentada uma visão geral dos tipos de codificação de voz usados. Foram explicados os codificadores de forma de onda, codificadores de fonte e codificadores híbridos, enfatizando o funcionamento do sistema CELP. Foram propostas melhorias ao sistema CELP apresentado, detalhando-se o funcionamento do sistema CELP já com estas melhorias. Apresentaram-se também as características do sistema e os ganhos por ele promovidos.

Capítulo 3

Bases para Implementação em Tempo Real

Este capítulo apresenta informações sobre como foi realizada a implementação em tempo real. Na Seção 3.1, é detalhado o OSS. Na Seção 3.2, é explicada a metodologia utilizada neste trabalho. Na Seção 3.3, é apresentado o diagrama de classes. Na Seção 3.4, é detalhada a classe **oss_audio**. Na Seção 3.5, é detalhada a classe **oss_wave**. Na Seção 3.6, é detalhada a classe **oss_celp**.

3.1 OSS – *Open Sound System*

O *Open Sound System* (OSS), segundo [5], é uma tentativa de se unificar a arquitetura de áudio digital para o sistema operacional UNIX. O OSS é um conjunto de *device drivers* (que são extensões do sistema operacional) que fornecem uma API (*application programming interface*) uniforme que contempla as principais arquiteturas UNIX. Apresenta suporte à *Sound Blaster* ou a placas de som compatíveis com o sistema de som do *Windows*, os quais podem ser conectados a qualquer estação UNIX, aceitando arquiteturas com barramentos ISA ou PCI. Há também suporte para máquinas com *hardware on-board* de áudio digital.

Tradicionalmente, cada distribuição UNIX fornece a própria API para processamento de áudio digital. Isso significa que aplicações escritas para uma determinada API de áudio para UNIX deveriam ser reescritas, com possível perda de funcionalidade, para outra versão do UNIX. Aplicações escritas na API OSS requerem apenas que sejam projetadas uma única vez e, depois, simplesmente recompiladas em qualquer arquitetura.

A maioria das estações fornecia, antes do OSS, apenas suporte para gravação e reprodução de áudio digital, o que é conhecido como *business audio*. O OSS apresenta, além disso, suporte a MIDI e a ambientes de manipulação de música eletrônica para a estação. Com o progresso do *audio streaming,* de síntese e reconhecimento de voz,

telefonia IP, Java e outras tecnologias multimídias, aplicações em UNIX podem fornecer as mesmas capacidades de áudio encontradas nos sistemas operacionais Windows, OS/2 e Macintosh.

3.1.1 Características do OSS

O OSS apresenta as seguintes características no campo da gravação e reprodução de áudio digital:

- Tipo de dado pode ser de 8 bits (*unsigned*) com Lei-μ ou 16 bits no formato PCM;
- Lei-A e IMA ADPCM (*Hardware* compatível com CS4321);
- Gravação e reprodução em estéreo ou mono;
- Freqüências de amostragem entre 4 kHz e 48 kHz;
- *Half-duplex* ou *full-duplex* (em *hardware* que suporta este modo);
- Apresenta suporte para acesso direto ao *buffer* de áudio via DMA:
    - Permite controlar o tempo de aplicações com bastante precisão, o que é necessário para interfaces em tempo real;
    - Apresenta um menor *overhead* de processamento, visto que a transferência de dados entre o *buffer* de aplicação e o *buffer* de DMA não é necessário;
- Capacidade de iniciar a gravação e a reprodução precisamente ao mesmo tempo (no modo *full-duplex*);
- Capacidade de sincronizar a gravação e a reprodução de áudio com a reprodução de MIDI.

3.1.2 OSS para Linux

OSS/Linux é uma implementação comercial dos *drivers* de áudio para Linux, os quais estão contidos no *Kernel* do Linux. Apresenta 100 % de compatibilidade com os

drivers do tipo *freeware* (conhecidos com OSS/Free). A API do *driver* do OSS é definido em linguagem C no arquivo de cabeçalho `<soundcard.h>`.

3.1.3 Dispositivos suportados pelo OSS

Vários *devices* (ou simplesmente dispositivos) são suportados pelo OSS. Entre eles, está o `/dev/dsp`, `/dev/mixer`, `/dev/sndstat`, `/dev/audio`, `/dev/sequencer` e o `/dev/music`.

Contudo, aquele que é utilizado neste trabalho é o `/dev/dsp`. Esse arquivo de dispositivo, juntamente com o `/dev/audio`, é o principal para uso em aplicações de voz digitalizada. Qualquer informação escrita neste dispositivo é reproduzida com os dispositivos de DAC/PCM/DSP da placa de som. A operação de leitura do dispositivo retorna a informação de áudio gravada a partir da entrada atual (o padrão para essa entrada é o microfone).

3.1.4 Comandos para o OSS

É possível realizar gravações a partir destes dispositivos usando chamadas a comandos comuns como `open`, `close`, `read` e `write`. Os parâmetros padrões dos arquivos de dispositivos (vistos anteriormente) estão selecionados para que seja possível gravar e reproduzir voz ou outro tipo de sinal com qualidade relativamente baixa.

É também possível se ajustar boa parte dos parâmetros dos dispositivos através de chamadas ao comando `ioctl`. Assim, pode-se determinar a freqüência de amostragem, o número de canais (se o sinal é mono ou estéreo), o formato através do qual o sinal será manipulado, o tamanho do bloco de leitura etc.

3.2 Metodologia

Para escrever o código do codificador CELP em tempo real, foi escolhida a Programação Orientada a Objeto (POO). Esta é um dos mais recentes passos relativos ao aumento da abstração na tecnologia de desenvolvimento de programas.

Com o aumento da abstração, têm-se as seguintes vantagens:

- Consegue-se desenvolver programas maiores e mais complexos;
- O esforço de programação torna-se mais eficiente;
- Reaproveitam-se mais facilmente componentes já desenvolvidos.

Mas, há um preço a se pagar. A programação orientada a objeto, apesar de todas as vantagens citadas, possui as seguintes desvantagens:

- O tamanho do código executável fica maior;
- A velocidade de execução dos programas geralmente fica um pouco menor.

A linguagem escolhida foi o C++, por se apresentar como uma linguagem bem difundida, com um bom suporte a POO e ser largamente utilizada na programação de chips para processamento de sinal (como DSPs). Além disso, o OSS, como explicado anteriormente, foi desenvolvido em C, que é um sub-conjunto do C++.

Uma outra preocupação que se teve ao projetar as classes foi a de programar em inglês, tornando possível que este trabalho (e todo o esforço de programação envolvido) fosse aproveitado como base para outros trabalhos em todo o mundo.

Outro detalhe importante é programar tendo em vista que outra pessoa utilizará este código e, portanto, deve-se programar, não para uma máquina, mas para uma pessoa. Assim, evitou-se utilizar recursos cujo significado pode ser mal-compreendido ou perdido. Entre estes recursos, pode-se citar os *flags*.

3.3 Diagrama de Classes

Ao desenvolver o codificador, a primeira preocupação que se tem seria o modo através do qual seriam realizadas a interface e a manipulação da placa de som. Assim,

seguindo o paradigma da POO e da componentização, vê-se a necessidade de encapsular essas funcionalidades dentro de uma classe genérica (ou de uso genérico). Para isso, foram criadas as classes **oss_audio** e **oss_wave**. É através delas que todo o resto do programa acessa a placa de som, tanto para gravação quanto para reprodução e eles serão mais detalhadas adiante.

A seguir, foi criada uma classe que implementa especificamente as funcionalidades presentes nos códigos contidos em [1, 2] que formam o codificador CELP. Essa classe, por esse motivo, foi denominada **oss_celp**.

Aqui, nota-se a diferenciação feita entre um problema genérico e um problema específico. Um desenvolvedor que se preocupe com POO deve ter sempre em mente esse conceito. É muito comum que um problema possa ser dividido em uma componente genérica (no caso, as componentes de manipulação da placa de som) e uma componente específica (no caso, a classe que implementa o codificador em si).

Dessa forma, pode-se resumir o que foi introduzido nesta seção e o que será explicado nas próximas seções na figura 3.1.

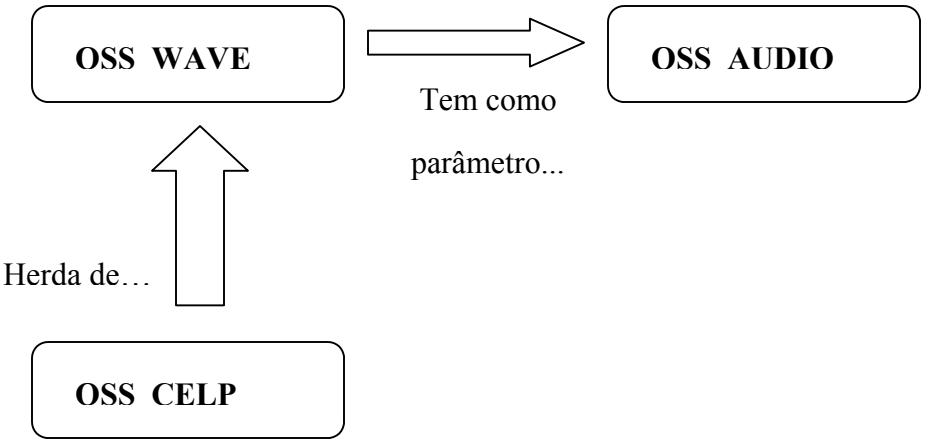

Figura 3.1 Diagrama de Classes.

3.4 A Classe oss_audio

Essa classe tem como objetivo ser uma interface entre a classe **oss_wave** e o OSS, que é o sistema de manipulação da placa de som. Códigos que utilizam esta classe requerem a inclusão do arquivo de cabeçalho **oss_wave.h**.

3.4.1 Métodos

Valor de Retorno	Nome do Método
-	oss_audio
void	Init
int	GetSampleRate
int	GetNumberOfChannels
int	GetAudioDevice
void	Close

Tabela 3.1 Métodos da classe oss_áudio.

A) Método oss_audio, o Construtor da Classe

Esse método é o construtor da classe e, por este motivo, tem a função de inicializar seus atributos. Este método tem como parâmetros a freqüência de amostragem e o número de canais referentes ao sinal que será manipulado por um objeto desta classe. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
```

Exemplo 3.1 Uso do construtor da classe oss_audio.

B) Método Init

Esse método tem como objetivo inicializar o *driver* de áudio. Além de validar os valores dos atributos que foram definidos no construtor, determinará também outras propriedades do *driver* que são, de certa forma, transparentes ao programador, como o formato da amostra, que, no caso, é o formato de 16 bits no modo *little endian*. Este método não possui argumentos. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
```

Exemplo 3.2 Uso do método Init.

C) Método GetSampleRate

Esse método tem como objetivo retornar o valor determinado pelo construtor como sendo a freqüência de amostragem. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
cout << "Frequencia de amostragem: " << AudioDriver.GetSampleRate() << endl;
```

Exemplo 3.3 Uso do método GetSampleRate.

D) Método GetNumberOfChannels

Esse método tem como objetivo retornar o valor determinado pelo construtor como sendo o número de canais, ou seja, se o sinal será manipulado em "mono" (um canal) ou "estéreo" (dois canais). Assim como o método anterior, não possui argumentos. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
cout << "Numero de canais: " << AudioDriver.GetNumberOfChannels() << endl;
```

Exemplo 3.4 Uso do método GetNumberOfChannels.

E) Método GetAudioDevice

Esse método tem como objetivo retornar o valor determinado pelo método Init como sendo o número de identificação do dispositivo de áudio. Esse número é gerado internamente no método Init quando o dispositivo é aberto, através do comando *open*. Assim como o método anterior, não possui argumentos. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
cout << "ID do dispositivo de audio: " << AudioDriver.GetAudioDevice() << endl;
```

Exemplo 3.5 Uso do método GetAudioDevice.

F) Método Close

Esse método tem como objetivo encerrar o uso do dispositivo de áudio inicializado pelo método Init. Assim como seu método opositor, não possui argumentos. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
cout << "Frequencia de amostragem: " << AudioDriver.GetSampleRate() << endl;
cout << "Numero de canais: " << AudioDriver.GetNumberOfChannels() << endl;
cout << "ID do dispositivo de audio: " << AudioDriver.GetAudioDevice() << endl;
AudioDriver.Close();
```

Exemplo 3.6 Uso do método Close.

3.5 A Classe oss_wave

Essa classe tem como objetivo manipular um vetor de amostras de áudio, apresentando a capacidade de gravar, tocar e concatenar vetores, entre outras. Códigos que utilizam esta classe requerem a inclusão do arquivo de cabeçalho **oss_wave.h**

3.5.1 Métodos

Valor de Retorno	Nome do Método
-	oss_wave
int	Record
int	Play

void	SetLength
int	GetLength
void	SetData
short	GetData
int	GetSampleRate
int	GetNumberOfChannels
void	Append
void	Equals
void	SaveMatFile

Tabela 3.2 Métodos da classe oss_wave.

A) Método oss_wave, o Construtor da Classe

Esse método é o construtor da classe e, por este motivo, tem a função de inicializar seus atributos. Este método tem como parâmetros a freqüência de amostragem e o número de canais referentes ao sinal que será manipulado por um objeto desta classe. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
AudioDriver.Close();
```

Exemplo 3.7 Uso do construtor da classe oss_wave.

B) Método Record

Esse método tem a função de gravar um vetor de áudio. Apresenta como argumentos o objeto do tipo **oss_audio** que servirá de interface com o OSS e o tempo (medido em segundos) durante o qual a gravação será realizada. Daí, pode ser calcular que o vetor terá tantas amostras quanto o produto entre a freqüência de amostragem e o tempo de gravação (por canal). A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
AudioDriver.Close();
```

Exemplo 3.8 Uso do método Record.

C) Método Play

Esse método tem a função de reproduzir um vetor de áudio. Apresenta como argumento o objeto do tipo **oss_audio** que servirá de interface com o OSS. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
	{
		cout << "Erro na gravacao!" << endl;
		exit(1);
	}
if (!AudioSignal.Play(AudioDriver))
	{
		cout << "Erro na reproducao!" << endl;
		exit(1);
	}
AudioDriver.Close();
```

Exemplo 3.9 Uso do método Play.

D) Métodos SetLength e GetLength

Esses métodos têm as funções de, respectivamente, determinar e retornar o valor do comprimento do vetor de áudio. A seguir, pode ser visto um exemplo do uso do método GetLength:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
	{
```

```
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
cout << "Comprimento do vetor: " << AudioSignal.GetLength() << endl;
AudioDriver.Close();
```

Exemplo 3.10 Uso do método GetLength.

E) Métodos SetData e GetData

Esses métodos têm as funções de, respectivamente, determinar e retornar o valor de alguma amostra do vetor de áudio. A seguir, pode ser visto um exemplo do uso do método GetData:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
for(int i=0; i<10; i++)
        cout << "V[" << i << "]=" << AudioSignal.GetData(i) << endl;
AudioDriver.Close();
```

Exemplo 3.10 Uso do método GetData.

F) Métodos GetSampleRate e GetNumberOfChannels

Esses métodos têm as funções de, respectivamente, retornar o valor da freqüência de amostragem e número de canais da mesma forma que acontece com a classe **oss_audio**. A seguir, pode ser visto um exemplo do uso destes métodos:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
cout << "Frequencia de amostragem: " << AudioSignal.GetSampleRate() << endl;
cout << "Numero de canais: " << AudioSignal.GetNumberOfChannels() << endl;
AudioDriver.Close();
```

Exemplo 3.11 Uso dos métodos GetSampleRate e GetNumberOfChannels.

G) Método Append

Esse método tem a função de concatenar dois vetores de áudio contidos em dois objetos da classe **oss_wave**. Apresenta como argumentos o objeto do tipo **oss_wave** será concatenado com o objeto (também da classe **oss_**wave em questão). Além da concatenação, esse método ainda realiza a atualização do valor do comprimento do vetor de áudio (internamente através da função SetLength). A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
```

```
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
oss_wave AudioSignal2(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
if (!AudioSignal2.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
}
AudioSignal2.Append(AudioSignal);
AudioDriver.Close();
```

Exemplo 3.12 Uso do método Append.

H) Método Equals

Esse método tem a função de igualar dois vetores de áudio contidos em dois objetos da classe **oss_wave**. Apresenta como argumentos o objeto do tipo **oss_wave** será igualado com o objeto (também da classe **oss_wave** em questão). Além disso, esse método ainda realiza a atualização do valor do comprimento do vetor de áudio (internamente através da função SetLength). A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);

AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
oss_wave AudioSignal2(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
```

```
if (!AudioSignal.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
AudioSignal2.Equals(AudioSignal);
if (!AudioSignal2.Play(AudioDriver))
        {
                cout << "Erro na reproducao!" << endl;
                exit(1);
        }

AudioDriver.Close();
```

Exemplo 3.13 Uso do método Equals.

I) Método SaveMatFile

Esse método tem a função de gerar um arquivo de extensão ".mat" que pode ser lido pelo Matlab para uma eventual análise ou manipulação matemática de qualquer natureza. A seguir, pode ser visto um exemplo do uso deste método:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 10;

oss_audio AudioDriver(Frequencia, Canais);
AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

if (!AudioSignal.Record(AudioDriver, Tempo))
        {
                cout << "Erro na gravacao!" << endl;
                exit(1);
        }
```

```
AudioSignal.SaveMatFile();
AudioDriver.Close();
```

Exemplo 3.14 Uso do método SaveMatFile.

No Matlab, esse arquivo representa uma *workspace* e pode ser recuperada através do seguinte comando:

```
save -ASCII oss_wave
```

Esse comando criará uma variável que será um vetor igual ao vetor contido no objeto da classe **oss_wave** em questão. Tal variável terá o nome de **oss_wave** e o vetor de áudio contido nela poderá ser reproduzido através do comando **soundsc.**

3.5.2 Funcionamento *on-line*

Para se testar o funcionamento *on-line*, pode-se usar as classes descritas até aqui da forma como é mostrado no exemplo 3.14. Aqui é feito um teste de *talk-through*. Isto denota um teste no qual a voz do usuário é reproduzida na saída do sistema copiando-se a entrada para a saída sem nenhum tipo de processamento:

```
int Frequencia = 8000;
int Canais = 1;
double Tempo = 0.02;

oss_audio AudioDriver(Frequencia, Canais);

AudioDriver.Init();
oss_wave AudioSignal(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());
oss_wave AudioSignal2(AudioDriver.GetSampleRate(), AudioDriver.GetNumberOfChannels());

while(1)
{
        if (!AudioSignal.Record(AudioDriver, Tempo))
                {
```

```
                        cout << "Erro na gravacao!" << endl;
                        exit(1);
                }
        AudioSignal2.Equals(AudioSignal);
        if (!AudioSignal2.Play(AudioDriver))
                {
                        cout << "Erro na reproducao!" << endl;
                        exit(1);
                }
}
AudioDriver.Close();
```

Exemplo 3.14 Uso do funcionamento *on line*.

Neste código, percebe-se a presença de um *loop* infinito. Neste *loop*, o que acontece é que os dois objetos da classe **oss_wave** atuam da seguinte forma:

- O objeto **AudioSignal** recebe o sinal de áudio preenchendo seu próprio vetor;
- Os valores das amostras contidas no vetor do objeto **AudioSignal** são transferidas para o objeto **AudioSignal2**;
- O objeto **AudioSignal2** envia suas amostras para serem reproduzidas pela placa de som do computador.

O funcionamento em tempo real é possível pois o OSS apresenta suporte ao modo *full-duplex* e, além disso, apresenta rotinas de baixo nível que possibilitam a gravação e a reprodução simultâneas, da forma que está descrita no exemplo 3.14.

3.6 A Classe oss_celp

Foi proposta, no início deste capítulo, uma idéia de que a maioria dos problemas pode ser dividida em duas partes: uma genérica e uma específica. Até aqui, falou-se da parte genérica do problema da codificação CELP em tempo real. Essa parte seria representada pelas duas classes explicadas nas duas últimas seções. Essas classes fazem o papel de manipular a placa de som e o vetor de áudio, além de possibilitar o funcionamento em tempo real.

Seguindo esta teoria, é necessário, agora, explicar a parte específica do problema, que seria a codificação em si. Os métodos presentes nesta classe são também bem específicos, ficando claro que seria bem difícil sua utilização fora do contexto da codificação CELP. É claro que sua utilização fica bem acessível para aplicações derivadas do sistema CELP.

Mesmo assim, com o uso da POO, fica bem mais nítido o funcionamento do sistema CELP, pois, desse modo, o código se aproxima de uma linguagem humana.

Dessa forma, então, foi criada a classe **oss_celp** que herda da classe **oss_wave** todos os seus métodos e atributos.

3.6.1 Métodos

Devido ao grande número de métodos contidos nesta classe, estes serão divididos nos seguintes grupos:

- Construtor
- Métodos de "*Set*" e "*Get*";
- Métodos para conversões;
- Métodos próprios para o sistema CELP
- Métodos para filtragem
- Métodos para *debug*

3.6.1.1 Construtor

O construtor da classe é o método **oss_celp**. Este é bem semelhante ao construtor da classe da qual herda suas propriedades, isto é, da **oss_wave**. Assim, tal método necessita, como argumentos, da freqüência de amostragem e do número de canais.

3.6.1.2 Métodos de "*Set*" e "*Get*"

Métodos de "*Set*"		**Métodos de "*Set*"**	
Valor de Retorno	**Nome do Método**	**Valor de Retorno**	**Nome do Método**
void	SetNumCoef(int)	int	GetNumCoef(void)
void	SetBlockSize(int)	int	GetBlockSize(void)
void	SetFCSize(int)	int	GetFCSize(void)
void	SetACSize(int)	int	GetACSize(void)
void	SetNumberOfSamples_AC(int)	int	GetNumberOfSamples_AC(void)
void	SetWindowSize(int)	int	GetWindowSize(void)
void	SetGamma(double)	double	GetGamma(void)
void	SetPercFilter(void)	double	GetPercFilter(int, int)
void	SetVoiceSubBlock(int)	-	-
void	SetPerc_ZeroInputResponse(void)	-	-
void	SetTargetSignal(void)	double	GetTargetSignal(int)
void	SetCompleteResponse(int)	-	-
void	SetDecodedVoiceSubBlock(int)	double	GetDecodedVoiceSubBlock(int)
-	-	double	GetCoefLPC(int)
-	-	double	GetCoefLSF(int)
-	-	double	GetCoefLSF_before(int)
-	-	double	GetCoefLPC_from_LSF(int)
-	-	double	GetCoefDLSF(int)

-	-	double	GetCoefLSF_Interpolated(int)
-	-	double	GetCoefLPC_from_LSF_Interpolated(int)
-	-	int	GetIndexAC(int)
-	-	double	GetGainAC(int)
-	-	int	GetIndexFC(int)
-	-	double	GetGainFC(int)

Tabela 3.3 Métodos de "*Set*" e "*Get*" da classe oss_celp.

A) Métodos SetNumCoef e GetNumCoef

Tais métodos têm como objetivo, respectivamente, determinar e retornar o número de coeficientes utilizados. Esse número será o número de coeficientes LPC, LSF, DLSF e seus derivados, como, por exemplo, os coeficientes LSF interpolados. O tipo de argumento do método *set* e o tipo de retorno do método *get* são inteiros.

B) Métodos SetBlockSize e GetBlockSize

Tais métodos têm como objetivo, respectivamente, determinar e retornar o comprimento do bloco de voz utilizado, medido em amostras. O método de *get* também é utilizado para manipular o tamanho do sub-bloco de voz, bastando dividir este valor pelo número de sub-blocos por bloco. O tipo de argumento do método *set* e o tipo de retorno do método *get* são inteiros.

C) Métodos SetFCSize e GetFCSize

Tais métodos têm como objetivo, respectivamente, determinar e retornar o tamanho do dicionário fixo utilizado. O termo "FC" denota o dicionário fixo, que, em inglês, é denominado *fixed codebook*. O tipo de argumento do método *set* e o tipo do retorno de método *get* são inteiros.

D) Métodos SetACSize e GetACSize

Tais métodos têm como objetivo, respectivamente, determinar e retornar o tamanho do dicionário adaptativo utilizado. O termo "AC" denota o dicionário adaptativo, que, em inglês, é denominado *adaptive codebook*. O tipo de argumento do método *set* e o tipo de retorno do método *get* são inteiros.

E) Métodos SetNumberOfSamples_AC e GetNumberOfSamples_AC

Tais métodos têm como objetivo, respectivamente, determinar e retornar o número de amostras do dicionário adaptativo utilizado. Esse número de amostras é calculado somando-se o tamanho do dicionário adaptativo com o tamanho de um sub-bloco de voz. Isso é feito para se ter tantas seqüências de excitação quanto for o tamanho do dicionário. O tipo de argumento do método *set* e o tipo de retorno do método *get* são inteiros.

F) Métodos SetWindowSize e GetWindowSize

Tais métodos têm como objetivo, respectivamente, determinar e retornar o tamanho da janela que irá recortar o sinal de voz na entrada do sistema. No sistema apresentado neste trabalho, é utilizada uma janela de Hamming cujo tamanho é igual ao

tamanho de um bloco de voz. O tipo de argumento do método *set* e o tipo de retorno do método *get* são inteiros.

G) Métodos SetGamma e GetGamma

Tais métodos têm como objetivo, respectivamente, determinar e retornar o valor do fator de ponderação. Este valor, denotado por γ, controla o quanto os componentes de baixa amplitude do sinal de voz serão ressaltados e apresenta um valor típico de 0,8. O tipo de argumento do método *set* e o tipo de retorno do método *get* são *double*.

H) Métodos SetPercFilter e GetPercFilter

Tais métodos têm como objetivo, respectivamente, determinar e retornar o valor dos coeficientes do filtro perceptivo ou de ponderação. O tipo de argumento do método *set* é *void* visto que esse filtro é calculado baseado unicamente no fator de ponderação e nos coeficientes LPC. O tipo de retorno do método *get* é *double* e os argumentos são um identificador (inteiro) e o índice do coeficiente. Esse identificador indica se o coeficiente deve vir do numerador (identificador zero) ou do denominador (identificador um).

I) Método SetVoiceSubBlock

Tal método tem como objetivo determinar o valor das amostras do sub-bloco de voz. O argumento deste método é um inteiro, que representa o identificador do sub-bloco do sinal de voz (começando-se de um) visto que tais amostras são retiradas do vetor que representa o bloco de voz inteiro. É preciso, desta forma, saber o número deste sub-bloco dentro do bloco de voz. O tipo de retorno do método é *void*.

J) Método SetPerc_ZeroInputResponse

Tal método tem como objetivo determinar a resposta à entrada zero do filtro de síntese alterado pelo filtro perceptivo. Não apresenta nenhum argumento. Tal valor será usado, internamente, pelo método descrito na próxima seção. O tipo de retorno do método é *void*.

L) Métodos SetTargetSignal e GetTargetSignal

Tais métodos têm como objetivo, respectivamente, determinar e retornar o valor do sinal alvo. Tal valor é calculado subtraindo-se do sub-bloco de voz atual a resposta à entrada zero do filtro de síntese alterado pelo filtro perceptivo. O tipo de retorno do método *get* é *double* e seu argumento é um inteiro que representa o índice da amostra do sinal-alvo.

M) Método SetCompleteResponse

Tal método tem como objetivo determinar o valor da resposta completa, isto é, da resposta da soma das excitações de ambos os dicionários (fixo e adaptativo) quando submetida ao filtro de síntese alterado pelo filtro perceptivo. Esse valor será usado no método **UpdateAdaptiveCodebook**, que será visto mais adiante. O tipo de retorno do método é *void* e o argumento da função é o sub-bloco atual do sinal de voz, começando de um.

N) Métodos SetDecodedVoiceSubBlock e GetDecodedVoiceSubBlock

Tais métodos têm como objetivo, respectivamente, determinar e retornar o valor das amostras do sub-bloco do sinal de voz decodificado. O tipo de argumento do método *set* é *int* visto que este conjunto de amostras é calculado baseado na informação de qual é o sub-bloco atual (começando de um). O tipo de retorno do método *get* é *double* e o

argumento é um inteiro que representa o índice da amostra do sub-bloco de voz decodificado.

O) Método GetCoefLPC

Tal método tem como objetivo retornar o valor dos coeficientes LPC. A única forma de se calcular tais coeficientes é através do método **AnalisysLPC.** O tipo de retorno do método é *double* e o argumento é um inteiro que representa o índice do coeficiente, começando de zero.

P) Método GetCoefLSF

Tal método tem como objetivo retornar o valor dos coeficientes LSF. A única forma de se calcular tais coeficientes é através do método **CoefLPC_to_CoefLSF.** O tipo de retorno do método é *double* e o argumento é um inteiro que representa o índice do coeficiente, começando de zero.

Q) Método GetCoefLSF_before

Tal método tem como objetivo retornar o valor dos coeficientes LSF referentes ao bloco anterior. A única forma de se determinar tais coeficientes é através do método **SaveCoefLSF_before.** Esses valores são utilizados para a interpolação dos coeficientes LSF, o que é feito pelo método **InterpolateCoefLSF**. O tipo de retorno do método é *double* e o argumento é um inteiro que representa o índice do coeficiente, começando de zero.

R) Método GetCoefLPC_from_LSF

Tal método tem como objetivo retornar o valor dos coeficientes LPC que foram calculados a partir de coeficientes LSF. A única forma de se calcular tais coeficientes é através do método **CoefLSF_to_CoefLPC.** O tipo de retorno do método é *double* e o argumento é um inteiro que representa o índice do coeficiente, começando de zero.

S) Método GetCoefDLSF

Tal método tem como objetivo retornar o valor dos coeficientes DLSF que foram calculados a partir de coeficientes LSF. A única forma de se calcular tais coeficientes é através do método **QuantLSF.** O tipo de retorno do método é *double* e o argumento é um inteiro que representa o índice do coeficiente, começando de zero.

3.6.1.3 Métodos para Conversões

Valor de Retorno	Nome do Método
void	CoefLPC_to_CoefLSF(void)
void	CoefLSF_to_CoefLPC(void)
void	CoefLSF_Interpolated_to_CoefLPC(void)

Tabela 3.4 Métodos para conversões da classe oss_celp.

A) Método CoefLPC_to_CoefLSF

Tal método tem como objetivo converter o valor dos coeficientes LPC que foram obtidos através do método **AnalisysLPC** para valores LSF**.** O tipo de retorno do método é *void* e não há argumentos.

B) Método CoefLSF_to_CoefLPC

Tal método tem como objetivo converter o valor dos coeficientes LSF (que foram obtidos através do método descrito na seção anterior) em coeficientes LPC para o fim de definir o numerador e o denominador do filtro de síntese, bem como do filtro perceptivo**.** O tipo de retorno do método é *void* e não há argumentos.

C) Método CoefLSF_Interpolated_to_CoefLPC

Tal método tem como objetivo converter o valor dos coeficientes LSF (que foram obtidos através do método descrito na seção anterior e, em seguida, interpolados) em coeficientes LPC para o fim de definir o numerador e o denominador do filtro de síntese, bem como do filtro perceptivo**.** O tipo de retorno do método é *void* e não há argumentos.

3.6.1.4 Métodos Próprios para o Sistema CELP

Valor de Retorno	Nome do Método
void	AnalysisLPC(void)
void	SaveCoefLSF_before(void)
void	InterpolateCoefLSF(int)
void	QuantLSF(void)
void	Search_AC(int)
void	Search_FC(int)
void	UpdateTargetSignal(int)
void	UpdateAdaptiveCodebook(void)

void	ReceiveDecodedVoiceSubBlock(oss_celp)
void	Zero_SynthFilter_States(void)
void	`ReceiveCoefInfo(oss_celp)`
void	`ReceiveCodebookInfo(oss_celp, int)`

Tabela 3.5 Métodos próprios para o sistema CELP da classe oss_celp.

A) Método AnalysisLPC

Este método tem como objetivo realizar a análise LPC de um bloco do sinal de voz já "janelado" e gerar os coeficientes LPC que emulam o trato vocal durante tal bloco. O tipo de retorno deste método é *void* e não há argumentos.

B) Método SaveCoefLSF_before

Este método tem como objetivo armazenar os valores dos coeficientes LSF em um vetor para que, no próximo bloco, os valores do bloco anterior ainda estejam acessíveis de modo a ser possível executar a interpolação dos coeficientes. O tipo de retorno deste método é *void* e não há argumentos.

C) Método InterpolateCoefLSF

Este método tem como objetivo interpolar os coeficientes LSF do bloco atual com os do bloco anterior (salvos pelo método anterior). Dessa forma, como foi explicado anteriormente, haverá uma estimação da posição do trato vocal para cada sub-bloco, ao invés de existir apenas uma estimação a cada bloco. O tipo de retorno deste método é *void* e o argumento da função é o sub-bloco atual do sinal de voz, começando de um.

D) Método QuantLSF

Esse método tem como objetivo quantizar os coeficientes LSF e preencher o vetor relativo aos coeficientes DLSF. O tipo de retorno é *void* e não há argumentos.

E) Método Search_AC e Search_FC

Esses métodos têm como objetivo realizar a busca, respectivamente, nos dicionários adaptativo e fixo, além de determinar os valores do índice da melhor excitação e do ganho. Cada dicionário apresenta um atributo para o índice e um atributo para o ganho, sendo que cada um desses atributos é um vetor de quatro posições. Isso é desta forma para que se possa reservar uma posição para cada sub-bloco. Por esta razão, deve-se informar, através do argumento do método (que é um inteiro), que sub-bloco do sinal de voz é o atual (começando de um). O tipo de retorno deste método é *void*.

F) Método UpdateTargetSignal

Esse método tem como objetivo atualizar o sinal alvo após a busca no dicionário adaptativo. Para isso, este método realiza, internamente, a subtração entre o sinal alvo atual e a resposta da melhor excitação do dicionário adaptativo multiplicada pelo ganho do dicionário adaptativo quando submetida ao filtro de síntese alterado pelo filtro perceptivo. O tipo de retorno deste método é *void* e o argumento é o número do sub-bloco do sinal de voz (começando de um).

G) Método UpdateAdaptiveCodebook

Esse método tem como objetivo atualizar o dicionário adaptativo com a resposta completa obtida através do método **SetCompleteResponse**. O tipo de retorno deste método é *void* e não há argumentos.

H) Método ReceiveDecodedVoiceSubBlock

Esse método tem como objetivo fazer com que o objeto da classe **oss_celp** em questão receba de outro objeto desta mesma classe o vetor contendo o sub-bloco de voz decodificado e, com esta informação, possa preencher seu vetor de áudio para que seja possível sua reprodução. O argumento deste método é o objeto do qual se deseja extrair o sub-bloco de voz decodificado e o tipo de retorno é *void*.

I) Método Zero_SynthFilter_States

Esse método tem como objetivo zerar o *buffer* que armazena os estados do filtro de síntese. O tipo de retorno é *void* e não há argumentos.

J) Método ReceiveCoefInfo

Esse método tem como objetivo atribuir os valores dos coeficientes do objeto argumento aos coeficientes do objeto em questão. O tipo de retorno é *void* e o argumento é o objeto da classe oss_celp do qual os coeficientes são oriundos.

L) Método ReceiveCodebookInfo

Esse método tem como objetivo atribuir os valores dos índices e ganhos do objeto argumento aos índices e ganhos do objeto em questão. O tipo de retorno é *void* e o argumento é o objeto da classe oss_celp do qual os índices e ganhos são oriundos e o número do sub-bloco atual.

3.6.1.5 Métodos para filtragem

Valor de Retorno	Nome do Método
void	ApplyWindow(void)
void	ApplySynthFilter_to_FC(void)
void	ApplyPercFilter_to_VoiceSubBlock(void)
void	ApplySynthFilter_to_AC(void)

Tabela 3.6 Métodos para filtragem da classe oss_celp.

A) Método ApplyWindow

Esse método tem como objetivo aplicar a janela de Hamming ao bloco do sinal de voz para que, então, possa ser realizada a análise LPC. O tipo de retorno deste método é *void* e não há argumentos.

B) Método ApplySynthFilter_to_FC

Esse método tem como objetivo submeter todo o dicionário fixo ao filtro de síntese alterado pelo filtro perceptivo. Isso é feito para que seja possível realizar uma busca mais rápida neste tipo de dicionário. O tipo de retorno deste método é *void* e não há argumentos.

C) Método ApplyPercFilter_to_VoiceSubBlock

Esse método tem como objetivo submeter o sub-bloco do sinal de voz ao filtro perceptivo. O tipo de retorno deste método é *void* e não há argumentos.

D) Método ApplySynthFilter_to_AC

Esse método tem como objetivo submeter todo o dicionário adaptativo ao filtro de síntese alterado pelo filtro perceptivo. O tipo de retorno deste método é *void* e não há argumentos.

3.6.1.6 Métodos para *Debug*

Valor de Retorno	Nome do Método
void	ShowCoefLPC(void)
void	ShowCoefLSF(void)
void	ShowCoefDLSF(void)
void	ShowCoefLPC_from_LSF(void)
void	ShowCoefLSF_before(void)
void	ShowCoefLSF_Interpolated(void)
void	ShowCoefLPC_from_LSF_Interpolated(void)
void	ShowPercFilter(void)
void	ShowTargetSignal(void)
void	ShowIndexAC(int)
void	ShowGainAC(int)
void	ShowIndexFC(int)
void	ShowGainFC(int)
void	ShowDecodedVoiceSubBlock(void)

Tabela 3.7 Métodos para *debug* da classe oss_celp.

Esses métodos têm como objetivo imprimir no monitor os valores aos quais são relativos para que se possa conferir o correto funcionamento do sistema em toda a sua extensão. O único detalhe importante está nos métodos relativos a índices e ganhos dos dicionários. O argumento desses métodos é um inteiro que representa o identificador do sub-bloco do sinal de voz, começando de um.

3.7 Conclusão

Este capítulo apresentou informações sobre como foi realizada a implementação em tempo real. Foram detalhados o OSS, a metodologia utilizada neste trabalho e o diagrama de classes. Em seguida, foram detalhadas as classes que atuaram como a base para o sistema: **oss_audio**, **oss_wave** e **oss_celp**.

Capítulo 4

Funcionamento do Sistema CELP e Testes de Performance

Neste capítulo serão apresentados os métodos conhecidos para avaliação de sistemas de codificação de voz na Seção 4.1. Na Seção 4.2, é detalhada a classe **oss_quality** que é usada para realizar estimações da qualidade do sistema. Na Seção 4.3, são apresentadas as características do sistema. Na Seção 4.4, é apresentada a interface com o usuário. Na Seção 4.5, é explicado o algoritmo do sistema CELP implementado. Na Seção 4.6, são apresentados os dados dos resultados dos experimentos e na Seção 4.7 são feitas análises sobre esses dados.

4.1 Avaliação do sistema

Três métodos de avaliação são propostos:

- Razão Sinal / Ruído (SNR)
- Razão Sinal / Ruído Segmentada (SNR_{seg})
- Razão Sinal / Ruído Segmentada Perceptiva ($SNR_{fw, seg}$)

4.1.1 Razão Sinal / Ruído (SNR)

A razão sinal / ruído (ou em inglês, *signal-to-noise ratio* - SNR) é uma forma de medida amplamente utilizada para sistemas de codificação. Existem diversas variações e entre elas, além da SNR clássica, estão a SNR segmentada e o SNR segmentada perceptiva (que serão explicadas mais adiante). É importante notar que as medições baseadas em SNR são adequadas somente para sistemas que têm como objetivo reproduzir a forma de onda original da entrada.

Seja *s(n)* o sinal de voz livre de ruído e *r(n)* o sinal correspondente já com o ruído introduzido, sendo que tais sinais são considerados como sendo de energia. O sinal de erro pode ser escrito como:

$$e(n) = s(n) - r(n) \tag{4.1}$$

A energia do sinal de erro pode ser calculada, então, da seguinte forma:

$$E_e = \sum_{n=-\infty}^{\infty} e^2(n) = \sum_{n=-\infty}^{\infty} [s(n) - r(n)]^2 \tag{4.2}$$

A energia contida no sinal de voz é definida como:

$$E_s = \sum_{n=-\infty}^{\infty} s^2(n) \tag{4.3}$$

A medida se SNR resultante (em dB) é obtida, então, da seguinte maneira:

$$SNR = 10.\log_{10} \frac{E_s}{E_e} = 10.\log_{10} \frac{\sum_{n=-\infty}^{\infty} s^2(n)}{\sum_{n=-\infty}^{\infty} [s(n) - r(n)]^2} \tag{4.4}$$

Assim como em outras formas de se avaliar a qualidade de um sinal processado por um codificador, o sinal original (livre de ruído) é necessário. Desse modo, esse tipo de medida só é possível quando ambos os sinais, o que representa a entrada e o que representa a saída, estão disponíveis.

A principal vantagem da medida de qualidade através deste algoritmo é a sua simplicidade, mas apresenta vários problemas, como se pode ver na seção seguinte.

4.1.2 Razão Sinal / Ruído Segmentada (SNR_{seg})

A Razão sinal / ruído clássica, explicada na seção anterior, é um estimador muito fraco da qualidade de um sinal de voz para uma grande variedade de distorções, segundo [4]. Isso é devido ao fato de que a SNR clássica não é correlacionada com nenhum atributo subjetivo da qualidade do sinal de voz. Além disso, este tipo de medida considera todos os erros na forma de onda do sinal de voz no domínio do tempo como tendo o mesmo peso. A energia do sinal é, geralmente, variante no tempo. Se for considerado que

o ruído é largamente distribuído na freqüência, com pouca flutuação de energia, a medida de SNR deverá variar de *frame* a *frame*. Uma medida de SNR falsamente alta pode ser obtida se o sinal de voz contém alta concentração de trechos sonoros, já que, como foi visto anteriormente, o ruído tem um maior efeito perceptivo em segmentos de baixa energia (como nas fricativas surdas).

Uma medida de qualidade muito mais fiel pode ser obtida se a SNR for medida através de vários períodos curtos e for calculada a média destes resultados. Essa forma de medida baseada em *frames* é denominada SNR segmentada e é formulada da seguinte maneira:

$$\mathrm{SNR}_{seg} = \frac{1}{M}\sum_{j=0}^{M-1} 10.\log_{10}\left[\frac{\sum_{n=N.j}^{(N+1).j} s^2(n)}{\sum_{n=N.j}^{(N+1).j}\left[s(n)-r(n)\right]^2}\right] \tag{4.5}$$

onde *N* é o tamanho de um *frame* em amostras. Para cada *frame*, tipicamente de 15 a 25 ms (neste trabalho foram usados 20 ms), é calculada a SNR e, por fim, através de uma operação de média aritmética entre as medidas de todos os segmentos do sinal, chega-se ao valor final de SNR segmentada.

Em alguns casos, problemas com este tipo de medida podem aparecer se *frames* de silêncio forem incluídos, devido ao fato de haver, possivelmente, valores de SNR negativos e de módulo elevado, comprometendo a média final.

4.1.3 Razão Sinal / Ruído Segmentada Perceptiva

Para resolver alguns dos maiores problemas apresentados pelas formas de medida de qualidade explicadas anteriormente, utiliza-se a Razão sinal / ruído segmentada perceptiva. Tal medida tem como objetivo considerar as diferentes componentes em freqüência do sinal como tendo importâncias diferenciadas na percepção de ruído por parte do ouvido humano. Para isso, faz uso do filtro perceptivo (discutido anteriormente).

Essa medida é formulada do seguinte modo:

$$\mathrm{SNR}_{\mathrm{seg}} = \frac{1}{\mathrm{M}} \sum_{\mathrm{j}=0}^{\mathrm{M}-1} 10.\log_{10} \left[\frac{\sum_{\mathrm{n}=\mathrm{N}.\mathrm{j}}^{(\mathrm{N}+1).\mathrm{j}} \mathrm{s}^2(\mathrm{n})}{\sum_{\mathrm{n}=\mathrm{N}.\mathrm{j}}^{(\mathrm{N}+1).\mathrm{j}} \mathrm{e}_{\mathrm{W}}(\mathrm{n})^2} \right] \tag{4.6}$$

onde o sinal de erro perceptivo $e_W(n)$ é obtido ao se filtrar o sinal de erro $e(n)$, que foi definido anteriormente, pelo filtro perceptivo $W(z)$. Esta é a única diferença em relação à SNR segmentada, contudo é isso que permite a distinção entre as componentes do sinal de voz.

4.1.4 Comparação entre os Métodos

Para se comparar os três métodos apresentados, utiliza-se uma tabela, contida em [4], onde é indicada a correlação entre as medidas de qualidade objetiva e a qualidade subjetiva. Esta última pode ser entendida como aceitabilidade do sinal ao ouvido humano, assim como a escala MOS.

Medida de Qualidade Objetiva	Coeficiente de Correlação Média
SNR	0,24
SNR segmentada	0,77
SNR segmentada perceptiva	0,93

Tabela 4.1 Comparação entre medidas de qualidade objetiva e subjetiva.

Segundo a tabela 4.1, o melhor método é a SNR segmentada perceptiva, por apresentar a maior correlação medidas de qualidade subjetiva. A utilidade de uma medida de qualidade objetiva está na sua capacidade de prever a qualidade subjetiva. Idealmente, ambas as formas de medida devem ser empregadas no momento de avaliar um sistema de codificação. Como regra geral, medidas objetivas são usadas em estágios iniciais de projeto do sistema para o ajuste de parâmetros do sinal de voz, e as medidas subjetivas são utilizadas para assegurar a aceitabilidade por parte do usuário.

Contudo, mesmo a Razão Sinal / Ruído Segmentada Perceptiva ainda é muito distante da realidade. Isso é devido ao fato de este método não levar em conta muitos princípios de psicoacústica que seriam essenciais para uma análise precisa, como, por exemplo, o mascaramento.

4.2 A Classe oss_quality

Essa classe tem como objetivo fornecer as três medidas de qualidade objetivas apresentadas na seção anterior.

4.2.1 Métodos

Valor de Retorno	Nome do Método
-	oss_quality
int	GetLength
void	SetSignal
void	SetEstimatedSignal
void	AccumulateSignalPower
void	AccumulateErrorPower
void	ZeroSignalPower
void	ZeroErrorPower
double	GetSNR
void	SetNumberOfSegments
int	GetNumberOfSegments
void	IncrementNumberOfSegments

void	AccumulateSNR_seg
double	GetSNR_seg
void	SetNumCoef
int	GetNumCoef
void	SetNumPercFilter
void	SetDenPercFilter
void	ZeroStatesPercFilter
void	AccumulatePercErrorPower
void	ZeroPercErrorPower
double	GetSNR_perc
double	GetSNR_seg_perc

Tabela 4.1 Métodos da classe oss_quality.

A) Método oss_quality, o Construtor da Classe

Esse método é o construtor da classe e, por este motivo, tem a função de inicializar seus atributos. Este método tem como parâmetro o comprimento do *frame* do sinal que será manipulado por um objeto desta classe. A seguir, pode ser visto um exemplo do uso deste método:

```
int tamanho = 160
oss_quality Medidor(tamanho);
```

Exemplo 4.1 Uso do método oss_quality.

B) Método GetLength

Tal método tem como objetivo retornar o comprimento do *frame* do sinal que deverá ser avaliado. O tipo de retorno do método é *int* e não há argumentos. Um exemplo de seu uso é mostrado a seguir:

```
int tamanho = 160
oss_quality Medidor(tamanho);


cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;
```

Exemplo 4.2 Uso do método GetLength.

C) Método SetSignal

Tal método tem como objetivo determinar as amostras do *frame* do sinal que deverá ser avaliado. Os argumentos são o índice da amostra (valor inteiro), começando de zero e o valor da amostra, cujo valor é *double*. Um exemplo de seu uso é mostrado a seguir. Neste exemplo, assumi-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;


for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
      }
  }
```

Exemplo 4.3 Uso do método SetSignal.

D) Método SetEstimatedSignal

Tal método tem como objetivo determinar as amostras do *frame* do sinal estimado que deverá ser avaliado. Os argumentos são o índice da amostra (valor inteiro), começando de zero, e o valor da amostra, cujo valor é *double*. Um exemplo de seu uso é mostrado a seguir. Neste exemplo, assumi-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal* e o sinal estimado é representado pela variável *sinal_estimado*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;

for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
        Medidor.SetEstimatedSignal(i, sinal_estimado(k* Medidor.GetLength() + i) );
      }
  }
```

Exemplo 4.4 Uso do método SetEstimatedSignal.

E) Métodos AccumulateSignalPower e AccumulateErrorPower

Tais métodos têm como objetivo acumular, respectivamente, a potência do *frame* do sinal que deverá ser avaliado e a potência do sinal de erro para que, posteriormente, seja calculada a SNR. Não há argumentos. Neste exemplo, assume-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal* e o sinal estimado é representado pela variável *sinal_estimado*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;

for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
        Medidor.SetEstimatedSignal(i, sinal_estimado(k* Medidor.GetLength() + i) );
      }
    Medidor.AccumulateSignalPower();
    Medidor.AccumulateErrorPower();
  }
```

Exemplo 4.5 Uso dos métodos AccumulateSignalPower e AccumulateErrorPower.

F) Método GetSNR

Tal método tem como objetivo calculadar a SNR. Não há argumentos. Neste exemplo, assume-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal* e o sinal estimado é representado pela variável *sinal_estimado*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;

for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
        Medidor.SetEstimatedSignal(i, sinal_estimado(k* Medidor.GetLength() + i) );
      }
    Medidor.AccumulateSignalPower();
```

```
    Medidor.AccumulateErrorPower();
  }


cout << "SNR: " << Medidor.GetSNR() << " dB" << endl;
```

Exemplo 4.6 Uso do método GetSNR.

G) Método AccumulateSNR_seg

Tal método tem como objetivo acumular a soma dos valores de SNR para que, posteriormente, seja calculada a média, e assim, a SNR segmentada. Não há argumentos.

H) Método IncrementNumberOfSegments

Tal método tem como objetivo incrementar o número de segmentos. Este número será usado no cálculo da média que produzirá o valor para a Razão sinal / ruído segmentada. Não há argumentos.

I) Método ZeroSignalPower e ZeroErrorPower

Tais métodos têm como objetivo zerar, respectivamente, a potência do *frame* do sinal que deverá ser avaliado e a potência do sinal de erro para que, posteriormente, seja calculada a SNR segmentada. Não há argumentos.

J) Método SetNumberOfSegments e GetNumberOfSegments

Tais métodos têm como objetivo, respectivamente, determinar e retornar o número de segmentos. O argumento do método de *"set"* e o valor de retorno do método de *"get"* são do tipo *int*.

L) Método GetSNR_seg

Tal método tem como objetivo calcular e retornar a SNR segmentada. Não há argumentos. A seguir, pode-se ver um exemplo no qual são mostrados os métodos explicados do item G ao item L. Neste exemplo, assume-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal* e o sinal estimado é representado pela variável *sinal_estimado*. Além disso, assume-se que os segmentos considerados são de quatro *frames*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;

for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
        Medidor.SetEstimatedSignal(i, sinal_estimado(k* Medidor.GetLength() + i) );
      }
    Medidor.AccumulateSignalPower();
    Medidor.AccumulateErrorPower();

    if ((k%4) == 0)
      {
                Medidor.AccumulateSNR_seg(Medidor.GetSNR());
                Medidor.ZeroSignalPower();
                Medidor.ZeroErrorPower();
                Medidor.IncrementNumberOfSegments();
      }
  }

cout << "SNR segmentada: " << Medidor.GetSNR_seg() << " dB" << endl;
```

Exemplo 4.7 Uso de métodos para cálculo de SNR segmentada.

M) Método SetNumCoef e GetNumCoef

Tais métodos têm como objetivo, respectivamente, determinar e retornar o número de coeficientes do filtro perceptivo utilizado no cálculo da Razão sinal / ruído segmentada perceptiva. O argumento do método de *"set"* e o valor de retorno do método de *"get"* são do tipo *int*.

N) Método SetNumPercFilter e SetDenPercFilter

Tais métodos têm como objetivo, respectivamente, determinar os coeficientes do numerador e do denominador do filtro perceptivo que será utilizado no cálculo da SNR segmentada perceptiva. Os argumentos são o índice da amostra (valor inteiro), começando de zero, e o valor da amostra, cujo valor é *double*.

O) Método ZeroStatesPercFilter

Tal método tem como objetivo zerar os estados do filtro perceptivo utilizado no cálculo da SNR segmentada perceptiva. Não há argumentos.

P) Método AccumulatePercErrorPower

Tal método tem como objetivo acumular a potência do sinal de erro submetido ao filtro perceptivo para que, posteriormente, seja calculada a SNR segmentada perceptiva. Não há argumentos.

Q) Método ZeroPercErrorPower

Tal método tem como objetivo zerar a potência do sinal de erro submetido ao filtro perceptivo para que, posteriormente, seja calculada a SNR segmentada perceptiva. Não há argumentos.

R) Método GetSNR_perc e GetSNR_seg_perc

Tais métodos têm como objetivo retornar, respectivamente, o valor da SNR perceptiva relativa a um determinado segmento e o valor da SNR segmentada perceptiva que é a média de todos os segmentos. Não há argumentos. A seguir, pode-se ver um exemplo no qual são mostrados os métodos explicados do item M ao item R. Neste exemplo, assume-se que *M* é o número de *frames* do sinal a ser avaliado, que é representado pela variável *sinal*, e o sinal estimado é representado pela variável *sinal_estimado*. Além disso, assume-se que os segmentos considerados são de quatro *frames* e os coeficientes do filtro perceptivo estão armazenados nas variáveis *NumPercFilter* e *DenPercFilter*.

```
int tamanho = 160
oss_quality Medidor(tamanho);
Medidor.SetNumCoef(10);
cout << "Comprimento do frame do sinal a ser avaliado: " << Medidor.GetLength() << endl;
cout << "Numero de coeficientes do filtro perceptivo: " << Medidor.GetNumCoef() << endl;

for(int k = 0; k < M; k++)
  {
    for(int i = 0; i < Medidor.GetLength(); i++)
      {
        Medidor.SetSignal(i, sinal(k* Medidor.GetLength() + i) );
        Medidor.SetEstimatedSignal(i, sinal_estimado(k* Medidor.GetLength() + i) );
      }

       for(int r=0; r<Medidor.GetNumCoef(); r++)
```

```
          Medidor.SetNumPercFilter(r, NumPercFilter(r));
        for(int r=0; r<Medidor.GetNumCoef(); r++)
          Medidor.SetDenPercFilter(r, DenPercFilter(r));


        Medidor.ZeroStatesPercFilter();
        Medidor.AccumulateSignalPower();
        Medidor.AccumulatePercErrorPower();


    if ((k%4) == 0)
      {
        Medidor.AccumulateSNR_seg(Medidor.GetSNR_perc());
        Medidor.ZeroSignalPower();
        Medidor.ZeroPercErrorPower();
        Medidor.IncrementNumberOfSegments();
      }
  }


cout << "SNR segmentada perceptiva: " << Medidor.GetSNR_seg_perc() << " dB" << endl;
```

Exemplo 4.8 Uso de métodos para cálculo de SNR segmentada perceptiva.

4.3 Características

O sistema contido neste trabalho apresenta as seguintes características:

- A máquina utilizada foi um Pentium III 800 MHz com 128 megas de memória RAM;
- A placa de som utilizada foi uma Sound Blaster Live;
- O microfone utilizado é do tipo dinâmico;
- O sistema operacional utilizado foi o Linux versão 8.0 da distribuição Red Hat;
- O fator γ do filtro perceptivo igual a 0,8;
- O número de coeficientes do filtro de síntese é igual a 10. Esse valor é largamente empregado por se obter um bom compromisso entre qualidade e taxa de bits, como visto em [1];
- Há a flexibilidade de se escolher os tamanhos de ambos os dicionários.

4.4 Interface com o Usuário

Para se cumprir, entre outros requisitos, as duas últimas características, desenvolveu-se uma interface com usuário que atua da seguinte forma:

1. O programa solicita o tamanho do dicionário fixo, o qual deve ser informado pelo usuário através do teclado;
2. O programa solicita o tamanho do dicionário adaptativo, o qual deve ser informado pelo usuário através do teclado;
3. O programa solicita o tempo durante o qual a codificação deverá ser executada, o qual deve ser informado pelo usuário através do teclado. Se o tempo for zero, isso indicará que o programa deverá ser executado indefinidamente. Se o tempo for diferente de zero, ao final do mesmo será informada a Razão sinal/ruído referente à codificação;

4.5 Algoritmo do Sistema CELP

Esta seção tem como objetivo descrever em detalhes o algoritmo usado neste sistema. Aqui, considera-se que o usuário já especificou o tempo de execução e o tamanho de ambos os dicionários.

1. Inicializar o *driver* de som;
2. Definir o número de coeficientes LPC e LSF;
3. Definir o tamanho do bloco de voz em amostras;
4. Definir o tamanho em amostras da janela utilizada para recortar o sinal de voz;
5. Definir o tamanho do dicionário fixo;
6. Definir o tamanho do dicionário adaptativo;

7. Definir o número de amostras do dicionário adaptativo como sendo o tamanho do dicionário adaptativo definido anteriormente somado ao tamanho de um sub-bloco de voz;
8. Definir um contador (que conta o número de blocos processados) como zero;
9. Zerar o contador que contará o número de blocos;
10. Zerar os estados do filtro de síntese;
11. Para cada bloco fazer o seguinte:
    a. Gravar a partir do microfone um bloco do sinal de voz;
    b. Aplicar uma janela de Hamming ao bloco de voz gravado;

       Isso é feito multiplicando-se o bloco do sinal de voz (ponto a ponto) pela janela de Hamming, que é definida pela equação 4.1:

$$w_H(n) = 0.54 + 0.46\cos\left(\frac{2\pi n}{M}\right) \tag{4.1}$$

       onde M é o número de pontos do bloco do sinal de voz

    c. Realizar a análise LPC no bloco de voz, já com a janela de Hamming aplicada;

       Isso é feito aplicando-se o algoritmo visto na Seção 2.4.1.2.

    d. Se o contador que conta o número de blocos não estiver em "0", salvar os coeficientes LSF relativos ao bloco anterior para um posterior cálculo para interpolação;
    e. Converter os coeficientes LPC do bloco atual para coeficientes LSF;
    f. Se o contador que conta o número de blocos não estiver em "0", fazer, para cada sub-bloco do sinal de voz, o seguinte:
        i. Definir o sub-bloco do sinal de voz;
        ii. Interpolar os coeficientes do bloco atual com os do bloco anterior para ter uma estimativa para os coeficientes do sub-bloco atual;

            Isso é feito usando-se a equação 4.2 da seguinte forma:

$$w_i^n = (1 - q_n).w_i^a + q_n.w_i^c \tag{4.2}$$

onde w_i^n são os coeficientes do n-ésimo sub-bloco, w_i^a são os coeficientes do bloco atual, w_i^c são os coeficientes do bloco corrente e $q_n = \{0{,}25; 0{,}5; 0{,}75; 1\}$.

iii. Converter os coeficientes LSF interpolados para coeficientes LPC;

iv. Definir os coeficientes do numerador e do denominador do filtro perceptivo;

Isso é feito da seguinte forma:

1. Definem-se os coeficientes de índice 0 do numerador e do denominador como sendo iguais a "1";
2. Para os coeficientes de índices 1 a N, onde N é o número total de coeficientes do filtro de síntese, faz-se o seguinte: o coeficiente do numerador de índice *i* será igual ao coeficiente LPC (oriundo da conversão dos coeficientes LSF interpolados) de índice *i-1*. O coeficiente do denominador de índice *i* será definido como a *i*-ésima potência do fator γ multiplicado pelo coeficiente do numerador do filtro perceptivo de índice *i*.

v. Aplicar o filtro perceptivo ao sub-bloco de voz;

vi. Definir o sinal-alvo;

Isso é feito subtraindo-se do sinal de voz (janelado e submetido ao filtro perceptivo) a resposta à entrada zero do filtro de síntese relativa ao sub-bloco anterior;

vii. Aplicar filtro de síntese ao dicionário adaptativo. O filtro de síntese, neste caso, é formado pelo denominador do filtro perceptivo. O numerador é, portanto, igual a um;

viii. Realizar busca no dicionário adaptativo;

Isso é feito da seguinte forma:

1. Da amostra de índice 1 até o fim do dicionário, lêem-se as N amostras (já filtradas pelo filtro de síntese), onde N é o

tamanho (em amostras) do sub-bloco do sinal de voz. Definem-se essas N amostras como sendo a seqüência candidata do dicionário adaptativo;

2. Calcula-se a correlação entre o sinal-alvo e a seqüência candidata;
3. Se esta correlação for maior do que zero, deve-se incrementar um contador que indica que houve mais uma seqüência candidata que gerou uma correlação com o sinal-alvo maior do que zero;
4. Calcula-se a autocorrelação da seqüência candidata;
5. Calcula-se o inverso do erro médio ponderado quadrático (EMPQ) segundo a equação 4.3:

$$(\mathrm{EMPQ})^{-1} = \frac{\left(\mathrm{Corr}_{\mathrm{Sinal\ \ Alvo,\ \ Sequencia\ \ Candidata}}\right)^2}{\mathrm{AutoCorr}_{\mathrm{Sequencia\ \ Candidata}}} \qquad (4.3)$$

6. Se o inverso do EMPQ for maior que o valor previamente armazenado, guarda-se o índice da seqüência candidata que o gerou e armazena-se este valor para o inverso de EMPQ para se continuar com a busca. A próxima seqüência candidata será aquela que começa na segunda amostra da seqüência candidata atual. A seqüência candidata que gerar o maior inverso para o EMPQ, isto é, o menor EMPQ, será denominada seqüência ótima;
7. Se o contador (que conta seqüências candidatas que geraram autocorrelação maior do que zero) estiver diferente de zero, deve-se ler a seqüência ótima e filtrá-la pelo filtro de síntese (alterado pelo filtro perceptivo). A saída desta filtragem será denominada resposta do dicionário adaptativo.

8. Calcula-se, então o ganho relativo ao dicionário adaptativo segundo a equação 4.4:

$$Ga = \frac{Corr_{Sinal\ \ Alvo,\ \ Resposta}}{AutoCorr_{Resposta}} \qquad (4.4)$$

9. Se o contador (que conta seqüências candidatas que geraram autocorrelação maior do que zero) estiver igual a zero, deve-se fazer o ganho relativo ao dicionário adaptativo igual a zero;

ix. Atualizar o sinal-alvo;

Isso é feito subtraindo-se do sinal-alvo atual a resposta do dicionário adaptativo (já multiplicada por seu ganho);

x. Aplicar filtro de síntese ao dicionário fixo;

xi. Realizar busca no dicionário fixo;

Isso é análogo ao que foi feito no dicionário adaptativo, com a diferença de que a seqüência candidata seguinte será aquela que começar na terceira amostra da seqüência candidata atual;

xii. Definir a resposta completa;

Isso é feito multiplicando-se a seqüência ótima do dicionário adaptativo pelo ganho do dicionário adaptativo, multiplicando-se a seqüência ótima do dicionário fixo pelo ganho do dicionário fixo, somando-se estes dois resultados e submetendo-se este resultado pelo filtro de síntese alterado pelo filtro perceptivo. Um detalhe importante é que é neste momento que se calcula a resposta do filtro de síntese à entrada zero;

xiii. Atualizar o dicionário adaptativo;

Isso é feito deslocando-se as amostras do dicionário adaptativo N amostra e, a seguir, inserindo-se a resposta completa nas últimas N amostras do dicionário adaptativo, onde N é o tamanho (em amostras) de um sub-bloco de voz;

xiv. Decodificar o sinal de voz;

Isso é feito da mesma forma que se calcula a resposta completa.

xv. Reproduzir o sinal de voz decodificado;

12. Encerrar o *driver* de som;

4.6 Aquisição de Dados

4.6.1 Metodologia

Para avaliar o sistema, foi feito o seguinte:

1. Para se obter a mesma voz para se testar o codificador, utilizou-se uma voz gravada em um CD. A voz contida nesta faixa é reproduzida sem qualquer outro som simultâneo.
2. Foi conectada apenas uma caixa de som ao reprodutor de CDs para que se pudesse ter o mesmo efeito de uma pessoa usando o sistema. Essa caixa de som foi fixada em um ponto e não mais foi removida ou deslocada.
3. Foi fixado um microfone na parte frontal da caixa de som citada acima. Tal microfone foi apontado para a caixa de som e seu posicionamento não foi mais alterado de forma alguma.
4. Para cada configuração de tamanho de dicionários, foram feitos dez testes. Os resultados destes dez testes são representados pelas dez primeiras linhas das tabelas 4.2 a 4.7. Cada teste se dava da seguinte forma:
    a. Preparava-se a faixa para ser reproduzida colocando o reprodutor de CDs em pausa;
    b. Executava-se o programa do codificador e informava-se ao mesmo os tamanhos dos dicionários, isto é, a configuração cujo efeito se desejava avaliar. Além disso, informava-se ao sistema o tempo durante o qual o sistema deveria ser executado. É claro que esse tempo era ajustado para o tempo da faixa de voz contida no CD;

c. Por fim, tira-se o reprodutor de CDs do estado em pausa ao mesmo tempo que se inicia a tarefa de codificação-decodificação-reprodução;
d. Ao fim da execução, o sistema informa os valores para a SNR clássica, para a SNR segmentada e para a SNR segmentada perceptiva;
e. Tais valores são anotados

As configurações testadas foram:

- Apenas um dicionário fixo, sendo que:
  a. O tamanho deste dicionário poderia assumir os seguintes valores: 64, 128, 256 e 380;
- Um dicionário fixo e um dicionário adaptativo, sendo que:
  a. O tamanho do dicionário fixo poderia assumir os seguintes valores: 64, 128, 256 e 380;
  b. Sendo que o tamanho do dicionário adaptativo poderia assumir os seguintes valores: 64, 128, 256, 512 e 1024;

O valor de 380 para o tamanho do dicionário fixo foi o maior valor que permitia uma execução normal por parte do sistema de codificação. Para valores superiores a este, a saída do codificador era constituída de ruído como estalos e zumbidos.

4.6.2 Apresentação dos Dados

De acordo, então, com a metodologia apresentada, foi possível montar as tabelas 4.2 a 4.7:

- O fato de o dicionário adaptativo assumir tamanho zero denota o fato de não haver dicionário adaptativo;
- Todos os valores de SNR, de todos os tipos, são medidos em dB;

- Pode-se ver nessas tabelas valores de média e desvio-padrão para as SNR de cada configuração. Isso é um abuso de cálculo já que, como tais valores estão em dB, às vezes não é muito correto realizar tais operações. Contudo, se os valores estão suficientemente próximos, é possível se ter uma idéia de como está variando a qualidade do sistema através das configurações.

Após as tabelas, são apresentados três gráficos. O primeiro contém os valores máximos de SNR segmentada perceptiva em função dos tamanhos de ambos os dicionários. O segundo contém os valores mínimos de SNR segmentada perceptiva em função dos tamanhos de ambos os dicionários. O terceiro contém os valores médios de SNR segmentada perceptiva em função dos tamanhos de ambos os dicionários.

Enquanto o primeiro gráfico dá uma idéia mais geral do desempenho do sistema, estabelecendo até onde este pode chegar, o segundo gráfico fornece uma visão do pior caso, estabelecendo uma margem de segurança, garantindo que o sistema funcione acima deste limiar na maior parte do tempo. O terceiro gráfico mostra o desempenho médio do sistema, isto é, a performance em torno do qual o sistema estará atuando.

Tamanho do Dicionário Adaptativo: zero											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
2,36	2,36	6,62	2,99	2,97	7,19	4,77	4,10	8,03	4,71	4,17	8,14
2,35	2,39	6,71	2,97	2,91	7,09	4,61	4,01	7,99	4,72	4,13	8,00
2,40	2,38	6,61	3,06	3,01	7,24	4,75	4,04	8,06	4,84	4,22	8,22
2,43	2,40	6,61	2,98	2,97	7,19	4,71	4,02	8,05	4,70	4,12	8,10
2,40	2,40	6,80	2,94	2,93	7,13	4,70	4,05	8,03	4,73	4,15	8,16
2,36	2,39	6,69	2,96	2,93	7,14	4,64	4,00	8,00	4,75	4,13	8,11
2,36	2,39	6,66	3,00	2,93	7,11	4,66	4,01	8,05	4,79	4,20	8,20
2,38	2,40	6,71	3,00	2,98	7,15	4,72	4,05	8,03	4,67	4,15	8,16
2,34	2,36	6,67	3,00	2,93	7,09	4,69	4,03	7,97	4,85	4,21	8,18
2,40	2,41	6,70	2,99	2,89	7,02	4,66	4,07	8,06	4,72	4,17	8,13
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
2,38	2,39	6,68	2,99	2,95	7,14	4,69	4,04	8,03	4,75	4,17	8,14
Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão
0,03	0,02	0,06	0,03	0,04	0,06	0,05	0,03	0,03	0,06	0,04	0,06
Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo
2,43	2,41	6,80	3,06	3,01	7,24	4,77	4,10	8,06	4,85	4,22	8,22
Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo
2,34	2,36	6,61	2,94	2,89	7,02	4,61	4,00	7,97	4,67	4,12	8,00

Tabela 4.2 Configurações com apenas um dicionário fixo.

Tamanho do Dicionário Adaptativo: 64											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
12,12	9,55	12,54	12,82	10,21	13,27	13,58	10,81	13,69	13,54	10,40	13,80
12,46	9,86	12,86	12,82	10,24	13,26	13,31	10,77	13,63	13,52	10,81	13,80
12,55	9,98	12,97	12,82	10,13	13,14	13,56	10,77	13,68	13,67	10,93	13,92
12,36	9,77	12,76	13,08	10,48	13,39	13,55	10,89	13,73	13,45	10,85	13,71
12,36	9,80	12,74	12,69	10,21	13,20	13,49	10,82	13,72	13,64	10,97	13,81
12,37	9,76	12,69	12,85	10,25	13,20	13,53	10,84	13,71	13,61	10,98	13,87
12,54	10,13	13,04	12,95	10,30	13,28	13,43	10,84	13,72	13,43	10,87	13,78
12,18	9,73	12,71	12,79	10,16	13,16	13,08	10,75	13,59	13,33	10,86	13,74
12,03	9,63	12,63	12,94	10,33	13,30	13,60	10,91	13,73	13,68	11,00	13,86
12,40	9,73	12,72	12,87	10,24	13,26	13,52	10,91	13,78	13,74	11,07	13,89
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
12,34	9,79	12,77	12,86	10,26	13,25	13,47	10,83	13,70	13,56	10,87	13,82
Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão
0,17	0,17	0,15	0,11	0,10	0,07	0,16	0,06	0,05	0,13	0,18	0,07
Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo
12,55	10,13	13,04	13,08	10,48	13,39	13,60	10,91	13,78	13,74	11,07	13,92
Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo
12,03	9,55	12,54	12,69	10,13	13,14	13,08	10,75	13,59	13,33	10,40	13,71

Tabela 4.3 Configurações com um dicionário adaptativo de tamanho 64.

Tamanho do Dicionário Adaptativo: 128											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
13,03	10,51	13,29	13,35	10,93	13,74	13,84	11,50	14,16	13,86	11,52	14,31
13,04	10,65	13,35	13,37	11,01	13,87	13,97	11,57	14,22	13,92	11,52	14,29
12,71	10,42	13,17	13,42	10,97	13,79	13,90	11,45	14,13	13,99	11,49	14,27
12,85	10,47	13,24	13,52	11,06	13,87	13,90	11,52	14,24	14,04	11,55	14,33
12,64	10,37	13,09	13,33	11,01	13,74	13,91	11,54	14,22	13,95	11,59	14,29
12,77	10,39	13,12	13,48	10,92	13,75	13,71	11,46	14,15	13,87	11,49	14,25
13,01	10,58	13,32	13,56	11,05	13,79	14,05	11,63	14,29	14,02	11,54	14,31
13,25	10,25	13,12	13,30	10,94	13,75	14,00	11,54	14,28	14,04	11,61	14,40
12,95	10,51	13,29	13,33	10,99	13,78	13,94	11,49	14,23	14,00	11,60	14,33
13,08	10,58	13,29	13,34	10,89	13,68	14,02	11,59	14,25	13,87	11,47	14,22
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
12,93	10,47	13,23	13,40	10,98	13,78	13,92	11,53	14,22	13,96	11,54	14,30
Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão	Desvio Padrão
0,19	0,12	0,09	0,09	0,06	0,06	0,10	0,06	0,05	0,07	0,05	0,05
Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo	Máximo
13,25	10,65	13,35	13,56	11,06	13,87	14,05	11,63	14,29	14,04	11,61	14,40
Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo	Mínimo
12,64	10,25	13,09	13,30	10,89	13,68	13,71	11,45	14,13	13,86	11,47	14,22

Tabela 4.4 Configurações com um dicionário adaptativo de tamanho 128.

Tamanho do Dicionário Adaptativo: 256											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
13,24	11,12	13,80	13,39	11,49	14,23	14,17	12,07	14,66	14,22	12,12	14,73
13,30	11,24	13,94	13,85	11,64	14,33	13,99	12,09	14,66	14,11	12,02	14,61
13,15	11,14	13,82	13,73	11,63	14,34	14,17	12,00	14,66	14,05	12,01	14,61
13,15	11,12	13,82	13,72	11,62	14,35	14,25	12,08	14,63	14,12	11,97	14,70
13,23	11,15	13,82	13,70	11,55	14,29	14,03	11,98	14,69	14,24	12,19	14,88
13,00	11,15	13,86	13,75	11,62	14,33	14,11	11,98	14,60	14,07	12,02	14,76
11,00	11,25	13,95	13,78	11,69	14,41	14,10	12,05	14,68	14,06	12,00	14,63
13,29	11,13	13,18	13,81	11,62	14,37	14,22	12,04	14,62	14,21	12,12	14,72
13,16	11,28	13,96	13,66	11,60	14,29	14,19	12,15	14,75	14,31	12,26	14,88
13,29	13,24	13,95	13,79	11,64	14,32	14,28	12,21	14,81	14,23	12,05	14,71
13,20	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
12,98	11,38	13,81	13,72	11,61	14,33	14,15	12,07	14,68	14,16	12,08	14,72
Desvio Padrão	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**
0,70	0,66	0,23	0,13	0,05	0,05	0,09	0,07	0,06	0,09	0,09	0,10
Máximo	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**
13,30	13,24	13,96	13,85	11,69	14,41	14,28	12,21	14,81	14,31	12,26	14,88
Mínimo	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**
11,00	11,12	13,18	13,39	11,49	14,23	13,99	11,98	14,60	14,05	11,97	14,61

Tabela 4.5 Configurações com um dicionário adaptativo de tamanho 256.

Tamanho do Dicionário Adaptativo: 512											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
13,43	11,79	14,49	14,07	12,22	14,92	14,41	12,93	15,49	14,44	12,78	15,39
13,66	11,89	14,62	13,98	12,22	14,91	14,38	12,75	15,29	14,69	13,08	15,63
13,53	11,85	14,52	13,93	12,21	14,93	14,14	12,58	15,17	14,53	13,00	15,62
13,52	11,79	14,56	13,99	12,25	15,00	14,35	12,79	15,37	14,38	13,04	15,64
13,53	11,76	14,41	14,03	12,31	15,08	14,44	12,89	15,39	14,40	12,84	15,46
13,68	11,87	14,54	14,03	12,24	14,97	14,36	12,77	15,40	14,40	13,00	15,64
13,50	11,78	14,49	14,00	12,21	14,97	14,46	12,79	15,40	14,45	13,06	15,69
13,52	11,88	14,61	14,22	12,42	15,09	14,47	12,76	15,30	14,53	12,99	15,58
13,58	11,89	14,57	13,94	12,20	14,95	14,43	12,80	15,33	14,34	12,88	15,52
13,65	11,93	14,62	14,24	12,36	15,13	14,55	12,86	15,41	14,46	12,93	15,54
Média	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
13,56	11,84	14,54	14,04	12,26	15,00	14,40	12,79	15,36	14,46	12,96	15,57
Desvio Padrão	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**
0,08	0,06	0,07	0,11	0,07	0,08	0,11	0,10	0,09	0,10	0,10	0,09
Máximo	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**
13,68	11,93	14,62	14,24	12,42	15,13	14,55	12,93	15,49	14,69	13,08	15,69
Mínimo	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**
13,43	11,76	14,41	13,93	12,20	14,91	14,14	12,58	15,17	14,34	12,78	15,39

Tabela 4.6 Configurações com um dicionário adaptativo de tamanho 512.

Tamanho do Dicionário Adaptativo: 1024											
Dicionario Fixo: 64			Dicionario Fixo: 128			Dicionario Fixo: 256			Dicionario Fixo: 380		
SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.	SNR clássico	SNR segment.	SNR percept.
14,00	12,39	15,07	14,49	12,87	15,57	14,75	13,35	15,85	14,49	13,26	15,75
13,89	12,48	15,11	14,49	12,91	15,59	14,88	13,27	15,75	14,85	13,23	15,75
13,99	12,04	15,07	14,31	12,81	15,45	14,95	13,38	15,91	14,76	13,04	15,60
13,97	12,51	15,19	14,37	12,69	15,44	14,79	13,34	15,78	14,90	13,14	15,63
14,00	12,49	15,13	14,22	12,73	15,43	14,63	13,40	15,88	14,90	13,18	15,62
14,07	12,54	15,19	14,26	12,72	15,42	14,67	13,23	15,73	14,72	13,08	15,62
13,81	12,32	15,04	14,48	12,81	15,50	14,88	13,23	15,74	14,94	13,38	15,91
14,04	12,47	15,12	14,37	12,82	15,50	14,83	13,24	15,73	14,54	13,27	15,79
13,83	12,39	15,01	14,42	12,85	15,48	14,76	13,08	15,65	14,99	13,40	15,95
13,75	12,44	15,10	14,26	12,78	15,44	14,02	13,20	15,68	14,97	13,38	15,94
Média	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
13,94	12,41	15,10	14,37	12,80	15,48	14,72	13,27	15,77	14,81	13,24	15,76
Desvio Padrão	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**	**Desvio Padrão**
0,11	0,14	0,06	0,10	0,07	0,06	0,26	0,10	0,09	0,18	0,13	0,14
Máximo	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**	**Máximo**
14,07	12,54	15,19	14,49	12,91	15,59	14,95	13,40	15,91	14,99	13,40	15,95
Mínimo	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**	**Mínimo**
13,75	12,04	15,01	14,22	12,69	15,42	14,02	13,08	15,65	14,49	13,04	15,60

Tabela 4.7 Configurações com um dicionário adaptativo de tamanho 1024.

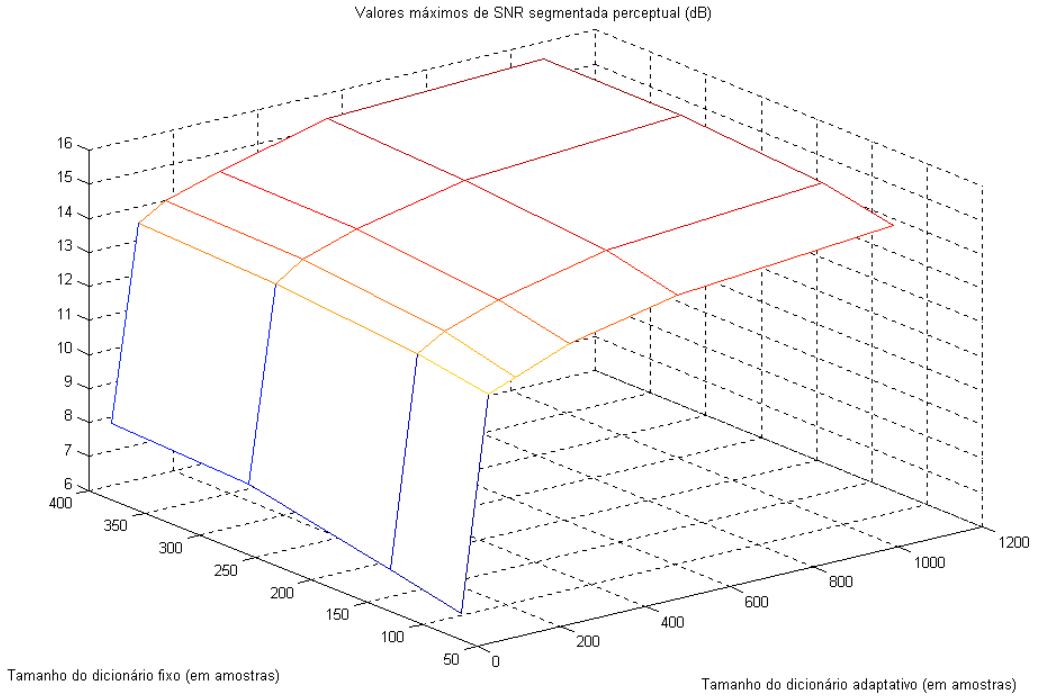

Figura 4.1 Valores máximos de SNR segmentada perceptiva.

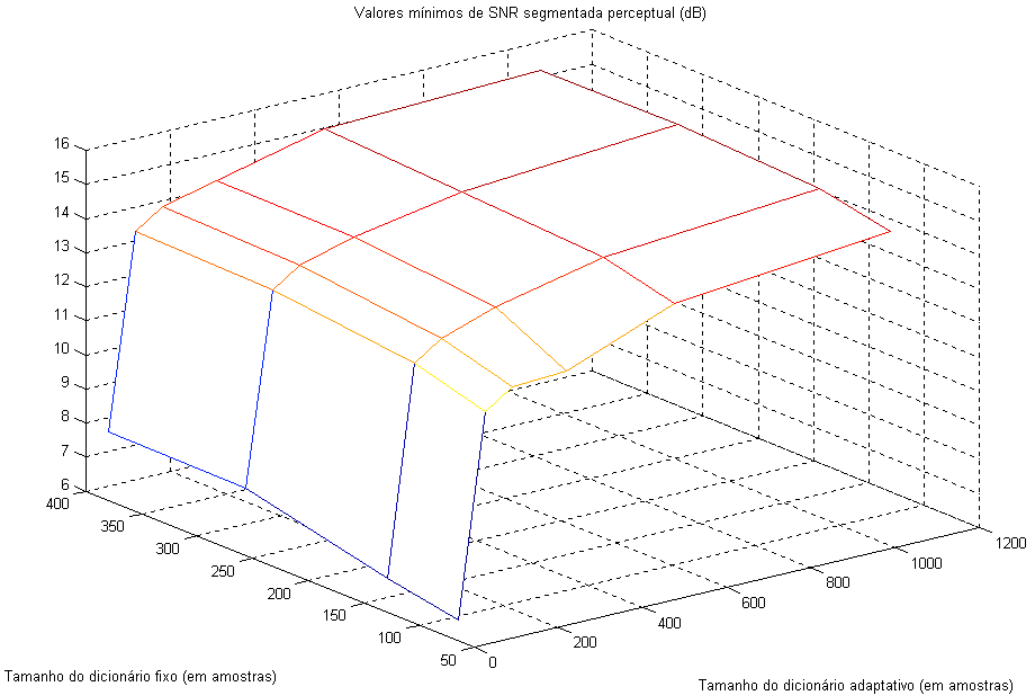

Figura 4.2 Valores mínimos de SNR segmentada perceptiva.

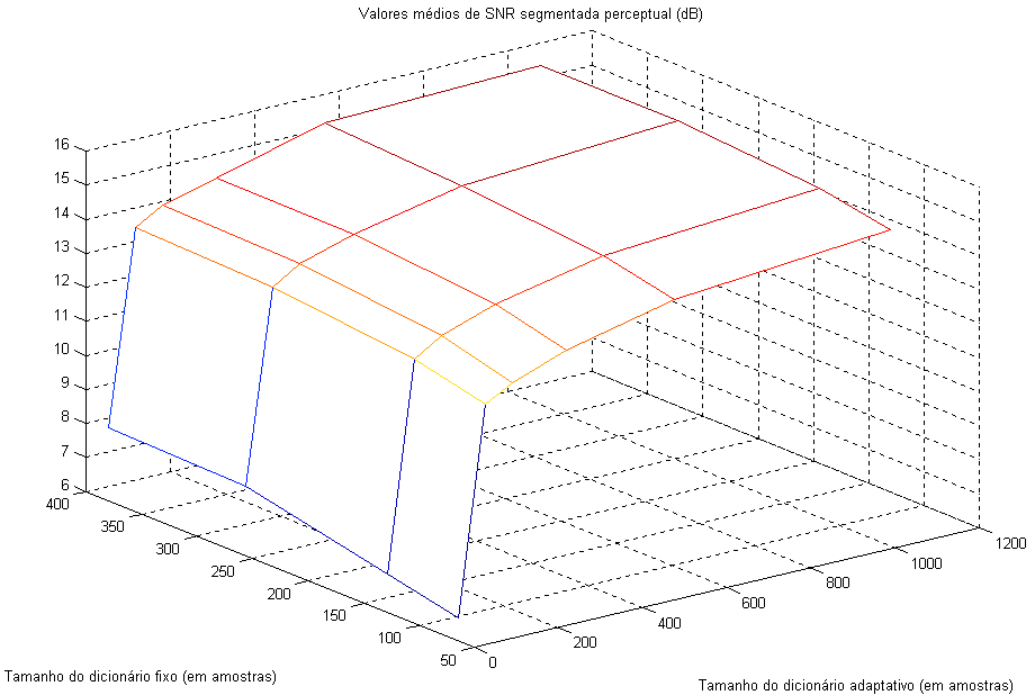

Figura 4.3 Valores médios de SNR segmentada perceptiva.

4.7 Análise dos Dados

Para realizar a análise dos dados apresentados na seção anterior, usar-se-á a Razão sinal / ruído segmentada perceptiva, por ser a melhor estimativa da qualidade subjetiva. As outras formas de medida também estão presentes para fim de comparação com a que foi efetivamente usada.

A qualidade do sinal de voz da saída do codificador é totalmente dependente do tamanho de ambos os dicionários. Para as configurações onde existia apenas o dicionário fixo, foram obtidas as mais baixas qualidades. Pode-se ver nas figuras 4.1, 4.2 e 4.3 que, à medida que se aumenta o tamanho do dicionário fixo, a qualidade aumenta proporcionalmente. Fato semelhante ocorre quando se aumenta o tamanho do dicionário adaptativo.

Além disso, pode-se analisar, a partir destas figuras e das tabelas mostradas anteriormente, como os dicionários contribuem individualmente para a variação da qualidade da voz decodificada. Para isso, foram montados os gráficos contidos nas figuras 4.4 e 4.5. Estes gráficos representam as contribuições individuais de cada dicionário. Tais figuras podem ser interpretadas da seguinte forma: considera-se, por exemplo, o primeiro gráfico da figura 4.4. Nele, é possível ver três barras, que representam pares ordenados. São estas (128; 0,46), (256; 0,89) e (380; 0,11). Isso indica que, quando não se usou o dicionário adaptativo, ao se variar o tamanho do dicionário fixo de 64 para 128, obteve-se uma melhora na SNR segmentada perceptiva de 0,46 dB. Analogamente, ao se variar o tamanho do dicionário fixo de 128 para 256, obteve-se uma melhora na SNR segmentada perceptiva de 0,89. Por fim, ao se variar o tamanho do dicionário fixo de 256 para 380, obteve-se uma melhora na SNR segmentada perceptiva de 0,11. Todos os demais gráficos podem ser entendidos de forma similar. Para facilitar a diferenciação entre contribuições positivas e negativas, foi traçada uma reta na altura da ordenada zero.

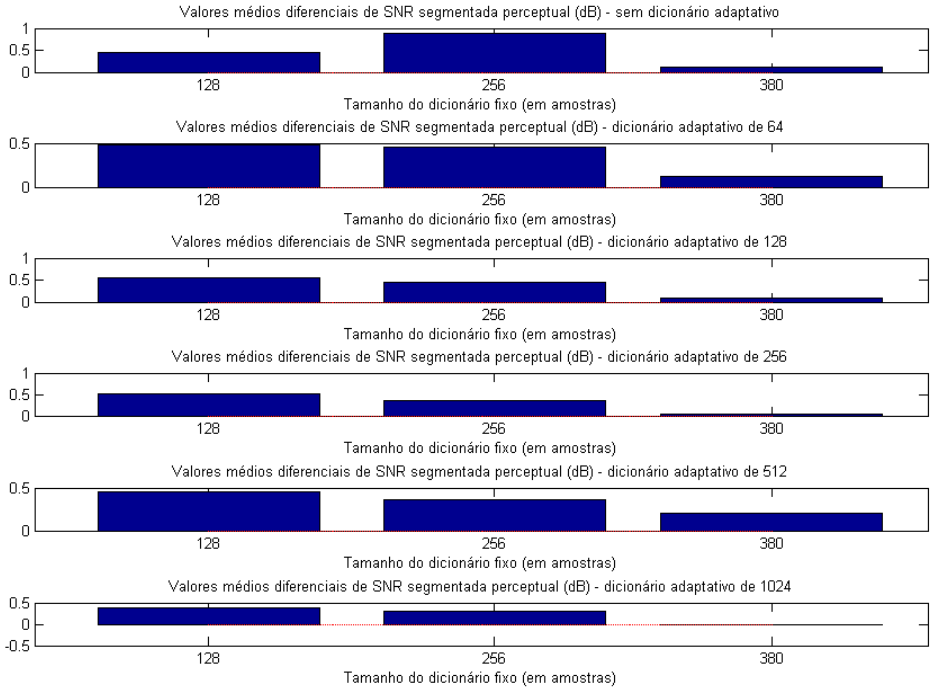

Figura 4.4 Contribuição do dicionário fixo.

Da mesma maneira, pode-se analisar os gráficos contidos na figura 4.5. Estes indicam a contribuição do dicionário adaptativo para a variação da SNR segmentada perceptiva da voz decodificada.

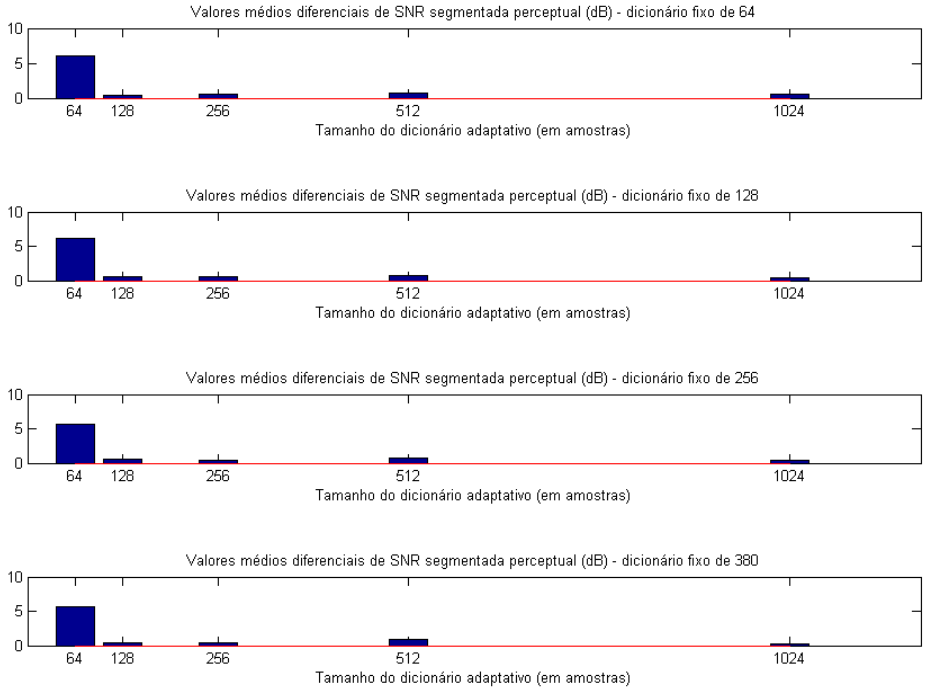

Figura 4.5 Contribuição do dicionário adaptativo.

Analisando-se estes gráficos, pode-se perceber que:

- Nem todos os aumentos de tamanho nos dicionários denotam melhorias na qualidade da voz decodificada. Isso pode ser visto pela variação negativa presente quando se varia o tamanho do dicionário fixo de 256 para 380, enquanto se mantém o dicionário adaptativo com o tamanho igual a1024. Isso ocorre pois, nesta condição, o tamanho do dicionário adaptativo tende a pesar no custo computacional;
- Ao analisar a figura 4.5, pode-se ver que, à medida que se aumenta o tamanho do dicionário adaptativo sua contribuição para a melhoria da qualidade da voz decodificada tende a cair. Contudo, a diferença entre usar o dicionário adaptativo e não usar é bem grande pelo fato de este ter grande poder de modelar sons sonoros;
- Por fim, analisando as tabelas, verifica-se que a melhor configuração é com o dicionário fixo de 256 e com o adaptativo de 1024, gerando uma média de 15,77

dB para SNR segmentada perceptiva para a voz decodificada da saída do sistema. Para esta mesma configuração, obteve uma qualidade mínima de 15,65 dB, que seria uma margem de segurança. A melhor performance obtida nesta configuração é de 15,91 dB, o que significa que, com estes tamanhos para os dicionários, a potência do sinal na saída do decodificador pode ser até 39 vezes maior que a potência do ruído;

Como o sistema *offline* foi feito com sinais de 8 *bits* e, este, com sinais de 16 *bits*, não se pode comparar a qualidade dos dois sistemas com bases nas medidas de qualidade objetiva.

4.9 Conclusão

Neste capítulo foram apresentados os métodos conhecidos para avaliação de sistemas de codificação de voz. Foi detalhada a classe **oss_quality** que é usada para realizar estimações da qualidade do sistema. Foram apresentadas as características do sistema. Foi apresentada a interface com o usuário. Foi explicado o algoritmo do sistema CELP implementado. Foram apresentados os dados dos resultados dos experimentos e análises sobre esses dados.

Capítulo 5

Conclusão

5.1 Resumo do Projeto

Este trabalho descreveu com detalhes o desenvolvimento de um sistema de codificação de voz CELP em tempo real implementado em *software*. Tal sistema foi derivado do sistema desenvolvido em [1] e que foi acelerado por [2].

Ao longo dessa dissertação, foram apresentados o sistema CELP, suas partes, seu diagrama de blocos e melhorias para seu processamento. Além disso, foram detalhadas as classes utilizadas para interfacear com o dispositivo de áudio, manipular os vetores do sinal de voz e atuar como base para o sistema CELP. Por fim, foram explicados métodos para se avaliar a qualidade do sinal de voz na saída do codificador, foi detalhada a classe que foi utilizada para se avaliar o sistema e foram analisados os resultados obtidos.

A seguir, são resumidos os conteúdos de cada capítulo:

- Capítulo 2:
    a. codificadores de forma de onda;
    b. codificadores de fonte ;
    c. codificadores híbridos, enfatizando o funcionamento do sistema CELP;
    d. melhorias ao sistema CELP apresentado, detalhando-se o funcionamento do sistema CELP já com estas melhorias;
    e. características do sistema e os ganhos por ele promovidos.
- Capítulo 3:
    a. OSS, Open Sound System;
    b. metodologia utilizada neste trabalho;
    c. diagrama de classes;

d. classes que atuaram como a base para o sistema: **oss_audio**, **oss_wave** e **oss_celp**.

- Capítulo 4:
    a. métodos conhecidos para avaliação de sistemas de codificação de voz;
    b. classe **oss_quality** que é usada para realizar estimações da qualidade do sistema;
    c. características do sistema;
    d. interface com o usuário;
    e. algoritmo do sistema CELP implementado;
    f. dados dos resultados dos experimentos;
    g. análises sobre esses dados.

5.2 Contribuições

O sistema inicial está descrito em [1] e foi escrito em linguagem C. Contudo, tal código apresentava um processamento muito lento. Tal código foi melhorado em [2], onde o processamento foi sensivelmente acelerado, mas o sistema ainda era formado por um extenso código referente a um sistema *off-line*. Isto significa que o sistema funcionava de modo a processar o sinal contido em um arquivo que já estava inicialmente armazenado na memória do computador.

Este trabalho resultou nas seguintes contribuições:

- Desenvolvimento de uma classe para manipular dispositivos de áudio em Linux através do OSS;
- Desenvolvimento de uma classe para manipular vetores de áudio;
- Desenvolvimento de uma classe para obter medidas objetivas da qualidade de um sinal de voz;
- Documentação de todas essas classes em forma de um manual de instruções;
- Fragmentação do código do codificador;

- Reestruturação do codificador segundo a lógica de Orientação a Objetos;
- Desenvolvimento de codificador em tempo real para Linux, obtendo-se uma Razão Sinal / Ruído média de 15,77 dB e máxima de 15,91 dB;

5.3 Propostas para Trabalhos Futuros

A seguir, são listadas as propostas para trabalhos futuros que tenham como objetivo a continuidade deste projeto ou seu aproveitamento:

- Estudo de tipos de quantização incluindo a quantização vetorial dos coeficientes;
- Estudo para a utilização de outras formas de implementação de dicionários fixos, como dicionários algébricos;
- Implementação deste sistema em DSP, também em tempo real;
- Estudo da utilização de filtros para a diminuição de ruídos na saída do codificador;
- Levar o sistema para utilização em banda larga;
- Implementação de sistema de voz sobre IP;
- Acoplar o codificador a um sistema de transmissão e recepção, como, por exemplo, CDMA, implementado em DSP.

Referências Bibliográficas

[1] MAIA, R. da S., *Codificação CELP e Análise Espectral de Voz*, Tese de M.Sc., PEE-COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2000

[2] B. de B. Oliveira, *Análise e Testes de um Codificador CELP*, Projeto Final DEL-Poli / UFRJ, Rio de Janeiro, RJ, Brasil, Abril de 2001

[3] DINIZ, P. S. R., da SILVA, E. A. B., NETTO, S. L., *Digital Signal Processing: System Analysis and Design*, Cambridge, UK, Cambridge, 2002

[4] DELLER, J. R., PROAKIS, J. G., HANSEN, J. H. L., *Discrete Time Process of Speech Signals*, New York, NY, USA, MacMillan, 1993

[5] http://www.opensound.com - *Site* sobre OSS (*Open Sound System*)

[6] JAMSA, K. , Klander, L., *C/C++ A Bíblia*, São Paulo, Makron Books, 1999

Apêndice A

Este apêndice descreve o modo através do qual se deve compilar o sistema, além da configuração necessária para seu funcionamento.

Para o correto funcionamento do programa, o computador deve possuir o sistema operacional Linux, placa de som com suporte ao modo *full-duplex*, microfone e caixas de som (ou fones de ouvido). Além disso, o usuário que estiver executando o programa deve possuir permissão para leitura e escrita em `/dev/dsp`.

Para a compilação do código, devem ser copiados para o mesmo diretório os seguintes arquivos:

- **oss_wave.h –** Arquivo de cabeçalho que contém os protótipos das classes *oss_audio* e *oss_wave*, além dos protótipos de seus métodos e declaração de seus atributos;
- **oss_wave.cpp –** Arquivo de código que contém as definições dos métodos das classes *oss_audio* e *oss_wave*;
- **oss_celp.h –** Arquivo de cabeçalho que contém os protótipos da classe *oss_celp*, além dos protótipos de seus métodos e declaração de seus atributos;
- **oss_celp.cpp –** Arquivo de código que contém as definições dos métodos da classe *oss_celp*;
- **oss_quality.h –** Arquivo de cabeçalho que contém os protótipos da classe *oss_quality*, além dos protótipos de seus métodos e declaração de seus atributos;
- **oss_quality.cpp –** Arquivo de código que contém as definições dos métodos da classe *oss_quality*;
- **codebook2048.dat** – Arquivo que armazena o dicionário fixo;
- **quantiza8.h –** Arquivo de cabeçalho necessário para a quantização;
- **main_celp.cpp** – Arquivo que contém a função *main.*

Para compilar o programa, usa-se a seguinte linha de comando:

```
g++ oss_wave.cpp oss_quality.cpp main_celp.cpp oss_celp.cpp -o oss_celp_teste -Wno-deprecated -DLINUX
```

Isso criará um arquivo executável de nome **oss_celp_teste**. Para executar o programa, deve-se digitar a seguinte linha de comando:

```
./oss_celp_teste
```

Apêndice B

Este apêndice enumera todos os métodos contidos em cada uma das classes que foram desenvolvidas para esse projeto. Isso é feito a fim de facilitar o trabalho do programador que for utilizar e/ou aperfeiçoar este sistema.

Além disso, ainda é exibido o diagrama de classes completo com vistas a melhorar a visualização do esforço de programação contido neste trabalho.

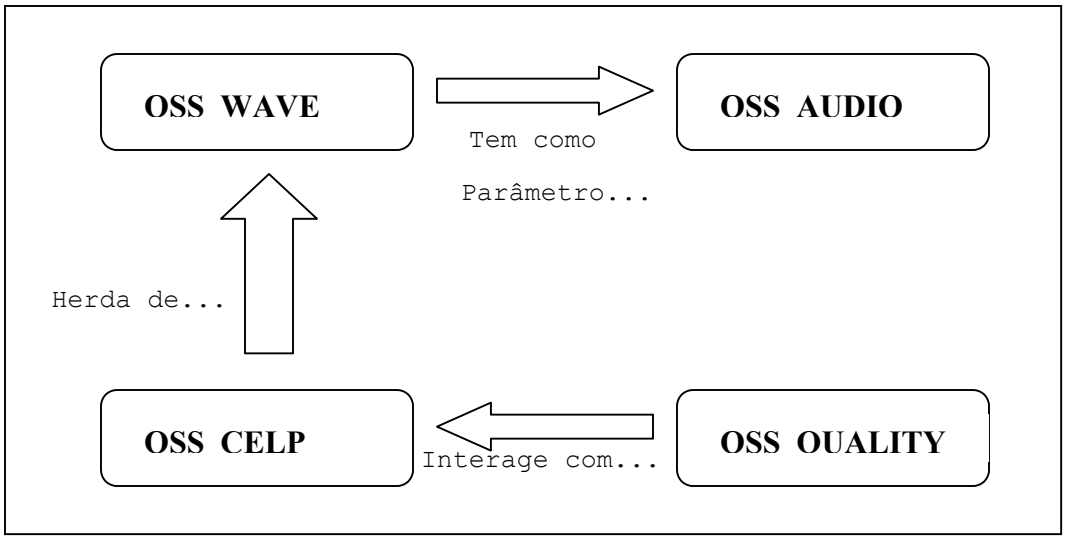

Tabela B.1 Diagrama de Classes

B.1 Métodos da Classe oss_audio

Valor de Retorno	Nome do Método
-	oss_audio(int, int)
void	Init()
int	GetSampleRate()
int	GetNumberOfChannels()
int	GetAudioDevice()

void	Close()

Tabela B.2 Métodos da classe oss_audio.

B.2 Métodos da Classe oss_wave

Valor de Retorno	**Nome do Método**
-	oss_wave(int, int)
int	Record(oss_audio, int)
int	Play(oss_audio)
void	SetLength(int)
int	GetLength()
void	SetData(int, signed short)
short	GetData(int)
int	GetSampleRate()
int	GetNumberOfChannels()
void	Append(oss_wave)
void	Equals(oss_wave)
void	SaveMatFile()

Tabela B.3 Métodos da classe oss_wave.

B.3 Métodos da Classe oss_celp

B.3.1 Construtor

O construtor da classe é o método **oss_celp**. Este é bem semelhante ao construtor da classe da qual herda suas propriedades, isto é, da **oss_wave**. Assim, tal método necessita, como argumentos, da freqüência de amostragem e do número de canais.

B.3.2 Métodos de "*Set*" e "*Get*"

Métodos de "*Set*"		**Métodos de "*Set*"**	
Valor de Retorno	**Nome do Método**	**Valor de Retorno**	**Nome do Método**
void	SetNumCoef(int)	int	GetNumCoef(void)
void	SetBlockSize(int)	int	GetBlockSize(void)
void	SetFCSize(int)	int	GetFCSize(void)
void	SetACSize(int)	int	GetACSize(void)
void	SetNumberOfSamples_AC(int)	int	GetNumberOfSamples_AC(void)
void	SetWindowSize(int)	int	GetWindowSize(void)
void	SetGamma(double)	double	GetGamma(void)
void	SetPercFilter(void)	double	GetPercFilter(int, int)
void	SetVoiceSubBlock(int)	-	-
void	SetPerc_ZeroInputResponse(void)	-	-
void	SetTargetSignal(void)	double	GetTargetSignal(int)

void	SetCompleteResponse(int)	-	-
void	SetDecodedVoiceSubBlock(int)	double	GetDecodedVoiceSubBlock(int)
-	-	double	GetCoefLPC(int)
-	-	double	GetCoefLSF(int)
-	-	double	GetCoefLSF_before(int)
-	-	double	GetCoefLPC_from_LSF(int)
-	-	double	GetCoefDLSF(int)
-	-	double	GetCoefLSF_Interpolated(int)
-	-	double	GetCoefLPC_from_LSF_Interpolated(int)
-	-	int	GetIndexAC(int)
-	-	double	GetGainAC(int)
-	-	int	GetIndexFC(int)
-	-	double	GetGainFC(int)

Tabela B.4 Métodos de "*Set*" e "*Get*" da classe oss_celp.

B.3.3 Métodos para Conversões

Valor de Retorno	**Nome do Método**
void	CoefLPC_to_CoefLSF(void)
void	CoefLSF_to_CoefLPC(void)
void	CoefLSF_Interpolated_to_CoefLPC(void)

Tabela B.5 Métodos para conversões da classe oss_celp.

B.3.4 Métodos Próprios para o Sistema CELP

Valor de Retorno	Nome do Método
void	AnalysisLPC(void)
void	SaveCoefLSF_before(void)
void	InterpolateCoefLSF(int)
void	QuantLSF(void)
void	Search_AC(int)
void	Search_FC(int)
void	UpdateTargetSignal(int)
void	UpdateAdaptiveCodebook(void)
void	ReceiveDecodedVoiceSubBlock(oss_celp)
void	Zero_SynthFilter_States(void)

Tabela B.6 Métodos próprios para o sistema CELP da classe oss_celp.

B.3.5 Métodos para filtragem

Valor de Retorno	Nome do Método
void	ApplyWindow(void)
void	ApplySynthFilter_to_FC(void)
void	ApplyPercFilter_to_VoiceSubBlock(void)
void	ApplySynthFilter_to_AC(void)

Tabela B.7 Métodos para filtragem da classe oss_celp.

B.3.6 Métodos para *Debug*

Valor de Retorno	Nome do Método
void	ShowCoefLPC(void)
void	ShowCoefLSF(void)
void	ShowCoefDLSF(void)
void	ShowCoefLPC_from_LSF(void)
void	ShowCoefLSF_before(void)
void	ShowCoefLSF_Interpolated(void)
void	ShowCoefLPC_from_LSF_Interpolated(void)
void	ShowPercFilter(void)
void	ShowTargetSignal(void)
void	ShowIndexAC(int)
void	ShowGainAC(int)
void	ShowIndexFC(int)
void	ShowGainFC(int)
void	ShowDecodedVoiceSubBlock(void)

Tabela B.8 Métodos para *debug* da classe oss_celp.

B.4 Métodos da Classe oss_quality

Valor de Retorno	Nome do Método
-	oss_quality
int	GetLength

void	SetSignal
void	SetEstimatedSignal
void	AccumulateSignalPower
void	AccumulateErrorPower
void	ZeroSignalPower
void	ZeroErrorPower
double	GetSNR
void	SetNumberOfSegments
int	GetNumberOfSegments
void	IncrementNumberOfSegments
void	AccumulateSNR_seg
double	GetSNR_seg
void	SetNumCoef
int	GetNumCoef
void	SetNumPercFilter
void	SetDenPercFilter
void	ZeroStatesPercFilter
void	AccumulatePercErrorPower
void	ZeroPercErrorPower
double	GetSNR_perc
double	GetSNR_seg_perc

Tabela B.9 Métodos da classe oss_quality.

Apêndice C

Este apêndice mostra os códigos que foram criados para a implementação do sistema CELP em tempo real. Tal sistema, como já foi explicado, foi escrito em C++ para o sistema operacional Linux (distribuição Red Hat 8.0).

C.1 OSS_WAVE.H

```
// OSS_WAVE.H
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 18/12/2002
// Version: 1.0

typedef unsigned short SAMPLE;

class oss_audio
{
 public:
  oss_audio(int SampleRate, int NumberOfChannels);
  void Init(void);
  void Close(void);
  int GetSampleRate(void);
  int GetNumberOfChannels(void);
  int GetAudioDevice(void);
 private:
  int Audio;
  int SampleRate;
  int NumberOfChannels;
};

class oss_wave
{
 public:
  oss_wave(int SampleRate, int NumberOfChannels);
  int Record(oss_audio AudioDev, double Time);
  int Play(oss_audio AudioDev);
  int GetLength(void);
  void SetLength(int Length);
  short GetData(int i);
  void SetData(int i, SAMPLE value);
  int GetNumberOfChannels(void);
  int GetSampleRate(void);
  void Append(oss_wave Other);
  void Equals(oss_wave Other);
  void SaveMatFile(void);
  void LoadDatFile(int, int);

 protected:
  int SampleRate;
  int NumberOfChannels;
  int Audio;
  SAMPLE *Data;
  int Length;
};
```

C.2 OSS_WAVE.CPP

```
// OSS_WAVE.CPP
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 18/12/2002
// Version: 1.0

#include <termios.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <limits.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>

#include <sys/ioctl.h>
#include <sys/soundcard.h>

#include "oss_wave.h"

// I define the variable MODE as a signed
// 16 bit audio format (little endian)
#define MODE AFMT_S16_LE

// I should comment that line if I want
// to debug a code based on this class.
// #define DEBUG

// Constructor for the OSS_AUDIO class.
// The arguments are the sample rate and the number
// of channels.
oss_audio::oss_audio(int ArgSampleRate, int ArgNumberOfChannels)
{
  // I define the appropriate attributes values according
  // to the arguments of the constructor.
  SampleRate = ArgSampleRate;
  NumberOfChannels = ArgNumberOfChannels;
}

// Function to initialize the audio device
void oss_audio::Init(void)
{
  // I open the audio device for reading and writing
  // The user should have reading and writing permissions
  // for the audio device (/dev/dsp).
  if ((Audio = open("/dev/dsp",O_RDWR, 0)) == -1)
    perror("/dev/dsp");

  // I set the audio device for full duplex mode
  // in order to read and write at the same time.
  if (ioctl(Audio, SNDCTL_DSP_SETDUPLEX, 0) == -1)
    perror("SNDCTL_DSP_SETDUPLEX");

  // I define the variable FORMAT according to MODE
  // (defined above as a #DEFINE)
  int Format = MODE;

  // I set the audio format
  if (ioctl(Audio, SNDCTL_DSP_SETFMT, &Format) == -1)
    perror("SNDCTL_DSP_SETFMT");

  // I catch the error for the operation above
```

```
  if (Format != MODE)
    fprintf(stderr,"soundcard does not support Signed 16 bit Little Endian\n");

  // I define the FORMAT value according to the number of
  // channels. 0 is for mono and 1 is for stereo sound.
  Format = NumberOfChannels - 1;

  // I define the number of channels for the audio device.
  if (ioctl(Audio, SNDCTL_DSP_STEREO, &Format) == -1)
      perror("SNDCTL_DSP_STEREO");

  // I catch the error for the operation above.
  if (Format != 0)
      fprintf(stderr,"soundcard does not support mono\n");

  // I define the sample rate for the audio device.
  if (ioctl(Audio, SNDCTL_DSP_SPEED, &SampleRate) == -1)
      perror("SNDCTL_DSP_SPEED");
}

// Function to close the audio device.
void oss_audio::Close(void)
{
  close(Audio);
}

// Function to get the sample rate defined.
int oss_audio::GetSampleRate(void)
{
  return(SampleRate);
}

// Function to get the number of channels defined.
int oss_audio::GetNumberOfChannels(void)
{
  return(NumberOfChannels);
}

// Function to get the audio device id number.
int oss_audio::GetAudioDevice(void)
{
  return(Audio);
}

// Constructor for the OSS_WAVE class.
// The arguments are the sample rate and the number
// of channels.
oss_wave::oss_wave(int ArgSampleRate, int ArgNumberOfChannels)
{
  // I define the appropriate attributes values according
  // to the arguments of the constructor
  SampleRate = ArgSampleRate;
  NumberOfChannels = ArgNumberOfChannels;

  // I define the pointer to the audio data as zero.
  Data = 0;
}

// Function to record audio signals from the microphone.
int oss_wave::Record(oss_audio AudioDev, double Time)
{
  // I define the variable AUXLENGTH
  int AuxLength;

  // The length should be as long as
  // the Sample Rate times the duration of the recording.
  // The duration of the recording is the argument
  // of the function, defined as TIME.
  Length = (int)(round(Time * SampleRate));

#ifdef DEBUG
```

```
  cout << "Number of samples to be recorded: " << Length << endl;

  cout << "Data: " << Data << endl;
#endif

  // If the audio data pointer is defined, the pointer must be deleted
  // to store some new information.
  if (Data)
    {
#ifdef DEBUG
      cout << "Getting some space for new samples..." << endl;
#endif
      // Here I actually delete the audio data pointer.
      delete [] Data;
    }

#ifdef DEBUG
  cout << "Allocating memory for new audio vector..." << endl;
#endif

  // I allocate memory for the audio data pointer.
  Data = new SAMPLE[Length];

  // I catch the error for the operation above.
  if (Data == NULL)
    cout << "Error on allocating memory for new audio vector." << endl;

  // I read the audio information from the microphone.
  // The arguments for the READ function is:
  // * the audio device
  // * the pointer to which the audio data will be copied
  // * the length of the audio data vector
  // The length of the audio data vector must be multiplied by 2
  // because only half of the time is recorded.
  if ((AuxLength = read(AudioDev.GetAudioDevice(), Data, sizeof(SAMPLE)*Length)) == -1)
    {
      perror("Audio Record");
      return(1);
    }

  // I catch the error for the operation above
  if (AuxLength < sizeof(SAMPLE)*Length)
    {
      fprintf(stderr,"Incomplete record");
      return(1);
    }
  return(0);
}

// Function to play audio signals
int oss_wave::Play(oss_audio AudioDev)
{
#ifdef DEBUG
  cout << "Number of samples to be played: " << Length << endl;
#endif
  // I read the audio information from the microphone.
  // The arguments for the WRITE function is:
  // * the audio device
  // * the pointer from which the audio data will be copied
  // * the length of the audio data vector
  // The length of the audio data vector must be multiplied by 2
  // because only half of the time is played.
  if ((write(AudioDev.GetAudioDevice(), Data, sizeof(SAMPLE)*Length)) == -1)
    {
      perror("Audio Write");
      return(1);
    }

  return(0);
}
```

```
// Function to get the length of the audio data vector.
int oss_wave::GetLength()
{
  return(Length);
}

// Function to set the length of the audio data vector.
void oss_wave::SetLength(int ArgLength)
{
  Length = ArgLength;
}

// Function to get a sample of the audio data vector.
short oss_wave::GetData(int i)
{
  return(Data[i]);
}

// Function to set the valeu of a sample of the audio data vector.
void oss_wave::SetData(int i, SAMPLE value)
{
  Data[i] = value;
}

// Function to get the sample rate.
int oss_wave::GetSampleRate(void)
{
  return(SampleRate);
}

// Function to get the number of channels.
int oss_wave::GetNumberOfChannels(void)
{
  return(NumberOfChannels);
}

// Function to concatenate two audio data vectors.
void oss_wave::Append(oss_wave Other)
{
  SAMPLE* DataAux;

  // I allocate memory for the audio data vector which length
  // is the sum of the two lengths.
  DataAux = new SAMPLE[GetLength() + Other.GetLength()];
  if (!DataAux)
    cout << "Error on memory alloc." << endl;

  // Here I actually concatenate the two audio data vectors
  // forming a new one.
  for (int i=0; i < GetLength() + Other.GetLength(); i++)
    if (i < GetLength())
      DataAux[i] = Data[i];
    else
      DataAux[i] = Other.Data[i - GetLength()];

  // I delete this audio data vector
  // if it is defined
  if (Data)
    delete [] Data;

  // I reallocate memory for the new vector.
  Data = new SAMPLE[GetLength() + Other.GetLength()];
  if (!Data)
    cout << "Error on memory alloc." << endl;

  // Then, I make the new pointer equal to the old one.
  Data = DataAux;
  // Finally, I refresh the audio data vector length.
  SetLength(GetLength() + Other.GetLength());
}
```

```
// Function to make an audio data vector equal to another.
void oss_wave::Equals(oss_wave Other)
{
  SAMPLE* DataAux;

  // I allocate memory for the new audio data vector
  DataAux = new SAMPLE[Other.GetLength()];
  if (!DataAux)
    cout << "Error on memory alloc." << endl;

  // So, I fill the new vector up
  for (int i=0; i < Other.GetLength(); i++)
      DataAux[i] = Other.Data[i];

  // If the audio data pointer is defined,
  // I should delete it
  if (Data)
    delete [] Data;

  // I allocate memory for the new vector
  Data = new SAMPLE[Other.GetLength()];
  if (!Data)
    cout << "Error on memory alloc." << endl;

  // I, then, make one vector equal to the other one
  Data = DataAux;
  // Finally I refresh the audio data vector length
  SetLength(Other.GetLength());
}

// Functio to save the audio data vector into a MAT file
void oss_wave::SaveMatFile(void)
{
  ofstream output;

  output.open("oss_wave.mat");

  if (!output)
    {
      cout << "Error writing .mat file!" << endl;
    }

  for (int i=0; i<GetLength(); i++)
    output << GetData(i) << endl;
}


// Functio to load the audio data vector from a MAT file
void oss_wave::LoadDatFile(int size, int index)
{
  // Local variable
  FILE *fp;
  short aux[1];

  SetLength(size);
  // If the audio data pointer is defined, the pointer must be deleted
  // to store some new information.
  if (Data)
    {
      // Here I actually delete the audio data pointer.
      delete [] Data;
    }

  Data = new SAMPLE[size];
  if (!Data)
    cout << "Error on memory alloc for data vector." << endl;

  // Opens binary file
  if ((fp = fopen("load_wave.dat","rb")) == NULL)
    {
      fprintf(stderr,"Error on opening data file.\n");
```

```
      exit(1);
    }

  // Reads data from file
  for(int i=0; i < index; i++)
    if (fread(aux, sizeof(SAMPLE), 1, fp) != 1)
      {
       fprintf(stderr,"Error on reading from data file.\n");
       exit(1);
      }

  // Reads data from file
  if (fread(Data, sizeof(SAMPLE), size, fp) != size)
    {
      fprintf(stderr,"Error on reading from data file.\n");
      exit(1);
    }

  // Closes data file
   fclose(fp);
}
```

C.3 OSS_CELP.H

```
// OSS_CELP.H
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 18/12/2002
// Version: 1.0

#include "oss_wave.h"
#include <math.h>
#include <iostream.h>

class oss_celp : public oss_wave
{
 public:

  // Constructor
  oss_celp(int argSampleRate, int argNumOfChan);

  // Functions for setting something
  void SetNumCoef(int);
  void SetBlockSize(int);
  void SetFCSize(int);
  void SetACSize(int);
  void SetNumberOfSamples_AC(int);
  void SetWindowSize(int);
  void SetGamma(double);
  void SetPercFilter(void);
  void SetVoiceSubBlock(int);
  void SetTargetSignal(void);
  void SetCompleteResponse(int);
  void SetDecodedVoiceSubBlock(int);

  // Functions for getting something
  int GetNumCoef(void);
  int GetBlockSize(void);
  int GetFCSize(void);
  int GetACSize(void);
  int GetNumberOfSamples_AC(void);
  double GetCoefLPC(int);
  double GetCoefLSF(int);
  double GetCoefLSF_before(int);
  double GetCoefLPC_from_LSF(int);
  double GetCoefDLSF(int);
  double GetCoefLSF_Interpolated(int);
  double GetCoefLPC_from_LSF_Interpolated(int);
```

```
 int GetWindowSize(void);
 double GetGamma(void);
 double GetPercFilter(int, int);
 double GetTargetSignal(int);
 int GetIndexAC(int);
 double GetGainAC(int);
 int GetIndexFC(int);
 double GetGainFC(int);
 double GetDecodedVoiceSubBlock(int);

 // Functions for conversions
 void CoefLPC_to_CoefLSF(void);
 void CoefLSF_to_CoefLPC(void);
 void CoefLSF_Interpolated_to_CoefLPC(void);

 // Functions for CELP
 void AnalysisLPC(void);
 void SaveCoefLSF_before(void);
 void InterpolateCoefLSF(int);
 void QuantLSF(void);
 void Search_AC(int);
 void Search_FC(int);
 void UpdateTargetSignal(int);
 void UpdateAdaptiveCodebook(void);
 void ReceiveDecodedVoiceSubBlock(oss_celp);
 void Zero_SynthFilter_States(void);
 void ReceiveCoefInfo(oss_celp);
 void ReceiveCodebookInfo(oss_celp, int);

 // Functions for filtering
 void ApplyWindow(void);
 void ApplySynthFilter_to_FC(void);
 void ApplyPercFilter_to_VoiceSubBlock(void);
 void ApplySynthFilter_to_AC(void);

 // Functions for showing / debugging something
 void ShowCoefLPC(void);
 void ShowCoefLSF(void);
 void ShowCoefDLSF(void);
 void ShowCoefLPC_from_LSF(void);
 void ShowCoefLSF_before(void);
 void ShowCoefLSF_Interpolated(void);
 void ShowCoefLPC_from_LSF_Interpolated(void);
 void ShowPercFilter(void);
 void ShowTargetSignal(void);
 void ShowIndexAC(int);
 void ShowGainAC(int);
 void ShowIndexFC(int);
 void ShowGainFC(int);
 void ShowDecodedVoiceSubBlock(void);

protected:
 int NumCoef;
 int BlockSize;
 int FCSize;
 int ACSize;
 int NumberOfSamples_AC;
 double* VoiceSubBlock;
 double* Perc_VoiceSubBlock;
 double* CoefLPC;
 double* CoefLSF_before;
 double* CoefLSF_Interpolated;
 double* CoefLSF;
 double* CoefLSF_quant;
 double* CoefDLSF;
 double* CoefLPC_from_LSF;
 double* CoefLPC_from_LSF_Interpolated;
 double* CoefLPC_from_LSF_before;
 int WindowSize;
 double* Window;
 double* WindowSignal;
```

```
  double Gamma;
  double* Num_PercFilter;
  double* Den_PercFilter;
  double* ZeroInput;
  double* ZeroInputResponse;
  double* Perc_ZeroInputResponse;
  double* States_SynthFilter;
  double* States_SynthFilter_Decod;
  double* States_Num_PercFilter;
  double* States_Den_PercFilter;
  double* AdaptiveCodebook;
  double* Perc_AdaptiveCodebook;
  double* Sequence_AC;
  double* Reading_AC;
  double* Response_AC;
  int Index_AC[4];
  double Gain_AC[4];
  double* Sequence_FC;
  double* Reading_FC;
  double* Response_FC;
  int Index_FC[4];
  double Gain_FC[4];
  double* TargetSignal;
  double* DecodedVoiceSubBlock;
  double* DecodedExcitation;
  double* Den_SynthFilter;
  double* Error;
  double ErrorPower;
  double SignalPower;
  double *SavedAdaptiveCodebook;

 public:

  float* FixedCodebook;
  double* Perc_FixedCodebook;
  double *CompleteExcitation;
  double* CompleteResponse;
};
```

C.4 OSS_CELP.CPP

```
// OSS_CELP.CPP
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 18/12/2002
// Version: 1.0

#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include "oss_celp.h"
#include "quantiza8.h"

// Constructor for the OSS_CELP class
oss_celp::oss_celp(int argSampleRate, int argNumChan) : oss_wave(argSampleRate,
argNumChan)
{
  SampleRate = argSampleRate;
  NumberOfChannels = argNumChan;
  Data = 0;
  SignalPower = 0;
  ErrorPower = 0;
}

// Function to determine the number of coefficients that will be
// used in the linear regression
void oss_celp::SetNumCoef(int argNumCoef)
{
  NumCoef = argNumCoef;
```

```
  // Allocating memory for pointers
  CoefLPC = new double[NumCoef];
  if (!CoefLPC)
    {
      cout << "Error on allocating memory for the LPC coefficients." << endl;
      exit(1);
    }

  CoefLSF = new double[NumCoef];
  if (!CoefLSF)
    {
      cout << "Error on allocating memory for the LSF coefficients." << endl;
      exit(1);
    }

  CoefLPC_from_LSF = new double[NumCoef];
  if (!CoefLPC_from_LSF)
    {
      cout << "Error on allocating memory for the LPC coefficients that were calculated
from the LSF ones." << endl;
      exit(1);
    }

  CoefLPC_from_LSF_Interpolated = new double[NumCoef];
  if (!CoefLPC_from_LSF_Interpolated)
    {
      cout << "Error on allocating memory for the LPC coefficients that were calculated
from the Interpolated LSF ones." << endl;
      exit(1);
    }

  CoefLPC_from_LSF_before = new double[NumCoef];
  if (!CoefLPC_from_LSF_before)
    {
      cout << "Error on allocating memory for the LPC coefficients that were calculated
from the LSF ones from the last block." << endl;
      exit(1);
    }

  CoefLSF_before = new double[NumCoef];
  if (!CoefLSF_before)
    {
      cout << "Error on allocating memory for the LSF coefficients from the last block."
<< endl;
      exit(1);
    }

  CoefLSF_quant = new double[NumCoef];
  if (!CoefLSF_quant)
    {
      cout << "Error on allocating memory for the quantized LSF coefficients." << endl;
      exit(1);
    }

  CoefDLSF = new double[NumCoef];
  if (!CoefDLSF)
    {
      cout << "Error on allocating memory for the DLSF coefficients." << endl;
      exit(1);
    }

  CoefLSF_Interpolated = new double[NumCoef];
  if (!CoefLSF_Interpolated)
    {
      cout << "Error on allocating memory for the interpolated LSF coefficients" << endl;
      exit(1);
    }

  Num_PercFilter = new double[NumCoef+1];
  if (!Num_PercFilter)
```

```
    {
      cout << "Error on allocating memory for the numerator of the perceptual filter" <<
endl;
      exit(1);
    }

  Den_PercFilter = new double[NumCoef+1];
  if (!Den_PercFilter)
    {
      cout << "Error on allocating memory for the denominator of the perceptual filter"
<< endl;
      exit(1);
    }

  States_Num_PercFilter = new double[NumCoef];
  if (!States_Num_PercFilter)
    {
      cout << "Error on allocating memory for the states of the numerator of the
perceptual filter" << endl;
      exit(1);
    }

  States_Den_PercFilter = new double[NumCoef];
  if (!States_Den_PercFilter)
    {
      cout << "Error on allocating memory for the states of the denominator of the
perceptual filter" << endl;
      exit(1);
    }

  States_SynthFilter = new double[NumCoef];
  if (!States_SynthFilter)
    {
      cout << "Error on allocating memory for the states of the denominator of the
synthesis filter" << endl;
      exit(1);
    }

  States_SynthFilter_Decod = new double[NumCoef];
  if (!States_SynthFilter_Decod)
    {
      cout << "Error on allocating memory for the states of the denominator of the
synthesis filter for decoding" << endl;
      exit(1);
    }

  Den_SynthFilter = new double[NumCoef+1];
  if (!Den_SynthFilter)
    {
      cout << "Error on allocating memory for the denominator of the synthesis filter" <<
endl;
      exit(1);
    }

  for(int i=0; i < NumCoef + 1; i++)
    CoefLSF_before[i] = 0;

  // Initializing the Synthesis and Perceptual Filters states
  for (int j = 0; j < GetNumCoef(); j++)
    States_SynthFilter[j] = States_Num_PercFilter[j] = States_Den_PercFilter[j] = 0.0;
}

// Function to determine the size of the voice block
void oss_celp::SetBlockSize(int size)
{
  BlockSize = size;

  VoiceSubBlock = new double[size/4];
  if (!VoiceSubBlock)
    {
      cout << "Error on allocating memory for the voice sub-block" << endl;
```

```
      exit(1);
    }

  Perc_VoiceSubBlock = new double[size/4];
  if (!Perc_VoiceSubBlock)
    {
      cout << "Error on allocating memory for the ponderated voice sub-block" << endl;
      exit(1);
    }

  // Allocating memory
  ZeroInput = new double[size/4];
  if (!ZeroInput)
    {
      cout << "Error on allocating memory for the Zero Input" << endl;
      exit(1);
    }

  // Filling the zero input up
  for(int i=0; i<size/4; i++)
    ZeroInput[i] = 0.0;

  Perc_ZeroInputResponse = new double[size/4];
  if (!Perc_ZeroInputResponse)
    {
      cout << "Error on allocating memory for the ponderated zero input response" <<
endl;
      exit(1);
    }

  ZeroInputResponse = new double[size/4];
  if (!ZeroInputResponse)
    {
      cout << "Error on allocating memory for the zero input response" << endl;
      exit(1);
    }

  TargetSignal = new double[size/4];
  if (!TargetSignal)
    {
      cout << "Error on allocating memory for the target signal" << endl;
      exit(1);
    }

  Sequence_AC = new double[size/4];
  if (!Sequence_AC)
    {
      cout << "Error on allocating memory for the sequence from the Adaptive Codebook" <<
endl;
      exit(1);
    }

  Reading_AC = new double[size/4];
  if (!Reading_AC)
    {
      cout << "Error on allocating memory for the sequence from the Adaptive Codebook" <<
endl;
      exit(1);
    }

  Response_AC = new double[size/4];
  if (!Response_AC)
    {
      cout << "Error on allocating memory for the response from the Adaptive Codebook" <<
endl;
      exit(1);
    }

  Sequence_FC = new double[size/4];
  if (!Sequence_FC)
    {
```

```
      cout << "Error on allocating memory for the sequence from the Fixed Codebook" <<
endl;
      exit(1);
    }

  Reading_FC = new double[size/4];
  if (!Reading_FC)
    {
      cout << "Error on allocating memory for the sequence from the Fixed Codebook" <<
endl;
      exit(1);
    }

  Response_FC = new double[size/4];
  if (!Response_FC)
    {
      cout << "Error on allocating memory for the response from the Fixed Codebook" <<
endl;
      exit(1);
    }

  CompleteResponse = new double[size/4];
  if (!CompleteResponse)
    {
      cout << "Error on allocating memory for the complete response" << endl;
      exit(1);
    }

  DecodedVoiceSubBlock = new double[size/4];
  if (!DecodedVoiceSubBlock)
    {
      cout << "Error on allocating memory for the decoded voice sub block" << endl;
      exit(1);
    }

  Error = new double[size/4];
  if (!Error)
    {
      cout << "Error on allocating memory for the error between the decoded and the
original signals" << endl;
      exit(1);
    }

  DecodedExcitation = new double[size/4];
  if (!DecodedExcitation)
    {
      cout << "Error on allocating memory for the decoded excitation" << endl;
      exit(1);
    }

  // Initializing the zero-input response and the ponderated zero-input response
  for (int j = 0; j < GetBlockSize()/4; j++)
    Perc_ZeroInputResponse[j] = ZeroInputResponse[j] = 0.0;

  CompleteExcitation = new double[GetBlockSize()/4];
  if (!CompleteExcitation)
    {
      cout << "Error on allocating memory for the Complete Excitation." << endl;
      exit(1);
    }
}

// Function to determine the size of the FC
void oss_celp::SetFCSize(int size)
{
  void load_from_file(const char file[], float variable[], int num_itens);

  FCSize = size;

  FixedCodebook = new float[2*size + (GetBlockSize()/4)];
  if (!FixedCodebook)
```

```
    {
      cout << "Error on allocating memory for the Fixed Codebook" << endl;
      exit(1);
    }

  Perc_FixedCodebook = new double[2*size + (GetBlockSize()/4)];
  if (!Perc_FixedCodebook)
    {
      cout << "Error on allocating memory for the ponderated Fixed Codebook" << endl;
      exit(1);
    }

  // Reading the sequences from the FC
  cout << "Setting up fixed codebook...";
  int NumberOfSamples_FC = 2 * GetFCSize() + GetBlockSize()/4;
  load_from_file("codebook2048.dat", FixedCodebook, NumberOfSamples_FC);
  cout << "OK" << endl;
}

// Function to determine the size of the AC
void oss_celp::SetACSize(int size)
{
  ACSize = size;

  AdaptiveCodebook = new double[size + GetBlockSize()/4];
  if (!AdaptiveCodebook)
    {
      cout << "Error on allocating memory for the Adaptive Codebook" << endl;
      exit(1);
    }

  Perc_AdaptiveCodebook = new double[size + GetBlockSize()/4];
  if (!Perc_AdaptiveCodebook)
    {
      cout << "Error on allocating memory for the ponderated Adaptive Codebook" << endl;
      exit(1);
    }
}

// Function to determine the number of samples for the AC
void oss_celp::SetNumberOfSamples_AC(int number)
{
  NumberOfSamples_AC = number;

  // Initializing the AC
  for (int j = 0; j < number; j++)
    AdaptiveCodebook[j] = 0.0;

  SavedAdaptiveCodebook = new double[number];
  if (!SavedAdaptiveCodebook)
    {
      cout << "Error on allocating memory for the Adaptive Codebook" << endl;
      exit(1);
    }
}

// Function to determine the size of the windows (in samples)
void oss_celp::SetWindowSize(int Size)
{
  WindowSize = Size;

  if ((Window = (double*)calloc(Size, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Nao foi possivel alocar memoria.\n");
      exit(1);
    }
  if ((WindowSignal = (double*)calloc(Size, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Nao foi possivel alocar memoria.\n");
      exit(1);
    }
```

```
}

// Function to determine the gamma factor for the perceptual filter
void oss_celp::SetGamma(double ArgGamma)
{
  Gamma = ArgGamma;
}

// Function to set the coefficients for the perceptual filter w(z)
void oss_celp::SetPercFilter(void)
{
  Num_PercFilter[0] = 1;
  Den_PercFilter[0] = 1;

  for(int i=1; i<=GetNumCoef(); i++)
    {
      Num_PercFilter[i] = - CoefLPC_from_LSF_Interpolated[i-1];
      Den_PercFilter[i] = (pow(GetGamma(), i)) * Num_PercFilter[i];
    }
}

// Function to fill the VoiceSubBlock vector
void oss_celp::SetVoiceSubBlock(int k)
{
  for(int i=0; i<(GetBlockSize()/4); i++)
    VoiceSubBlock[i] = GetData(i + (GetBlockSize()/4)*(k-1));
}

// Function to determine the target signal
void oss_celp::SetTargetSignal(void)
{
  for(int i=0; i<(GetBlockSize()/4); i++)
    {
      TargetSignal[i] = Perc_VoiceSubBlock[i] - ZeroInputResponse[i];
    }
}

// Function to determine the complete response
// considering the fixed and adaptive dicionarys
// for a certain voice sub-block
void oss_celp::SetCompleteResponse(int k)
{
  // Necessary funtion prototypes
  void filt_sp1(double s_input[], double s_output[], double den_filter[],
              double buffer[], int signal_size, int buffer_size);

  void filt_sp2(double s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size);
  void read_sequence(double AdaptiveCodebook[], double Reading_AdaptiveCodebook[],
                 int index, int size_ad, int size_seq);

  // Reading the candidate sequence from the AC which has already gone into the synthesis
filter
  read_sequence(AdaptiveCodebook, Reading_AC, Index_AC[k-1], GetNumberOfSamples_AC(),
GetBlockSize()/4);

  // Reading the candidate sequence from the FC which has already gone into the synthesis
filter
  for(int b=0; b<GetBlockSize()/4; b++)
    {
      Reading_FC[b] = FixedCodebook[b+2*(Index_FC[k-1]-1)];
    }

  // Applying the gain
  for(int i=0; i<GetBlockSize()/4; i++)
    {
      Reading_AC[i] *= Gain_AC[k-1];
      Reading_FC[i] *= Gain_FC[k-1];
      CompleteExcitation[i] = Reading_AC[i] + Reading_FC[i];
    }
```

```
  for(int j=0; j<GetNumCoef(); j++)
    States_SynthFilter[j] = 0.0;

  // Applying Synthesis Filter
  filt_sp1(CompleteExcitation, CompleteResponse, Den_PercFilter, States_SynthFilter,
GetBlockSize()/4, GetNumCoef());
  filt_sp1(ZeroInput, ZeroInputResponse, Den_PercFilter, States_SynthFilter,
GetBlockSize()/4, GetNumCoef());
}

// Function to determine the decoded voice
// sub-block
void oss_celp::SetDecodedVoiceSubBlock(int k)
{
  // Necessary function prototype
  void filt_sp1(double s_input[], double s_output[], double den_filter[],
              double buffer[], int signal_size, int buffer_size);
  void read_sequence(double AdaptiveCodebook[], double Reading_AdaptiveCodebook[],
                 int index, int size_ad, int size_seq);

  // Reading the candidate sequence from the AC which has already gone into the synthesis
filter
  read_sequence(AdaptiveCodebook, Reading_AC, Index_AC[k-1], GetNumberOfSamples_AC(),
GetBlockSize()/4);

  // Reading the candidate sequence from the FC which has already gone into the synthesis
filter
  for(int b=0; b<GetBlockSize()/4; b++)
    {
      Reading_FC[b] = FixedCodebook[b+2*(Index_FC[k-1]-1)];
    }

  for(int i=0; i<(GetBlockSize()/4); i++)
    {
      DecodedExcitation[i] = (Gain_AC[k-1] * Reading_AC[i]) + (Gain_FC[k-1] *
Reading_FC[i]);
    }

  // Calculating the denominator of the Synthesis Filter
  Den_SynthFilter[0] = 1.0;

  for (int j = 0; j < GetNumCoef(); j++)
    {
      Den_SynthFilter[j+1] = - CoefLPC_from_LSF_Interpolated[j];
    }

  // Applying the decoded excitation into the Synthesis
  // to calculate the decoded voice sub block
  filt_sp1(DecodedExcitation, DecodedVoiceSubBlock, Den_SynthFilter,
States_SynthFilter_Decod, GetBlockSize()/4, GetNumCoef());

  // >>>>>>>>>>>>>>>>>>>>  UNCOMMENT THESE TWO LINES BELOW TO TEST TALK THROUGH
<<<<<<<<<<<<<<<<<<<<
  //for(int w=0; w<GetBlockSize()/4; w++)
  //  DecodedVoiceSubBlock[w] = GetData( (k-1)*(GetBlockSize()/4) + w );
}

// Function to get the number of coefficients that will be used
// in the linear regression
int oss_celp::GetNumCoef(void)
{
  return(NumCoef);
}

// Function to get the size of the voice block
int oss_celp::GetBlockSize(void)
{
  return(BlockSize);
}

// Function to get the size of the FC
```

```
int oss_celp::GetFCSize(void)
{
  return(FCSize);
}

// Function to get the size of the AC
int oss_celp::GetACSize(void)
{
  return(ACSize);
}

// Function to get the number of samples of the AC
int oss_celp::GetNumberOfSamples_AC(void)
{
  return(NumberOfSamples_AC);
}

// Function to get the LPC coefficient of index "i"
double oss_celp::GetCoefLPC(int i)
{
  return(CoefLPC[i]);
}

// Function to get the LSF coefficient of index "i"
double oss_celp::GetCoefLSF(int i)
{
  return(CoefLSF[i]);
}

// Function to get the LSF coefficient of index "i" from the last block
double oss_celp::GetCoefLSF_before(int i)
{
  return(CoefLSF_before[i]);
}

// Function to get the LPC coefficient of index "i" that come from the
// LSF coefficients
double oss_celp::GetCoefLPC_from_LSF(int i)
{
  return(CoefLPC_from_LSF[i]);
}

// Function to get the DLSF coefficient od index "i"
double oss_celp::GetCoefDLSF(int i)
{
  return(CoefDLSF[i]);
}

// Function to get the LSF coefficient of index "i"
// that have been interpolated in the sub-block k
double oss_celp::GetCoefLSF_Interpolated(int i)
{
  return(CoefLSF_Interpolated[i]);
}

// Function to get the LSF coefficient of index "i"
// that have been interpolated in the sub-block k
double oss_celp::GetCoefLPC_from_LSF_Interpolated(int i)
{
  return(CoefLPC_from_LSF_Interpolated[i]);
}

// Function to get the size of the window (in samples)
int oss_celp::GetWindowSize(void)
{
  return(WindowSize);
}

// Function to get the gamma factor for the perceptual filter
double oss_celp::GetGamma(void)
{
```

```
  return(Gamma);
}

// Function to get a coefficient from the numerator (a=0) or the denominator (a=1)
// of the perceptual filter
double oss_celp::GetPercFilter(int a, int b)
{
  if (a==0)
    return(Num_PercFilter[b]);

  if (a==1)
    return(Den_PercFilter[b]);
}

// Function to get a sample of the target signal
double oss_celp::GetTargetSignal(int i)
{
  return(TargetSignal[i]);
}

// Function to get the index of the AC
// for some voice sub-block
int oss_celp::GetIndexAC(int k)
{
  return(Index_AC[k]);
}

// Function to get the gain of the AC
// for some voice sub-block
double oss_celp::GetGainAC(int k)
{
  return(Gain_AC[k]);
}

// Function to get the index of the FC
// for some voice sub-block
int oss_celp::GetIndexFC(int k)
{
  return(Index_FC[k]);
}

// Function to get the gain of the FC
// for some voice sub-block
double oss_celp::GetGainFC(int k)
{
  return(Gain_FC[k]);
}

// Function to get a sample of the decoded
// voice sub block
double oss_celp::GetDecodedVoiceSubBlock(int i)
{
  return(DecodedVoiceSubBlock[i]);
}

// Function to convert the LPC coefficients into LSF coefficients.
// The new LSF coefficients are stored in the attribute CoefLSF_from_LPC
void oss_celp::CoefLPC_to_CoefLSF(void)
{
  void cheby1(double *g, int ord);
  void cacm283(double a[], int ord, double r[]);
  double g1[100], g2[100];
  double g1r[100], g2r[100];
  int even;
  int g1_order, g2_order;
  int orderd2;

  int i, j;
  /*int swap;*/
  double Factor;
```

```
  /* Compute the lengths of the x polynomials. */

  even = (NumCoef & 1) == 0;  /* True if order is even. */
  if (even)
    g1_order = g2_order = NumCoef/2;
  else
    {
      fprintf(stderr,"Odd order not implemented yet\n");
      exit(1);
      g1_order = (NumCoef+1)/2;
      g2_order = g1_order - 1;
    }

  /* Compute the first half of K & R F1 & F2 polynomials. */

  /* Compute half of the symmetric and antisymmetric polynomials. */
  /* Remove the roots at +1 and -1. */

  orderd2=(NumCoef+1)/2;
  g1[orderd2] = 1;
  for(i=1;i<=orderd2;i++) g1[g1_order-i] = -(CoefLPC[NumCoef-i]+CoefLPC[i-1]);
  g2[orderd2] = 1;
  for(i=1;i<=orderd2;i++) g2[orderd2-i] = CoefLPC[NumCoef-i]-CoefLPC[i-1];

  if(even)
    {
      for(i=1; i<=orderd2;i++) g1[orderd2-i] -= g1[orderd2-i+1];
      for(i=1; i<=orderd2;i++) g2[orderd2-i] += g2[orderd2-i+1];
    }

  else
    for(i=2; i<=orderd2;i++)
      g2[orderd2-i] += g2[orderd2-i+2];   /* Right? */

  /* Convert into polynomials in cos(alpha) */

  cheby1(g1,g1_order);
  cheby1(g2,g2_order);
  Factor = 0.5;

  /* Find the roots of the 2 even polynomials.*/

  cacm283(g1,g1_order,g1r);
  cacm283(g2,g2_order,g2r);

  /* Convert back to angular frequencies in the range 0 to pi. */

  for(i=0, j=0 ; ; )
    {
      CoefLSF[j++] = acos(Factor * g1r[i]);
      if(j >= NumCoef) break;
      CoefLSF[j++] = acos(Factor * g2r[i]);
      if(j >= NumCoef) break;
      i++;
    }
}

// Function to convert the LSF coefficients into LPC coefficients.
// The new LPC coefficients are stored in the attribute CoefLPC_from_LSF
void oss_celp::CoefLSF_to_CoefLPC(void)
{
  // Necessary function prototype
  void polimulti(double poli1[], double poli2[], double polir[], int ordem1, int ordem2);

  // Local variables
  double pa1[3], pa2[3], pa3[3], pa4[3], pa5[3], pa6[5], pa7[5], pa8[9];
  double qa1[3], qa2[3], qa3[3], qa4[3], qa5[3], qa6[5], qa7[5], qa8[9];
  double p[11], q[11];
  int j;

  // Calculating the Q(z) and P(z) polynomials
```

```
  pa1[0] = pa1[2] = pa2[0] = pa2[2] = pa3[0] = pa3[2] = pa4[0] = pa4[2] = pa5[0] = pa5[2]
= 1.0;
  qa1[0] = qa1[2] = qa2[0] = qa2[2] = qa3[0] = qa3[2] = qa4[0] = qa4[2] = qa5[0] = qa5[2]
= 1.0;

  pa1[1] = -2*cos(CoefLSF[0]);
  pa2[1] = -2*cos(CoefLSF[2]);
  pa3[1] = -2*cos(CoefLSF[4]);
  pa4[1] = -2*cos(CoefLSF[6]);
  pa5[1] = -2*cos(CoefLSF[8]);

  qa1[1] = -2*cos(CoefLSF[1]);
  qa2[1] = -2*cos(CoefLSF[3]);
  qa3[1] = -2*cos(CoefLSF[5]);
  qa4[1] = -2*cos(CoefLSF[7]);
  qa5[1] = -2*cos(CoefLSF[9]);

  polimulti(pa1, pa2, pa6, 2, 2);
  polimulti(pa3, pa4, pa7, 2, 2);
  polimulti(pa6, pa7, pa8, 4, 4);
  polimulti(pa8, pa5, p, 8, 2);

  polimulti(qa1, qa2, qa6, 2, 2);
  polimulti(qa3, qa4, qa7, 2, 2);
  polimulti(qa6, qa7, qa8, 4, 4);
  polimulti(qa8, qa5, q, 8, 2);

  // Calculating the LPCs
  for (j = 1; j <= NumCoef; j++)
    CoefLPC_from_LSF[j - 1] = -(p[j] + p[j - 1] + q[j] - q[j - 1]) / 2;
}

// Function to convert the LSF coefficients into LPC coefficients.
// The new LPC coefficients are stored in the attribute CoefLPC_from_LSF
void oss_celp::CoefLSF_Interpolated_to_CoefLPC(void)
{
  // Necessary function prototype
  void polimulti(double poli1[], double poli2[], double polir[], int ordem1, int ordem2);

  // Local variables
  double pa1[3], pa2[3], pa3[3], pa4[3], pa5[3], pa6[5], pa7[5], pa8[9];
  double qa1[3], qa2[3], qa3[3], qa4[3], qa5[3], qa6[5], qa7[5], qa8[9];
  double p[11], q[11];
  int j;

  // Calculating the Q(z) and P(z) polynomials
  pa1[0] = pa1[2] = pa2[0] = pa2[2] = pa3[0] = pa3[2] = pa4[0] = pa4[2] = pa5[0] = pa5[2]
= 1.0;
  qa1[0] = qa1[2] = qa2[0] = qa2[2] = qa3[0] = qa3[2] = qa4[0] = qa4[2] = qa5[0] = qa5[2]
= 1.0;

  pa1[1] = -2*cos(CoefLSF_Interpolated[0]);
  pa2[1] = -2*cos(CoefLSF_Interpolated[2]);
  pa3[1] = -2*cos(CoefLSF_Interpolated[4]);
  pa4[1] = -2*cos(CoefLSF_Interpolated[6]);
  pa5[1] = -2*cos(CoefLSF_Interpolated[8]);

  qa1[1] = -2*cos(CoefLSF_Interpolated[1]);
  qa2[1] = -2*cos(CoefLSF_Interpolated[3]);
  qa3[1] = -2*cos(CoefLSF_Interpolated[5]);
  qa4[1] = -2*cos(CoefLSF_Interpolated[7]);
  qa5[1] = -2*cos(CoefLSF_Interpolated[9]);

  polimulti(pa1, pa2, pa6, 2, 2);
  polimulti(pa3, pa4, pa7, 2, 2);
  polimulti(pa6, pa7, pa8, 4, 4);
  polimulti(pa8, pa5, p, 8, 2);

  polimulti(qa1, qa2, qa6, 2, 2);
  polimulti(qa3, qa4, qa7, 2, 2);
  polimulti(qa6, qa7, qa8, 4, 4);
```

```
  polimulti(qa8, qa5, q, 8, 2);

  // Calculating the LPCs
  for (j = 1; j <= NumCoef; j++)
    CoefLPC_from_LSF_Interpolated[j - 1] = -(p[j] + p[j - 1] + q[j] - q[j - 1]) / 2;
}

// Function to make the LPC analisys for the block defined by the samples
// contained in the "data" vector. In that case, it is considered that the
// block is of the same size as the vector. The whole vector is one block.
void oss_celp::AnalysisLPC(void)
{
  // Local variables
  int i, j;
  double aux, Inverse_EMPQ, *corr, *ao, *k;
  float win_binomial[11] = {1.0, 0.999375390381012, 0.997503900885916, 0.994392535006671,
                            0.990052916711755, 0.984501218113371, 0.977758059085197,
0.969848379702430,
                            0.960801286608190, 0.950649874630354, 0.939431025178194};

  if ((corr = (double*)calloc(NumCoef + 1, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Error on allocating memory for the auto-correlation sequence.\n");
      exit(1);
    }

  if ((k = (double*)calloc(NumCoef, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Error on allocating memory for the gain vector.\n");
      exit(1);
    }

  if ((ao = (double*)calloc(NumCoef, sizeof(double))) == NULL)
    {
      fprintf(stderr, "Error on allocating memory for the coefficients vector.\n");
      exit(1);
    }

  // Determine the auto-correlation for the voice block
  for (j = 0; j <= NumCoef; j++)
    {
      corr[j] = 0.0;
      for (i = j; i < Length + 1; i++)
        corr[j] += (double)(WindowSignal[i] * WindowSignal[i - j]);
      corr[j] *= win_binomial[j];
    }

  // Calculating the LPCs using the recursive algorithm
  // of Levinson-Durbin
  ao[0] = k[0] = (corr[1] / corr[0]);
  Inverse_EMPQ = corr[0] * (1 - k[0] * k[0]);
  for (i = 1; i < NumCoef; i++)
    {
      aux = 0.0;
      for (j = 0; j <= i - 1; j++)
        aux += ao[j] * corr[i - j];
      k[i] = (corr[i + 1] - aux) / Inverse_EMPQ;
      CoefLPC[i] = k[i];
      for (j = 0; j <= i - 1; j++)
        CoefLPC[j] = ao[j] - k[i] * ao[i - j - 1];
      Inverse_EMPQ *= (1 - k[i] * k[i]);
      for (j = 0; j <= i; j++)
        ao[j] = CoefLPC[j];
    }
  free(corr);
  free(k);
  free(ao);
}

// Function to save the LSF coefficients from the last block
void oss_celp::SaveCoefLSF_before(void)
```

```
{
  for(int i = 0; i < NumCoef; i++)
    CoefLSF_before[i] = CoefLSF[i];
}

// Function to interpolate the LSF coefficients of the current block with
// the ones of the last block
void oss_celp::InterpolateCoefLSF(int NumRelativeSubBlock)
{
  double q[4];
  double aux;

  q[0] = 0.25;
  q[1] = 0.5;
  q[2] = 0.75;
  q[3] = 1.0;

  // Here I actually make the interpolation
  for (int i = 0; i < NumCoef; i++)
    {
      aux = (1 - q[NumRelativeSubBlock - 1]) * CoefLSF_before[i] +
                q[NumRelativeSubBlock - 1]  * CoefLSF[i];
      CoefLSF_Interpolated[i] = aux;
    }
}

// Function to make the quantization of the LSF coefficients
void oss_celp::QuantLSF(void)
{
  // A necessary function prototype
  float quant_esc(double x, float partition[], float codebook[], int codebook_size);

  // Quantization
  CoefDLSF[0] = quant_esc(CoefLSF[0], part0, cb0, 16);
  CoefLSF_quant[0] = CoefDLSF[0];

  CoefDLSF[1] = quant_esc(CoefLSF[1]- CoefLSF_quant[0], part1, cb1, 16);
  CoefLSF_quant[1] = CoefLSF_quant[0] + CoefDLSF[1];

  CoefDLSF[2] = quant_esc(CoefLSF[2]- CoefLSF_quant[1], part2, cb2, 16);
  CoefLSF_quant[2] = CoefLSF_quant[1] + CoefDLSF[2];

  CoefDLSF[3] = quant_esc(CoefLSF[3]- CoefLSF_quant[2], part3, cb3, 16);
  CoefLSF_quant[3] = CoefLSF_quant[2] + CoefDLSF[3];

  CoefDLSF[4] = quant_esc(CoefLSF[4]- CoefLSF_quant[3], part4, cb4, 16);
  CoefLSF_quant[4] = CoefLSF_quant[3] + CoefDLSF[4];

  CoefDLSF[5] = quant_esc(CoefLSF[5]- CoefLSF_quant[4], part5, cb5, 16);
  CoefLSF_quant[5] = CoefLSF_quant[4] + CoefDLSF[5];

  CoefDLSF[6] = quant_esc(CoefLSF[6]- CoefLSF_quant[5], part6, cb6, 16);
  CoefLSF_quant[6] = CoefLSF_quant[5] + CoefDLSF[6];

  CoefDLSF[7] = quant_esc(CoefLSF[7]- CoefLSF_quant[6], part7, cb7, 16);
  CoefLSF_quant[7] = CoefLSF_quant[6] + CoefDLSF[7];

  CoefDLSF[8] = quant_esc(CoefLSF[8]- CoefLSF_quant[7], part8, cb8, 16);
  CoefLSF_quant[8] = CoefLSF_quant[7] + CoefDLSF[8];

  CoefDLSF[9] = quant_esc(CoefLSF[9]- CoefLSF_quant[8], part9, cb9, 16);
  CoefLSF_quant[9] = CoefLSF_quant[8] + CoefDLSF[9];
}

// Function to make the search in the Adaptive Codebook
// and fill up the attributes Index_AC and Gain_AC
// for each voice sub-block
void oss_celp::Search_AC(int k)
{
  // Necessary function prototype
  void read_sequence(double AdaptiveCodebook[], double Reading_AdaptiveCodebook[],
```

```
                    int index, int size_ad, int size_seq);
  double inner_prod(double vector1[], double vector2[], int size);
  void filt_sp2(double s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size);

  // Local variables
  double error = 0.0;
  int NonZeroCorrelationCounter = 0;
  double Correlation_TargetSignal_SequenceAC;
  double AutoCorrelation_SequenceAC;
  double Correlation_TargetSignal_ResponseAC;
  double AutoCorrelation_ResponseAC;
  double Inverse_EMPQ;

  // Searching for the sequence which minimize the EMPQ
  for(int i=1; i<GetACSize(); i++)
    {
      // Reading the candidate sequence from the AC which has already gone into the
synthesis filter
      read_sequence(Perc_AdaptiveCodebook, Sequence_AC, i, GetNumberOfSamples_AC(),
GetBlockSize()/4);

      // Calculating the index which minimizes the EMPQ using the candidate sequence from
the AC
      Correlation_TargetSignal_SequenceAC = inner_prod(TargetSignal, Sequence_AC,
GetBlockSize()/4);
      if (Correlation_TargetSignal_SequenceAC > 0)
       {
         NonZeroCorrelationCounter++;
         AutoCorrelation_SequenceAC = inner_prod(Sequence_AC, Sequence_AC,
GetBlockSize()/4);
         Inverse_EMPQ = Correlation_TargetSignal_SequenceAC *
Correlation_TargetSignal_SequenceAC /
           AutoCorrelation_SequenceAC;
         // Comparing the EMPQ to the last and retains the smaller one, along with the
index
         if (Inverse_EMPQ > error)
           {
             Index_AC[k-1] = i;
             error = Inverse_EMPQ;
           }
       }
    }

  // If the search was successful, then...
  if (NonZeroCorrelationCounter != 0)
    {
      // Calculating the best excitation from the AC
      read_sequence(AdaptiveCodebook, Reading_AC, Index_AC[k-1], GetNumberOfSamples_AC(),
GetBlockSize()/4);

      // Calculating the response to the best excitation from the AC
      filt_sp2(Reading_AC, Response_AC, Den_PercFilter, GetBlockSize()/4, GetNumCoef());

      // Calculating the gain for the AC
      Correlation_TargetSignal_ResponseAC = inner_prod(TargetSignal, Response_AC,
GetBlockSize()/4);
      AutoCorrelation_ResponseAC = inner_prod(Response_AC, Response_AC,
GetBlockSize()/4);
      Gain_AC[k-1] = Correlation_TargetSignal_ResponseAC / AutoCorrelation_ResponseAC;
    }

  else
    {
      for(int i = 0; i < GetBlockSize()/4; i++)
       Sequence_AC[i] = Response_AC[i] = 0.0;
      Gain_AC[k-1] = 0.0;
    }
}

// Function to make the search in the Fixed Codebook
```

```
// and fill up the attributes Index_FC and Gain_FC
// for each voice sub-block
void oss_celp::Search_FC(int k)
{
  // Necessary function prototype
  double inner_prod(double vector1[], double vector2[], int size);
  void filt_sp2(double s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size);

  // Local variables
  double error = 0.0;
  int increment_df = 0;
  int NonZeroCorrelationCounter = 0;
  double Correlation_TargetSignal_SequenceFC;
  double AutoCorrelation_SequenceFC;
  double Correlation_TargetSignal_ResponseFC;
  double AutoCorrelation_ResponseFC;
  double Inverse_EMPQ;

  // Searching for the sequence which minimize the EMPQ
  for(int i=1; i<=GetFCSize(); i++)
    {
      // Reading the candidate sequence from the FC
      for(int j=0; j<GetBlockSize()/4; j++)
       Sequence_FC[j] = Perc_FixedCodebook[increment_df + j];

      // Calculating the index which minimizes the EMPQ using the candidate sequence from
the FC

      Correlation_TargetSignal_SequenceFC = inner_prod(TargetSignal, Sequence_FC,
GetBlockSize()/4);

      if (Correlation_TargetSignal_SequenceFC > 0)
        {
          NonZeroCorrelationCounter++;
          AutoCorrelation_SequenceFC = inner_prod(Sequence_FC, Sequence_FC,
GetBlockSize()/4);
          Inverse_EMPQ = Correlation_TargetSignal_SequenceFC *
Correlation_TargetSignal_SequenceFC /
            AutoCorrelation_SequenceFC;
          // Comparing the EMPQ to the last and retains the smaller one, along with the
index
          if (Inverse_EMPQ > error)
            {
              Index_FC[k-1] = i;
              error = Inverse_EMPQ;
            }
        }

      // Incrementing "increment_df"
      // So that the FC would have a step of 2 units
      increment_df += 2;
    }

  if (NonZeroCorrelationCounter != 0)
    {
      // Reading the best excitation from the FC
      for(int w=0; w<GetBlockSize()/4; w++)
       Reading_FC[w] = FixedCodebook[2*Index_FC[k-1] + w];

      // Calculating the response to the best excitation from the FC
      filt_sp2(Reading_FC, Response_FC, Den_PercFilter, GetBlockSize()/4, GetNumCoef());

      // Calculating the gain for the FC
      Correlation_TargetSignal_ResponseFC = inner_prod(TargetSignal, Response_FC,
GetBlockSize()/4);
      AutoCorrelation_ResponseFC = inner_prod(Response_FC, Response_FC,
GetBlockSize()/4);
      Gain_FC[k-1] = Correlation_TargetSignal_ResponseFC / AutoCorrelation_ResponseFC;
    }
```

```
  else
    {
      for(int i = 0; i < GetBlockSize()/4; i++)
       Sequence_FC[i] = Response_FC[i] = 0.0;
      Gain_FC[k-1] = 0.0;
    }
}

// Function to update the target signal after the search in the
// AC and before the search in the FC
void oss_celp::UpdateTargetSignal(int k)
{
  for(int i=0; i<GetBlockSize()/4; i++)
    TargetSignal[i] -= Gain_AC[k-1] *  Response_AC[i];
}

// Function to update the AdaptiveCodebook
void oss_celp::UpdateAdaptiveCodebook(void)
{
  for(int i=0; i<GetNumberOfSamples_AC(); i++)
    {
      if (i < (GetNumberOfSamples_AC()-(GetBlockSize()/4)))
        {
          AdaptiveCodebook[i] = AdaptiveCodebook[i + (GetBlockSize()/4)];
        }
      else
        {
          AdaptiveCodebook[i] = CompleteExcitation[i - (GetNumberOfSamples_AC() -
(GetBlockSize()/4))];
        }
    }
}

// Function to receive the decoded voice sub block
// from another oss_celp object in a certain sub block
void oss_celp::ReceiveDecodedVoiceSubBlock(oss_celp Other)
{
  SAMPLE* DataAux;

  // I allocate memory for the new audio data vector
  DataAux = new SAMPLE[Other.GetBlockSize()/4];
  if (!DataAux)
    cout << "Error on memory alloc." << endl;

  // So, I fill the new vector up
  for (int i=0; i < Other.GetBlockSize()/4; i++)
    {
      DataAux[i] = (SAMPLE)(Other.GetDecodedVoiceSubBlock(i));
    }

  // If the audio data pointer is defined,
  // I should delete it
  if (Data)
    delete [] Data;

  // I allocate memory for the new vector
  Data = new SAMPLE[Other.GetBlockSize()/4];
  if (!Data)
    cout << "Error on memory alloc." << endl;

  // I, then, make one vector equal to the other one
  Data = DataAux;
  // Finally I refresh the audio data vector length
  SetLength(Other.GetBlockSize()/4);
}

// Function to equals the states of the synthesis filter
void oss_celp::Zero_SynthFilter_States(void)
{
  for(int i=0; i<GetNumCoef(); i++)
    {
```

```
      States_SynthFilter[i] = 0.0;
      States_SynthFilter_Decod[i] = 0.0;
    }
}

// Function to receive info about the coefficients from other oss_celp object
// in order to decode the voice signal
void oss_celp::ReceiveCoefInfo(oss_celp other)
{
  for(int i=0; i < other.GetNumCoef(); i++)
    CoefLSF[i] = other.GetCoefLSF(i);
}

// Function to receive info about the codebooks from other oss_celp object
// in order to decode the voice signal
void oss_celp::ReceiveCodebookInfo(oss_celp other, int k)
{
  Index_AC[k-1] = other.Index_AC[k-1];
  Index_FC[k-1] = other.Index_FC[k-1];
  Gain_AC[k-1] = other.Gain_AC[k-1];
  Gain_FC[k-1] = other.Gain_FC[k-1];
}

// Function to apply a Hamming window to the voice signal in order to
// make the LPC analisys
void oss_celp::ApplyWindow(void)
{
  void hamming(double janela[], int tamanho);
  hamming(Window, Length + 1);

  for (int i=0; i < Length+1; i++)
    WindowSignal[i] = Window[i] * Data[i];
}

// Function to aaply the perceptual filter to the FC
void oss_celp::ApplySynthFilter_to_FC(void)
{
  // Necessary function prototype
  void filt_sp3(float s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size);

  // Saving the filter input
  for(int i=0; i<GetNumberOfSamples_AC(); i++)
    SavedAdaptiveCodebook[i] = AdaptiveCodebook[i];

  int RealFCSize = 2*GetFCSize() + (GetBlockSize()/4);
  filt_sp3(FixedCodebook, Perc_FixedCodebook, Den_PercFilter,
          RealFCSize, GetNumCoef());

  // Loading the filter input
  for(int i=0; i<GetNumberOfSamples_AC(); i++)
    AdaptiveCodebook[i] = SavedAdaptiveCodebook[i];
}

// Function to apply the perceptual filter to the
// voice sub-block
void oss_celp::ApplyPercFilter_to_VoiceSubBlock(void)
{
  // Necessay function prototype
  void filt_pz(double s_input[], double s_output[], double num_filter[], double
den_filter[],
             double buffer_e[], double buffer_s[], int signal_size, int buffer_size);

  filt_pz(VoiceSubBlock, Perc_VoiceSubBlock, Num_PercFilter, Den_PercFilter,
         States_Num_PercFilter, States_Den_PercFilter, GetBlockSize()/4, GetNumCoef());
}

// Function to aaply the synthesis filter to the AC
void oss_celp::ApplySynthFilter_to_AC(void)
{
  // Necessary function prototype
```

```
  void filt_sp2(double s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size);

  // Saving the filter input
  for(int i=0; i<GetNumberOfSamples_AC(); i++)
    SavedAdaptiveCodebook[i] = AdaptiveCodebook[i];

  filt_sp2(AdaptiveCodebook, Perc_AdaptiveCodebook, Den_PercFilter,
          GetNumberOfSamples_AC(), GetNumCoef());

  // Loading the filter input
  for(int i=0; i<GetNumberOfSamples_AC(); i++)
    AdaptiveCodebook[i] = SavedAdaptiveCodebook[i];
}

// Function to show the LPC coefficients
void oss_celp::ShowCoefLPC(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LPC[" << i << "]=" << GetCoefLPC(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the LSF coefficients
void oss_celp::ShowCoefLSF(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LSF[" << i << "]=" << GetCoefLSF(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the DLSF coefficients
void oss_celp::ShowCoefDLSF(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "DLSF[" << i << "]=" << GetCoefDLSF(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the LPC coefficients that come from the
// LSF coefficients
void oss_celp::ShowCoefLPC_from_LSF(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LPC[" << i << "]=" << GetCoefLPC_from_LSF(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the LSF coefficients from the last block
void oss_celp::ShowCoefLSF_before(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LSF_b[" << i << "]=" << GetCoefLSF_before(i) << "; ";
    }
  cout << endl;
  cout << endl;
}
```

```
// Function to show the LSF coefficients that have been interpolated
void oss_celp::ShowCoefLSF_Interpolated(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LSF_i[" << i << "]=" << GetCoefLSF_Interpolated(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the LSF coefficients that have been interpolated
void oss_celp::ShowCoefLPC_from_LSF_Interpolated(void)
{
  for(int i=0; i < GetNumCoef(); i++)
    {
      cout << "LPC_i[" << i << "]=" << GetCoefLPC_from_LSF_Interpolated(i) << "; ";
    }
  cout << endl;
  cout << endl;
}

// Function to show the coefficients of the Perceptual Filter
void oss_celp::ShowPercFilter(void)
{
  for(int i=0; i<=1; i++)
    {
      for(int j=0; j<=GetNumCoef(); j++)
        {
          if (i==0)
            cout << " W_num(" << j <<")=" << GetPercFilter(i, j);

          if (i==1)
            cout << " W_den(" << j <<")=" << GetPercFilter(i, j);
        }
      cout << endl;
    }
  cout << endl;
  cout << endl;
}

// Function to show the samples of the target signal
void oss_celp::ShowTargetSignal(void)
{
  for(int i=0; i<(GetBlockSize()/4); i++)
    cout << " TS[" << i << "]=" << GetTargetSignal(i);
  cout << endl;
  cout << endl;
}

// Function to show the index of the AC which presents
// the best excitation
void oss_celp::ShowIndexAC(int k)
{
  cout << "Index_AC[" << k << "]=" << GetIndexAC(k-1) << endl;
}

// Function to show the gain of the AC which belongs to
// the best excitation
void oss_celp::ShowGainAC(int k)
{
  cout << "Gain_AC[" << k << "]=" << GetGainAC(k-1) << endl;
}

// Function to show the index of the AC which presents
// the best excitation
void oss_celp::ShowIndexFC(int k)
{
  cout << "Index_FC[" << k << "]=" << GetIndexFC(k-1) << endl;
}
```

```
// Function to show the gain of the FC which belongs to
// the best excitation
void oss_celp::ShowGainFC(int k)
{
  cout << "Gain_FC[" << k << "]=" << GetGainFC(k-1) << endl;
}

// Function to show the decoded voice sub block
void oss_celp::ShowDecodedVoiceSubBlock(void)
{
  cout << endl;
  for(int i=0; i<(GetBlockSize()/4); i++)
    cout << " Decoded[" << i << "]=" << (SAMPLE)GetDecodedVoiceSubBlock(i);
  cout << endl;
  cout << endl;
}

////////////////// FUNCTION BANK ////////////////////////

/*--------------------------------------------------------------------------*/
/* Function: polimulti()                                                    */
/* Goal: multiply two polynomials                                           */
/* Return: none.                                                            */
/*--------------------------------------------------------------------------*/

void polimulti(double poli1[], double poli2[], double polir[], int ordem1, int ordem2)
{
  // Local variables
  int i, j;

  // Multiplying the polynomials
  for (j = 0; j <= (ordem1 + ordem2); j++)
    {
      polir[j] = 0;
      for (i = 0; i <= ordem1; i++)
        {
          if ((j - i) >= 0 && (j - i) <= ordem2)
            polir[j] += poli1[i] * poli2[j - i];
        }
    }
}

void cheby1(double *g, int ord)
{
  int i, j;
  /*int k;*/

  for(i=2; i<= ord; i++)
    {
      for(j=ord; j > i; j--)
        g[j-2] -= g[j];
      g[j-2] -= 2.0*g[j];
    }
}

void cacm283(double a[], int ord, double r[])
{
  int i, k;
  double val, p, delta, error;
  double rooti;
  int swap;

  for(i=0; i<ord;i++)
    r[i] = 2.0 * (i+0.5) / ord - 1.0;

  for(error=1 ; error > 1.e-12; )
    {
      error = 0;
      for( i=0; i<ord; i++)
        {
          /* Update each point. */
```

```
        rooti = r[i];
        val = a[ord];
        p = a[ord];
        for(k=ord-1; k>= 0; k--)
          {
            val = val * rooti + a[k];
            if (k != i) p *= rooti - r[k];
          }
        delta = val/p;
        r[i] -= delta;
        error += delta*delta;
      }
  }

  /* Do a simple bubble sort to get them in the right order. */

  do
    {
      double tmplsp;
      swap = 0;
      for(i=0; i<ord-1;i++)
        {
          if(r[i] < r[i+1])
            {
              tmplsp = r[i];
              r[i]=r[i+1];
              r[i+1]=tmplsp;
              swap++;
            }
        }
    }
  while (swap > 0);
}

/*--------------------------------------------------------------------------*/
/* Function: hamming()                                                      */
/* Goal: obtain a Hamming window of the specified size                      */
/* Return: none.                                                            */
/*--------------------------------------------------------------------------*/

void hamming(double janela[], int tamanho)
{
  // Local variable
  int j;
  double pi;

  // Calculating PI
  pi = 4.0 * atan(1.0);

  // Calculating the window
  for (j = 0; j < tamanho; j++)
    janela[j] = 0.54 - 0.46 * cos((2.0 * pi * j) / (tamanho - 1));
}

/*--------------------------------------------------------------------------*/
/* Function: quant_esc()                                                    */
/* Goal: make the quantization for a determined scalar, given               */
/*             the partition vectors and the codebook.                      */
/* Return: quantized value                                                  */
/*--------------------------------------------------------------------------*/

float quant_esc(double x, float particao[], float codebook[], int tam_codebook)
{
  // Local variables
  int i, j;
  float y;

  // Initializing the codebook index
  i = 0;

  // Calculating the quantized value
```

```
  for (j=0; j < tam_codebook-1; j++)
    {
      if (x > particao[j])
       i = j + 1;
    }
  y = codebook[i];

  // Returning the calculated value
  return y;
}

/*--------------------------------------------------------------------*/
/* Function: filt_pz()                                                */
/* Goal: filter a signal using a digital filter which has poles and */
/*       zeros, considering the initial state                         */
/* Return: none.                                                      */
/*--------------------------------------------------------------------*/

void filt_pz(double s_input[], double s_output[], double num_filter[], double
den_filter[],
             double buffer_e[], double buffer_s[], int signal_size, int buffer_size)
{
  // Local variables
  int i, j;
  double acum_e, acum_s;

  // Begin the filtering
  for (j = 0; j < signal_size; j++)
    {
      // Multiply the samples contained in the buffers by the coefficients
      acum_e = acum_s = 0.0;
      for (i = 0; i < buffer_size; i++)
        {
          acum_e += buffer_e[i] * num_filter[i + 1];
          acum_s += buffer_s[i] * den_filter[i + 1];
        }

      // Calculating output
      s_output[j] = num_filter[0] * s_input[j] + acum_e - acum_s;
      s_output[j] /= den_filter[0];

      // Updating buffers
      for (i = buffer_size - 1; i > 0; i--)
        {
          buffer_e[i] = buffer_e[i-1];
          buffer_s[i] = buffer_s[i-1];
        }
      buffer_e[0] = s_input[j];
      buffer_s[0] = s_output[j];
    }
}


/*--------------------------------------------------------------------*/
/* Function: filt_sp1()                                               */
/* Goal: filter a signal using a digital filter which has only poles */
/*       considering the initial state                                */
/* Return: none.                                                      */
/*--------------------------------------------------------------------*/

void filt_sp1(double s_input[], double s_output[], double den_filter[],
              double buffer[], int signal_size, int buffer_size)
{
  // Local variables
  int i, j;
  double acum;

  // Begin the filtering
  for (j = 0; j < signal_size; j++)
    {
      // Multiply the samples contained in the buffers by the coefficients
```

```
      acum = 0.0;
      for (i = 0; i < buffer_size; i++)
        acum += buffer[i] * den_filter[i + 1];

      // Calculating output
      s_output[j] = s_input[j] - acum;
      s_output[j] /= den_filter[0];

      // Updating buffers
      for (i = buffer_size - 1; i > 0; i--)
        buffer[i] = buffer[i-1];
      buffer[0] = s_output[j];
    }
}

/*--------------------------------------------------------------------*/
/* Function: filt_sp2()                                               */
/* Goal: filter a signal using a digital filter which has poles and */
/*       zeros, without considering the initial state                 */
/* Return: none.                                                      */
/*--------------------------------------------------------------------*/

void filt_sp2(double s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size)
{
  // Local variables
  int i, j;
  double acum, *buffer;

  // Allocating memory
  if ((buffer = (double*)calloc(buffer_size, sizeof(double))) == NULL)
    {
      fprintf(stderr,"Error on allocating memory.\n");
      exit(1);
    }

  // Begin the filtering
  for (j = 0; j < signal_size; j++)
    {
      // Multiply the samples contained in the buffers by the coefficients
      acum = 0.0;
      for (i = 0; i < buffer_size; i++)
        acum += buffer[i] * den_filter[i + 1];

      // Calculating output
      s_output[j] = s_input[j] - acum;
      s_output[j] /= den_filter[0];

      // Updating buffer
      for (i = buffer_size - 1; i > 0; i--)
        buffer[i] = buffer[i-1];
      buffer[0] = s_output[j];
    }
}

/*--------------------------------------------------------------------*/
/* Function: filt_sp3()                                               */
/* Goal: filter a signal using a digital filter which has poles and */
/*       zeros, without considering the initial state                 */
/* Return: none.                                                      */
/*--------------------------------------------------------------------*/

void filt_sp3(float s_input[], double s_output[], double den_filter[],
              int signal_size, int buffer_size)
{
  // Local variables
  int i, j;
  double acum, *buffer;

  // Allocating memory
  if ((buffer = (double*)calloc(buffer_size, sizeof(double))) == NULL)
```

```
    {
      fprintf(stderr,"Error on allocating memory.\n");
      exit(1);
    }

  // Begin the filtering
  for (j = 0; j < signal_size; j++)
    {
      // Multiply the samples contained in the buffers by the coefficients
      acum = 0.0;
      for (i = 0; i < buffer_size; i++)
       acum += buffer[i] * den_filter[i + 1];

      // Calculating output
      s_output[j] = s_input[j] - acum;
      s_output[j] /= den_filter[0];

      // Updating buffers
      for (i = buffer_size - 1; i > 0; i--)
       buffer[i] = buffer[i-1];
      buffer[0] = s_output[j];
    }
}

/*--------------------------------------------------------------------*/
/* Function: inner_prod()                                             */
/* Goal: calculate the inner product between two vectors.             */
/* Return: inner product value.                                       */
/*--------------------------------------------------------------------*/

double inner_prod(double vector1[], double vector2[], int size)
{
  // Local variables
  int j;
  double prod;

  // Calculating the inner product
  prod = 0.0;
  for (j = 0; j < size; j++)
    prod += vector1[j] * vector2[j];

  // Return its value
  return prod;
}

/*--------------------------------------------------------------------*/
/* Function: load_from_file()                                         */
/* Goal: reads from data file and loads a especified variable         */
/* Return: none                                                       */
/*--------------------------------------------------------------------*/

void load_from_file(const char file[], float variable[], int num_itens)
{
  // Local variable
  FILE *fp;

  // Opens binary file
  if ((fp = fopen(file,"rb")) == NULL)
    {
      fprintf(stderr,"Error on opening data file.\n");
      exit(1);
    }

  // Reads data from file
  if (fread(variable, sizeof(float), num_itens, fp) != num_itens)
    {
      fprintf(stderr,"Error on reading from data file.\n");
      exit(1);
    }

  // Closes data file
```

```
   fclose(fp);

}

/*-------------------------------------------------------------------------*/
/* Function: read_sequence()                                               */
/* Goal: read a candidate sequence from the adaptive dicionary using       */
/*       fraccionary delays                                                */
/* Return: none.                                                           */
/*-------------------------------------------------------------------------*/

void read_sequence(double AdaptiveCodebook[], double Reading_AdaptiveCodebook[], int
index, int size_ad, int size_seq)
{
  // Local variables
  int i, delay;

  for(int i=0; i < size_seq; i++)
    Reading_AdaptiveCodebook[i] = AdaptiveCodebook[index + i];
}
```

C.5 OSS_QUALITY.H

```
// OSS_QUALITY.H
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 19/3/2003
// Version: 1.0

class oss_quality
{
 public:
  oss_quality(int Size);
  int GetLength(void);

  void SetSignal(int Index, double Value);
  void SetEstimatedSignal(int Index, double Value);
  void AccumulateSignalPower(void);
  void AccumulateErrorPower(void);
  void ZeroSignalPower(void);
  void ZeroErrorPower(void);
  double GetSNR(void);

  void SetNumberOfSegments(int);
  int GetNumberOfSegments(void);
  void IncrementNumberOfSegments(void);

  void AccumulateSNR_seg(double);
  double GetSNR_seg(void);

  void SetNumCoef(int);
  int GetNumCoef(void);
  void SetNumPercFilter(int Index, double Value);
  void SetDenPercFilter(int Index, double Value);
  void ZeroStatesPercFilter(void);
  void AccumulatePercErrorPower(void);
  void ZeroPercErrorPower(void);
  double GetSNR_perc(void);
  double GetSNR_seg_perc(void);

private:
  int Length;
  int NumberOfSegments;
  double *Signal;
  double *EstimatedSignal;
  double *ErrorSignal;
  double *PercErrorSignal;
  double SignalPower;
  double ErrorPower;
```

```
  double PercErrorPower;

  double SNR_seg;

  int NumCoef;
  double *NumPercFilter;
  double *DenPercFilter;
  double *StatesNumPercFilter;
  double *StatesDenPercFilter;
};
```

C.6 OSS_QUALITY.CPP

```
// OSS_QUALITY.CPP
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 19/3/2003
// Version: 1.0

#include <termios.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <limits.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>

#include "oss_quality.h"

// Constructor for the class
oss_quality::oss_quality(int Size)
{
  Length = Size;

  // Allocating memory for pointers
  Signal = new double[Size];
  if (!Signal)
    {
      cout << "Error on allocating memory for the signal." << endl;
      exit(1);
    }

  EstimatedSignal = new double[Size];
  if (!EstimatedSignal)
    {
      cout << "Error on allocating memory for the estimated signal." << endl;
      exit(1);
    }

  ErrorSignal = new double[Size];
  if (!ErrorSignal)
    {
      cout << "Error on allocating memory for the error signal." << endl;
      exit(1);
    }

  PercErrorSignal = new double[Size];
  if (!PercErrorSignal)
    {
      cout << "Error on allocating memory for the filtered error signal." << endl;
      exit(1);
    }
```

```
  SignalPower = 0;
  ErrorPower = 0;
  PercErrorPower = 0;

  NumberOfSegments = 0;
  SNR_seg = 0;
}

// Funtion to get the length of the signals
int oss_quality::GetLength(void)
{
  return(Length);
}

// Funtion to set the signal
void oss_quality::SetSignal(int Index, double Value)
{
  Signal[Index] = Value;
}

// Funtion to set the estimated signal
void oss_quality::SetEstimatedSignal(int Index, double Value)
{
  EstimatedSignal[Index] = Value;
}

// Function to acumulate the power of the error between the original and the decoded
signals
void oss_quality::AccumulateErrorPower(void)
{
  // Calculating the error between the decoded and the original signal
  for(int i=0; i<Length; i++)
    ErrorSignal[i] = Signal[i] - EstimatedSignal[i];

  // Acumulating...
  for(int i=0; i<Length; i++)
    ErrorPower += (ErrorSignal[i] * ErrorSignal[i]);
}

// Function to acumulate the power of the original signal
void oss_quality::AccumulateSignalPower(void)
{
  // Acumulating...
  for(int i=0; i<Length; i++)
    SignalPower += (Signal[i] * Signal[i]);
}

// Function to zero the signal power
void oss_quality::ZeroSignalPower(void)
{
  SignalPower = 0;
}

// Function to zero the error signal power
void oss_quality::ZeroErrorPower(void)
{
  ErrorPower = 0;
}

// Function to get the Signal Noise Ratio
double oss_quality::GetSNR(void)
{
  // Calculating the SNR
  return(10*log10(SignalPower/ErrorPower));
}

// Function to set the number of segments
void oss_quality::SetNumberOfSegments(int number)
{
  NumberOfSegments = number;
```

```
}

// Function to get the number of segments
int oss_quality::GetNumberOfSegments(void)
{
  return(NumberOfSegments);
}

// Function to increment the number of segments
void oss_quality::IncrementNumberOfSegments(void)
{
  NumberOfSegments++;
}

// Function to acumulate the power of the original signal
void oss_quality::AccumulateSNR_seg(double argSNR)
{
  // Acumulating...
  SNR_seg += argSNR;
}

// Function to get the Signal Noise Ratio Segmented
double oss_quality::GetSNR_seg(void)
{
  // Calculating the SNR segmented
  return(SNR_seg/NumberOfSegments);
}

// Function to set the number of coefficients
void oss_quality::SetNumCoef(int number)
{
  NumCoef = number;

  NumPercFilter = new double[number];
  if (!NumPercFilter)
    {
      cout << "Error on allocating memory for the Perceptual Filter Numerator." << endl;
      exit(1);
    }

  DenPercFilter = new double[number];
  if (!DenPercFilter)
    {
      cout << "Error on allocating memory for the Perceptual Filter Denominator." <<
endl;
      exit(1);
    }

  StatesNumPercFilter = new double[number];
  if (!StatesNumPercFilter)
    {
      cout << "Error on allocating memory for the Perceptual Filter Numerator States." <<
endl;
      exit(1);
    }

  StatesDenPercFilter = new double[number];
  if (!StatesDenPercFilter)
    {
      cout << "Error on allocating memory for the Perceptual Filter Denominator States."
<< endl;
      exit(1);
    }
}

// Function to get the number of coefficients
int oss_quality::GetNumCoef(void)
{
  return(NumCoef);
}
```

```
// Funtion to set the the Perceptual Filter numerator
void oss_quality::SetNumPercFilter(int Index, double Value)
{
  NumPercFilter[Index] = Value;
}

// Funtion to set the the Perceptual Filter denominator
void oss_quality::SetDenPercFilter(int Index, double Value)
{
  DenPercFilter[Index] = Value;
}

// Function to zero the states of the Perceptual Filter
void oss_quality::ZeroStatesPercFilter(void)
{
  for(int i=0; i<NumCoef; i++)
    {
      StatesNumPercFilter[i] = 0;
      StatesDenPercFilter[i] = 0;
    }
}

// Function to acumulate the power of the error between the original and the decoded
signals
void oss_quality::AccumulatePercErrorPower(void)
{
  // Necessay function prototype
  void filter(double s_input[], double s_output[], double num_filter[], double
den_filter[],
              double buffer_e[], double buffer_s[], int signal_size, int buffer_size);

  // Calculating the error between the decoded and the original signal
  for(int i=0; i<Length; i++)
    ErrorSignal[i] = Signal[i] - EstimatedSignal[i];

  // Filtering...
  filter(ErrorSignal, PercErrorSignal, NumPercFilter, DenPercFilter,
         StatesNumPercFilter, StatesDenPercFilter, Length, GetNumCoef());

  // Acumulating...
  for(int i=0; i<Length; i++)
    PercErrorPower += (PercErrorSignal[i] * PercErrorSignal[i]);
}

// Function to zero the error signal power
void oss_quality::ZeroPercErrorPower(void)
{
  PercErrorPower = 0;
}

// Function to get the Perceptual Signal Noise Ratio
double oss_quality::GetSNR_perc(void)
{
  // Calculating the SNR
  return(10*log10(SignalPower/PercErrorPower));
}

// Function to get the perceptual segmented SNR
double oss_quality::GetSNR_seg_perc(void)
{
  // Calculating the SNR segmented
  return(SNR_seg/NumberOfSegments);
}

/*-------------------------------------------------------------------*/
/* Function: filter()                                                */
/* Goal: filter a signal using a digital filter which has poles and */
/*       zeros, considering the initial state                        */
/* Return: none.                                                     */
/*-------------------------------------------------------------------*/
```

```
void filter(double s_input[], double s_output[], double num_filter[], double
den_filter[],
            double buffer_e[], double buffer_s[], int signal_size, int buffer_size)
{
  // Local variables
  int i, j;
  double acum_e, acum_s;

  // Begin the filtering
  for (j = 0; j < signal_size; j++)
    {
      // Multiply the samples contained in the buffers by the coefficients
      acum_e = acum_s = 0.0;
      for (i = 0; i < buffer_size; i++)
        {
          acum_e += buffer_e[i] * num_filter[i + 1];
          acum_s += buffer_s[i] * den_filter[i + 1];
        }

      // Calculating output
      s_output[j] = num_filter[0] * s_input[j] + acum_e - acum_s;
      s_output[j] /= den_filter[0];

      // Updating buffers
      for (i = buffer_size - 1; i > 0; i--)
        {
          buffer_e[i] = buffer_e[i-1];
          buffer_s[i] = buffer_s[i-1];
        }
      buffer_e[0] = s_input[j];
      buffer_s[0] = s_output[j];
    }
}
```

C.7 MAIN_CELP.CPP

```
// MAIN_CELP.CPP
// Author: Filipe Castello da Costa Beltrao Diniz
// Date: 19/3/2003
// Version: 1.0

// Including the necessary headers
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include "oss_celp.h"
#include "oss_quality.h"

// Declaring global variables
double window = 0.02; // The time duration of the voice block measured in seconds
int    samples = 160; // The number of samples for the voice block
int    fs = 8000;     // The sampling frequency for the voice signal: 8 kHz
int    nc = 1;        // The number of channels for the voice signal: mono

// Declaring the object for the audio device
oss_audio AudioDev(fs, nc);

// Declaring the objects for the CELP implementation
oss_celp BufferCod(AudioDev.GetSampleRate(), AudioDev.GetNumberOfChannels());
oss_celp BufferDecod(AudioDev.GetSampleRate(), AudioDev.GetNumberOfChannels());
oss_celp BufferPlay(AudioDev.GetSampleRate(), AudioDev.GetNumberOfChannels());

// Declaring the objects for the signal quality evaluation
oss_quality MeasureSNR(samples/4);
oss_quality MeasureSNRseg(samples/4);
```

```
oss_quality MeasureSNRperc(samples/4);

// Function MAIN
int main()
{
  cout << "Real-time C E L P Codec"  << endl;

  cout << "Initializing sound device...";
  AudioDev.Init();
  cout << "OK" << endl;

  // Setting up the number of coefficients for the oss_celp object
  // and the oss_quality object that will be responsible for the
  // perceptual SNR
  BufferCod.SetNumCoef(10);
  BufferDecod.SetNumCoef(10);
  MeasureSNRperc.SetNumCoef(10);

  double gamma = 0.8;   // The gamma factor for the perceptual filter

  cout << "Number of coefficients: " <<
  BufferCod.GetNumCoef() << endl;

  // Asking the user to enter the size for the fixed codebook
  int size_FC;
  cout << "Specify the size for the fixed codebook ( max: 380): ";
  cin >> size_FC;
  cout << endl;

  // Asking the user to enter the size for the adaptive codebook
  int size_AC;
  cout << "Specify the size for the adaptive codebook: ";
  cin >> size_AC;
  cout << endl;

  // Preparing to set up the size of the adaptive codebook
  int samples_AC = size_AC + (int)(window*fs/4) - 1;

  // Setting up the voice block size
  BufferCod.SetBlockSize((int)(window * fs));
  BufferDecod.SetBlockSize((int)(window * fs));
  cout << "Block size: " << BufferCod.GetBlockSize() << endl;
  // Setting up the window size
  // Notice that the window size is the same as for the voice block
  BufferCod.SetWindowSize((int)(window * fs));
  BufferDecod.SetWindowSize((int)(window * fs));
  cout << "Window size: " << BufferCod.GetWindowSize() << endl;

  // Setting up the sizes for both codebooks
  BufferCod.SetFCSize(size_FC);
  BufferCod.SetACSize(size_AC);
  BufferCod.SetNumberOfSamples_AC(samples_AC);
  BufferDecod.SetFCSize(size_FC);
  BufferDecod.SetACSize(size_AC);
  BufferDecod.SetNumberOfSamples_AC(samples_AC);

  // Showing the selected sizes for the user
  cout << "Fixed codebook size: " << BufferCod.GetFCSize() << endl;
  cout << "Adaptive codebook size: " << BufferCod.GetACSize() << endl;
  cout << "Number of samples for the adaptive codebook: " <<
BufferCod.GetNumberOfSamples_AC() << endl;

  // Asking the user to enter the time during which the codec must be running
  int period;
  cout << "For how long the codec must be running? ";
  cin >> period;
  cout << endl;

  // Declaring some local variables
  int counter = 0; // Variable to count throughout the blocks
```

```
getchar();
BufferCod.Zero_SynthFilter_States();
BufferDecod.Zero_SynthFilter_States();

// The segment counter must be incremented
// in order to include the first block
MeasureSNRseg.IncrementNumberOfSegments();
MeasureSNRperc.IncrementNumberOfSegments();

// If the user specifies the period different than zero...
if (period!=0)
  // Setting up the end of the period
  while(counter < period*50)
    {
     // Increment block counter
     counter++;
     // Show the elapsed time every second
     if ( (counter%50 == 0) || (counter == 1) )
       cout << "Elapsed time: " << counter/50 << " sec" << endl;

     // Record some voice signal from the microphone
     if (BufferCod.Record(AudioDev, window))
      {
        cout << "Error on recording sound." << endl;
        exit(1);
      }

     // Apply a hamming window to the voice signal
     // in order to execute tha LPC Analysis
     BufferCod.ApplyWindow();
     BufferCod.AnalysisLPC();

     // If it is not the first block,
     // save the LSF coefficients
     // for interpolation
     if (counter != 1)
       BufferCod.SaveCoefLSF_before();

     // Convert the LPC coefficients to LSF coefficients
     BufferCod.CoefLPC_to_CoefLSF();

     // If it is not the first block,
     // save the LSF coefficients
     // for interpolation
     if (counter != 1)
       BufferDecod.SaveCoefLSF_before();

     // SENDING THE LSF COEFFICIENTS
     BufferDecod.ReceiveCoefInfo(BufferCod);

     // If it is not the first block...
     if (counter != 1)
       // This FOR loop means to indicate
       // which sub-block I am in
       for(int k=1; k<=4; k++)
         {
           BufferCod.SetVoiceSubBlock(k);

           BufferCod.InterpolateCoefLSF(k);
           BufferCod.CoefLSF_Interpolated_to_CoefLPC();

           BufferCod.SetGamma(gamma);
           BufferCod.SetPercFilter();

           BufferCod.ApplyPercFilter_to_VoiceSubBlock();

           BufferCod.SetTargetSignal();

           BufferCod.ApplySynthFilter_to_AC();
           BufferCod.Search_AC(k);
```

```
BufferCod.UpdateTargetSignal(k);

BufferCod.ApplySynthFilter_to_FC();
BufferCod.Search_FC(k);

// SENDING THE CODEBOOK INFO
BufferDecod.ReceiveCodebookInfo(BufferCod, k);

// DECODING...
BufferDecod.InterpolateCoefLSF(k);
BufferDecod.CoefLSF_Interpolated_to_CoefLPC();
BufferDecod.SetDecodedVoiceSubBlock(k);
BufferDecod.SetGamma(1);
BufferDecod.SetPercFilter();
BufferDecod.SetCompleteResponse(k);
BufferDecod.UpdateAdaptiveCodebook();
// END OF DECODING PROCESS.

// BufferCod.SetDecodedVoiceSubBlock(k);
BufferCod.SetCompleteResponse(k);
BufferCod.UpdateAdaptiveCodebook();

BufferPlay.ReceiveDecodedVoiceSubBlock(BufferDecod);
BufferPlay.Play(AudioDev);

for(int r=0; r<BufferCod.GetBlockSize()/4; r++)
 MeasureSNR.SetSignal(r, BufferCod.GetData((k-1)*BufferCod.GetBlockSize()/4
 + r) );
for(int s=0; s<BufferCod.GetBlockSize()/4; s++)
 MeasureSNR.SetEstimatedSignal(s, BufferPlay.GetData((k-
 1)*BufferPlay.GetBlockSize()/4 + s) );

for(int r=0; r<BufferCod.GetBlockSize()/4; r++)
 MeasureSNRseg.SetSignal(r, BufferCod.GetData((k-
 1)*BufferCod.GetBlockSize()/4 + r) );
for(int s=0; s<BufferCod.GetBlockSize()/4; s++)
 MeasureSNRseg.SetEstimatedSignal(s, BufferPlay.GetData((k-
 1)*BufferPlay.GetBlockSize()/4 + s) );

for(int r=0; r<BufferCod.GetBlockSize()/4; r++)
 MeasureSNRperc.SetSignal(r, BufferCod.GetData((k-
 1)*BufferCod.GetBlockSize()/4 + r) );
for(int s=0; s<BufferCod.GetBlockSize()/4; s++)
 MeasureSNRperc.SetEstimatedSignal(s, BufferPlay.GetData((k-
 1)*BufferPlay.GetBlockSize()/4 + s) );

for(int r=0; r<BufferCod.GetNumCoef(); r++)
 MeasureSNRperc.SetNumPercFilter(r, BufferCod.GetPercFilter(0, r));
for(int r=0; r<BufferCod.GetNumCoef(); r++)
 MeasureSNRperc.SetDenPercFilter(r, BufferCod.GetPercFilter(1, r));

MeasureSNR.AccumulateSignalPower();
MeasureSNR.AccumulateErrorPower();

MeasureSNRseg.AccumulateSignalPower();
MeasureSNRseg.AccumulateErrorPower();
if (k==4)
  {
    MeasureSNRseg.AccumulateSNR_seg(MeasureSNRseg.GetSNR());
    MeasureSNRseg.ZeroSignalPower();
    MeasureSNRseg.ZeroErrorPower();
    MeasureSNRseg.IncrementNumberOfSegments();
  }

MeasureSNRperc.ZeroStatesPercFilter();
MeasureSNRperc.AccumulateSignalPower();
MeasureSNRperc.AccumulatePercErrorPower();
if (k==4)
  {
    MeasureSNRperc.AccumulateSNR_seg(MeasureSNRperc.GetSNR_perc());
    MeasureSNRperc.ZeroSignalPower();
```

```
                MeasureSNRperc.ZeroPercErrorPower();
                MeasureSNRperc.IncrementNumberOfSegments();
              }
            }
      }
  else
    while(1)
      {
       counter++;
       if ( (counter%50 == 0) || (counter == 1) )
         cout << "Elapsed time: " << counter/50 << " sec" << endl;

       if (BufferCod.Record(AudioDev, window))
         {
           cout << "Error on recording sound." << endl;
           exit(1);
         }

       BufferCod.ApplyWindow();
       BufferCod.AnalysisLPC();

       if (counter != 1)
         BufferCod.SaveCoefLSF_before();
       BufferCod.CoefLPC_to_CoefLSF();

       if (counter != 1)
         for(int k=1; k<=4; k++)
           {
             BufferCod.SetVoiceSubBlock(k);

             BufferCod.InterpolateCoefLSF(k);
             BufferCod.CoefLSF_Interpolated_to_CoefLPC();

             BufferCod.SetGamma(gamma);
             BufferCod.SetPercFilter();

             BufferCod.ApplyPercFilter_to_VoiceSubBlock();

             BufferCod.SetTargetSignal();

             BufferCod.ApplySynthFilter_to_AC();
             BufferCod.Search_AC(k);

             BufferCod.UpdateTargetSignal(k);

             BufferCod.ApplySynthFilter_to_FC();
             BufferCod.Search_FC(k);

             BufferCod.SetCompleteResponse(k);
             BufferCod.SetDecodedVoiceSubBlock(k);
             BufferCod.UpdateAdaptiveCodebook();

             BufferPlay.ReceiveDecodedVoiceSubBlock(BufferCod);
             BufferPlay.Play(AudioDev);
           }
      }

  cout << "Number of segments: " << MeasureSNRperc.GetNumberOfSegments() << endl;
  cout << "Classic Signal-to-Noise Ratio (SNR): " << MeasureSNR.GetSNR() << " dB" <<
endl;
  cout << "Segmented Signal-to-Noise Ratio (SNR_seg): " << MeasureSNRseg.GetSNR_seg() <<
" dB" << endl;
  cout << "Perceptual segmented Signal-to-Noise Ratio (SNR_fw_seg): " <<
MeasureSNRperc.GetSNR_seg_perc() << " dB" << endl;
  AudioDev.Close();
}
```

C.8 QUANTIZA8.H

```
/*-------------------------------------------------------------------*/
/* QUANTIZA8.H                                                       */
/*                                                                   */
/* Arquivo de cabecalho contendo as definicoes das particoes e       */
/* dicionarios para a quantizacao escalar diferencial das frequencias */
/* do espectro de linha (LSF) e quantizacao escalar dos ganhos dos   */
/* dicionarios adaptativo e fixo.                                    */
/*-------------------------------------------------------------------*/

/* Particoes e dicionarios para os ganhos */

float cb_alfa[64]={0.000893795551638, 0.041835916787676, 0.085556370351797,
0.113101297842831, 0.141713181265795, 0.174520403445877, 0.206681391189327,
0.235161933546835, 0.264554441670896, 0.297038880700646, 0.325819592941087,
0.358098967167711, 0.391821050581906, 0.420085595497871, 0.452872440303770,
0.484741570465221, 0.515473604096934, 0.550171356536141, 0.586268231042189,
0.619252280430499, 0.651214179304077, 0.680335779135679, 0.709467325540363,
0.738150707348702, 0.764243758733728, 0.794464073706121, 0.826486106310860,
0.858000144995226, 0.887354105535661, 0.917965465034118, 0.949316571229578,
0.977986611098336, 1.006339198931354, 1.037228791111056, 1.068131557976650,
1.097599497379135, 1.129918263816476, 1.161748760102151, 1.198080820504875,
1.231254962017847, 1.260572116123707, 1.294027654238037, 1.322287669009015,
1.355292143130367, 1.388800519377934, 1.424571898272679, 1.450596254343975,
1.480077613922036, 1.505850542850074, 1.540968329497666, 1.571356075145555,
1.606679168775145, 1.635234975122213, 1.660523826445399, 1.695179561043177,
1.733562797601428, 1.758749390876438, 1.782961180311480, 1.808166411104511,
1.847356958625124, 1.869347980804206, 1.914083460193905, 1.959866241225382,
1.986634928289086};

float part_alfa[63]={0.021364856169657, 0.063696143569737, 0.099328834097314,
0.127407239554313, 0.158116792355836, 0.190600897317602, 0.220921662368081,
0.249858187608865, 0.280796661185771, 0.311429236820866, 0.341959280054399,
0.374960008874808, 0.405953323039889, 0.436479017900821, 0.468807005384495,
0.500107587281077, 0.532822480316538, 0.568219793789165, 0.602760255736344,
0.635233229867288, 0.665774979219878, 0.694901552338021, 0.723809016444532,
0.751197233041215, 0.779353916219925, 0.810475090008490, 0.842243125653043,
0.872677125265443, 0.902659785284890, 0.933641018131848, 0.963651591163957,
0.992162905014845, 1.021783995021205, 1.052680174543853, 1.082865527677892,
1.113758880597806, 1.145833511959314, 1.179914790303513, 1.214667891261361,
1.245913539070777, 1.277299885180872, 1.308157661623526, 1.338789906069691,
1.372046331254150, 1.406686208825307, 1.437584076308327, 1.465336934133005,
1.492964078386055, 1.523409436173870, 1.556162202321611, 1.589017621960350,
1.620957071948679, 1.647879400783806, 1.677851693744288, 1.714371179322303,
1.746156094238933, 1.770855285593959, 1.795563795707996, 1.827761684864818,
1.858352469714665, 1.891715720499055, 1.936974850709644, 1.973250584757234};

float cb_beta[64]={-0.049424408391575, -0.047385107996578, -0.045728002583410, -
0.044270491278391, -0.043264944495707, -0.042067744515553, -0.040284964915407, -
0.038049993134037, -0.036282690278334, -0.034968977485494, -0.033641929603871, -
0.032159945892842, -0.030416040320499, -0.028837413761267, -0.027461077942302, -
0.026004390506475, -0.024511759675677, -0.022757997674721, -0.020880833985011, -
0.019347862423193, -0.017758287576719, -0.016357825502620, -0.014940785695009, -
0.013364598380344, -0.011489292365985, -0.009775847521798, -0.008170442074042, -
0.006429429754271, -0.004705740040280, -0.003158815029506, -0.001731714444675, -
0.000534533455254, 0.000562666243796, 0.001831968397185, 0.003265211033122,
0.004827368274163, 0.006423793113895, 0.008096999760567, 0.009758463336130,
0.011380832128119, 0.012837534236255, 0.014286685476486, 0.015942350569851,
0.017605872917246, 0.019194739177374, 0.020691003789584, 0.022277544031749,
0.023836182859221, 0.025444975995800, 0.026936181764071, 0.028500293409446,
0.030073925068073, 0.031827269079533, 0.033054666608963, 0.034470527283532,
0.035992419510285, 0.037370289255256, 0.039335892412314, 0.041029796678521,
0.043264798459508, 0.044740254671797, 0.046137658872339, 0.046959254512298,
0.048909653614935};

float part_beta[63]={-0.048404758194077, -0.046556555289994, -0.044999246930900, -
0.043767717887049, -0.042666344505630, -0.041176354715480, -0.039167479024722, -
```

```
0.037166341706185, -0.035625833881914, -0.034305453544682, -0.032900937748357, -
0.031287993106671, -0.029626727040883, -0.028149245851784, -0.026732734224388, -
0.025258075091076, -0.023634878675199, -0.021819415829866, -0.020114348204102, -
0.018553074999956, -0.017058056539669, -0.015649305598815, -0.014152692037677, -
0.012426945373164, -0.010632569943891, -0.008973144797920, -0.007299935914157, -
0.005567584897276, -0.003932277534893, -0.002445264737091, -0.001133123949965,
0.000014066394271, 0.001197317320490, 0.002548589715154, 0.004046289653642,
0.005625580694029, 0.007260396437231, 0.008927731548349, 0.010569647732125,
0.012109183182187, 0.013562109856371, 0.015114518023168, 0.016774111743548,
0.018400306047310, 0.019942871483479, 0.021484273910667, 0.023056863445485,
0.024640579427511, 0.026190578879935, 0.027718237586759, 0.029287109238760,
0.030950597073803, 0.032440967844248, 0.033762596946248, 0.035231473396909,
0.036681354382771, 0.038353090833785, 0.040182844545417, 0.042147297569015,
0.044002526565653, 0.045438956772068, 0.046548456692319, 0.047934454063617};

/*float part_alfa[15]={0.178798537962681, 0.336863128089299, 0.464555026256744,
0.577892379968758, 0.682328080383915, 0.779183776972088, 0.867326787148985,
0.950380326907653, 1.037096539677762, 1.134277121708598, 1.256393043725464,
1.404376356155321, 1.558305047491990, 1.702109482664061, 1.842729162499622};

float cb_alfa[16]={0.090152491599960, 0.267444584325401, 0.406281671853196,
0.522828380660292, 0.632956379277224, 0.731699781490605, 0.826667772453571,
0.907985801844400, 0.992774851970905, 1.081418227384618, 1.187136016032578,
1.325650071418350, 1.483102640892292, 1.633507454091687, 1.770711511236435,
1.914746813762810};

float part_beta[31]={-0.047470013584357, -0.044512169652036, -0.041097218572299, -
0.036809387583606, -0.033012569682746, -0.029540264712877, -0.025604981766582, -
0.021829080221179, -0.018402561585128, -0.015341240429064, -0.012403125020266, -
0.009509918797114, -0.006715479636319, -0.004123914717704, -0.001829902720310,
0.000009466284907, 0.001811052792571, 0.004063556586115, 0.006624352359525,
0.009383622864734, 0.012284475465621, 0.015411002495124, 0.018715220119041,
0.022108571456799, 0.025579118141511, 0.028942293893770, 0.032230186928221,
0.035670215625160, 0.039560464694379, 0.043250334157493, 0.046378722921206};

float cb_beta[32]={-0.048985146856528, -0.045954880312187, -0.043069458991885, -
0.039124978152713, -0.034493797014499, -0.031531342350994, -0.027549187074760, -
0.023660776458405, -0.019997383983953, -0.016807739186302, -0.013874741671825, -
0.010931508368707, -0.008088329225521, -0.005342630047116, -0.002905199388291, -
0.000754606052328, 0.000773538622143, 0.002848566962998, 0.005278546209231,
0.007970158509818, 0.010797087219651, 0.013771863711591, 0.017050141278657,
0.020380298959425, 0.023836843954173, 0.027321392328849, 0.030563195458692,
0.033897178397750, 0.037443252852569, 0.041677676536190, 0.044822991778796,
0.047934454063617};*/

/* Particoes e diconarios para as diferencas das LSF */

float cb0[16]={0.072407903144394, 0.100658431630177, 0.124008872747483,
0.143865241944157, 0.159991224708484, 0.180017159382504, 0.200557568659917,
0.220890881266554, 0.242469278260079, 0.265425748682481, 0.293244572076513,
0.315352329509528, 0.343011381881636, 0.367452515439591, 0.392857895451690,
0.444125673092013};

float cb1[16]={0.037382194722816, 0.066164735708266, 0.092335519009562,
0.121316187946308, 0.150649675200874, 0.180340667909010, 0.211169346534004,
0.238054081527895, 0.266276687785733, 0.296242494222506, 0.325978363640153,
0.358828390899289, 0.390147993164778, 0.432918705152659, 0.471065340921044,
0.644300518663287};

float cb2[16]={0.074253521959799, 0.103470257322683, 0.134134907514706,
0.167634281560915, 0.197657974587770, 0.229852062425989, 0.262499000331462,
0.295127849321933, 0.330038575943443, 0.367538039540775, 0.400652133370041,
0.434571735711270, 0.466756976316794, 0.512237943411042, 0.574998973300961,
0.658654821370170};

float cb3[16]={0.088762915886600, 0.141420305901675, 0.183629102521759,
0.225569371140999, 0.270164434326305, 0.314928620223238, 0.356206960310794,
0.396462046678053, 0.442078123594392, 0.485349085182054, 0.531019917078453,
0.578928013599770, 0.630625107487479, 0.686875136318929, 0.738375000979514,
0.815300138360185};
```

```
float cb4[16]={0.080275943424657, 0.134823513665360, 0.189776480237868,
0.238300353028462, 0.286682950544450, 0.335714003264665, 0.387171633807212,
0.441218217435768, 0.494103114733221, 0.552096841145461, 0.602274636170048,
0.646165212967930, 0.693399491254751, 0.753339456438345, 0.839844843308202,
0.952754539571523};

float cb5[16]={0.074548130410185, 0.119177943162966, 0.161492252111636,
0.208059526095671, 0.254018890415106, 0.296948063488892, 0.339548313200504,
0.381772579000959, 0.425726592798761, 0.468504927993540, 0.513264218786128,
0.560485164042384, 0.642031179459868, 0.725359596114697, 0.845850258592563,
0.994529412051966};

float cb6[16]={0.093360088382960, 0.146007470907300, 0.191399272419039,
0.234569255761764, 0.274375969930060, 0.314761658766922, 0.354450717128406,
0.390862557890504, 0.435624202288044, 0.478762925721652, 0.523558817241844,
0.566976718767956, 0.620209865829910, 0.687420563639455, 0.784986411547219,
0.959666506077641};

float cb7[16]={0.066295785308115, 0.107845496517064, 0.143733199004129,
0.178774865190489, 0.207536485157857, 0.237292698856789, 0.266557740482310,
0.296301923342842, 0.326238204505458, 0.354925116201829, 0.384119765429104,
0.416980872188534, 0.458736061968133, 0.510335286320022, 0.597108253601529,
0.717072410357009};

float cb8[16]={0.097753559205830, 0.152539480023109, 0.194511411093453,
0.227332243857691, 0.255481079817469, 0.282752156173793, 0.313260905558370,
0.345480851668608, 0.377478116215232, 0.410612073326079, 0.445094259146135,
0.484274426457249, 0.529378250618326, 0.573041677868958, 0.633721116466827,
0.736139125171183};

float cb9[16]={0.089817441150800, 0.126579870388171, 0.161724764796744,
0.193334653580903, 0.221303923890452, 0.246980637382744, 0.273365333267359,
0.299980741646941, 0.329918854578125, 0.359999528708937, 0.389640774014590,
0.426774999670037, 0.463430886322379, 0.504741865972167, 0.544370597084599,
0.585957370956267};

float part0[15]={0.086533167387285, 0.112333652188830, 0.133937057345820,
0.151928233326320, 0.170004192045494, 0.190287364021211, 0.210724224963236,
0.231680079763317, 0.253947513471280, 0.279335160379497, 0.304298450793020,
0.329181855695582, 0.355231948660613, 0.380155205445640, 0.418491784271852};

float part1[15]={0.051773465215541, 0.079250127358914, 0.106825853477935,
0.135982931573591, 0.165495171554942, 0.195755007221507, 0.224611714030950,
0.252165384656814, 0.281259591004120, 0.311110428931330, 0.342403377269721,
0.374488192032033, 0.411533349158719, 0.451992023036852, 0.557682929792165};

float part2[15]={0.088861889641241, 0.118802582418695, 0.150884594537811,
0.182646128074342, 0.213755018506879, 0.246175531378725, 0.278813424826697,
0.312583212632688, 0.348788307742109, 0.384095086455408, 0.417611934540656,
0.450664356014032, 0.489497459863918, 0.543618458356001, 0.616826897335565};

float part3[15]={0.115091610894137, 0.162524704211717, 0.204599236831379,
0.247866902733652, 0.292546527274771, 0.335567790267016, 0.376334503494423,
0.419270085136222, 0.463713604388223, 0.508184501130253, 0.554973965339111,
0.604776560543625, 0.658750121903204, 0.712625068649221, 0.776837569669850};

float part4[15]={0.107549728545008, 0.162299996951614, 0.214038416633165,
0.262491651786456, 0.311198476904557, 0.361442818535938, 0.414194925621490,
0.467660666084494, 0.523099977939341, 0.577185738657754, 0.624219924568989,
0.669782352111340, 0.723369473846548, 0.796592149873274, 0.896299691439863};

float part5[15]={0.096863036786575, 0.140335097637301, 0.184775889103653,
0.231039208255388, 0.275483476951999, 0.318248188344698, 0.360660446100731,
0.403749585899860, 0.447115760396150, 0.490884573389834, 0.536874691414256,
0.601258171751126, 0.683695387787283, 0.785604927353630, 0.920189835322265};

float part6[15]={0.119683779645130, 0.168703371663170, 0.212984264090402,
0.254472612845912, 0.294568814348491, 0.334606187947664, 0.372656637509455,
0.413243380089274, 0.457193564004848, 0.501160871481748, 0.545267768004900,
0.593593292298933, 0.653815214734683, 0.736203487593337, 0.872326458812430};
```

```
float part7[15]={0.087070640912589, 0.125789347760597, 0.161254032097309,
0.193155675174173, 0.222414592007323, 0.251925219669550, 0.281429831912576,
0.311270063924150, 0.340581660353644, 0.369522440815467, 0.400550318808819,
0.437858467078333, 0.484535674144077, 0.553721769960776, 0.657090331979269};

float part8[15]={0.125146519614469, 0.173525445558281, 0.210921827475572,
0.241406661837580, 0.269116617995631, 0.298006530866082, 0.329370878613489,
0.361479483941920, 0.394045094770656, 0.427853166236107, 0.464684342801692,
0.506826338537787, 0.551209964243642, 0.603381397167892, 0.684930120819005};

float part9[15]={0.108198655769486, 0.144152317592457, 0.177529709188823,
0.207319288735677, 0.234142280636598, 0.260172985325052, 0.286673037457150,
0.314949798112533, 0.344959191643531, 0.374820151361764, 0.408207886842313,
0.445102942996208, 0.484086376147273, 0.524556231528383, 0.565163984020433};
```