UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ELETRÔNICA

# CODIFICAÇÃO DE VOZ UTILIZANDO RECORRÊNCIA DE PADRÕES MULTIESCALA

Autor: Gabriel Candido Ribeiro Abrahão

Orientador: Sérgio Lima Netto

Co-Orientador: Eduardo Antônio Barros da Silva

Examinador: Gelson Vieira Mendonça

Examinador: Lara Feio

DEL
Novembro de 2005

*Dedico este projeto final a meus pais,*
*cuja perseverança tem servido*
*de exemplo para meus passos*
*e guiado minhas escolhas.*

# Agradecimentos

Meus sinceros agradecimentos são para todos os que me incentivaram e orientaram durante o período de elaboração deste trabalho. Em especial aos professores que direcionaram o desenvolvimento acadêmico do projeto, aos meus pais e a minha namorada Naraiana que sempre me motivou, o meu muito obrigado, pois sem o seu apoio não teria completado este importante passo da minha carreira acadêmica.

# Resumo

Este trabalho propõe a implementação de um algoritmo de compressão baseado na aproximação de sinais através do casamento de padrões particionados em escalas recorrentes. Este algoritmo, chamado de Multidimensional Multiscale Parser, foi desenvolvido inicialmente no campo da compressão de imagens e este trabalho visa estudar a viabilidade de sua aplicação à codificação de voz.

O algoritmo funciona basicamente particionando o sinal original em blocos e buscando aproximações em um dicionário, porém este bloco inicial pode ser subdividido, formando assim uma árvore binária. Desta maneira podemos obter aproximações mais eficientes destes subblocos que minimizem a distorção causada ao sinal. Além disso, a concatenação destas aproximações realimenta os dicionários, que de maneira adaptativa, passam a representar de forma mais acurada o sinal de entrada utilizando um número menor de subdivisões.

A implementação deste codificador realizada neste trabalho gerou resultados que mostram a viabilidade da aplicação deste método a compressão de voz quando comparados frente a outras técnicas.

# Abstract

This work proposes the implementation of a compression algorithm based on the approximation of the input signal through pattern matching splitted in recurring scales. This algorithm, called Multidimensional Multiscale Parser, was developed initially in the image compression field and this work intends to evaluate its application in the voice coding field.

This algorithm works by splitting the original signal in blocks and searching for the best match in a dictionary, but this initial block can also be subdivided, thus creating a binary tree. In that way we can obtain more efficient representations of those sub blocks that minimize the distortion caused in the signal. Beyond that, the concatenation of those vectors is used to feedback the dictionary in an adaptive fashion, improving its capability of representing accurately the input signal using less subdivisions.

The implementation of the proposed algorithm in this work achieved results that show that this method can also be successfully applied to the voice encoding problem.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# 1 Introdução

A codificação de voz lida com o problema de buscar uma representação eficiente dos sinais de voz no domínio digital. Um dos objetivos primordiais deste processo é comprimir o sinal original, ou seja conseguir representá-lo utilizando o menor número de bits possível e assim otimizar seu armazenamento ou transmissão. Lembrando sempre que o objetivo final é reconverter esse sinal ao domínio analógico e que, para isso, é importante controlar o nível de distorção introduzida neste processo para que se mantenha a inteligibilidade da fala.

Os codificadores de voz são em muitos aspectos semelhantes aos algoritmos utilizados para a compressão de áudio. Ambos se valem de modelos psicoacústicos para avaliar nossa compreensão do mundo que nos cerca e entender as imperfeições do nosso aparelho auditivo. De posse dessas informações, esses algoritmos buscam introduzir perdas no sinal de tal modo que estas passem despercebidas para a maioria dos ouvintes porém, reduzindo a taxa de codificação.

A principal vantagem no projeto dos codificadores de voz, é que a fala humana é um caso particular do áudio genérico e assim apresenta uma complexidade significantemente menor. Podemos dessa forma obter modelos estatísticos mais precisos que mantenham uma fidelidade aceitável e contenham menos informação. Outro ponto importante, é que na transmissão da fala humana, muitas vezes se aceita uma relação de compromisso. Nela se abre mão da qualidade do sinal em troca da agilidade ou maior capacidade de transmissão, desde que a mensagem original seja capaz de ser recuperada. Sempre enfatizando que a mensagem original, além de seu conteúdo literal, contém outras informações implícitas na voz. Entre elas, a identidade do locutor, emoções, entonação e timbre. Todos fatores relevantes no processo de recuperação da informação enviada pelo transmissor.

## 1.1 Histórico

Partindo desses preceitos, a codificação de voz digital vem sendo estudada desde a década de 1940 e teve seu início com o PCM (*Pulse Code Modulation*) ou modulação por pulsos, que consiste simplesmente em representar digitalmente um sinal de voz amostrado em intervalos regulares. A grande motivação da época era o surgimento de aplicações militares para a transmissão encriptada de mensagens de voz. Assim a primeira aplicação da invenção do engenheiro inglês Alec Reeves, foi um equipamento chamado SIGSALY utilizado para transmissão de ligações telefônicas criptografadas durante a Segunda Guerra Mundial (BENNETT, 1983).

Após a guerra, os algoritmos evoluíram rapidamente e diversos avanços incrementais foram adicionados à tecnologia dos chamados codificadores de forma de onda, principalmente motivados pelo crescente uso da telefonia. Surgiram as técnicas de mapeamento logarítmico, padronizadas com os nomes de lei A e lei $\mu$, onde se começou a explorar as imperfeições da nossa audição para se reduzir a quantidade de informação a ser transmitida. Vieram também as técnicas que buscavam explorar a correlação estatística entre as amostras de voz, entre elas o DPCM (modulação por pulsos diferencial) e o ADPCM (modulação por pulsos diferencial adaptativa). Cujo objetivo era transmitir apenas uma informação incremental ou seja, a diferença em relação a amostra anterior.

Com o aumento da capacidade computacional e a crescente necessidade de se possibilitar mais conversas simultâneas, pelos mesmos canais cuja capacidade já estava saturada, houve a primeira quebra de paradigma na codificação de voz. Ao invés de se analisar cada amostra individualmente, a chamada quantização e codificação escalar, as novas técnicas analisavam blocos de amostras. O passo inicial dessa idéia foi a quantização vetorial, onde se busca representar por um índice de um dicionário um vetor contendo diversas amostras do sinal original. Essa técnica possibilita uma taxa de compressão muito grande, porém com um custo elevado na complexidade computacional.

No entanto a necessidade da obtenção de taxas ainda mais baixas continuava e motivou o surgimento paralelo de uma nova família de codificadores conhecidos como paramétricos ou vocoders. A diferença chave na abordagem desses algoritmos é que eles buscam modelar a voz humana através de um conjunto reduzido de parâmetros. Esses modelos tentam, através de uma aproximação matemática do funcionamento dos órgãos humanos responsáveis pela fala, reproduzir a estatística encontrada nos sinais de voz e, de posse desses parâmetros, re-sintetizar a voz original no receptor.

Nos vocoders o trato vocal é representado como um filtro digital variante no tempo e que é excitado pelo ar expelido dos pulmões, representado por ruído branco no caso de segmentos não-vozeados ou por um trem de pulsos equivalente a vibração das cordas vocais no caso de segmentos vozeados. Nas implementações de vocoders aplicados à compressão de voz, iniciadas por volta de 1970, utiliza-se a predição linear para se obter a conformação do espectro do sinal. E assim surgem a família de codificadores LPC (*Linear Prediction Coder*) onde se obtém um sinal inteligível a baixíssimas taxas porém longe de conter todas as características da voz original. Dessa forma a aplicação dos vocoders se manteve principalmente no campo militar, onde a mensagem explícita era o fundamental e as baixas taxas permitiam que fossem aplicados pesados algoritmos de criptografia sobre o fluxo de dados gerado.

Porém a partir da década de 80, na tentativa de se ocupar o espaço deixado entre os codificadores de forma de onda e os paramétricos, surge uma nova abordagem que dá origem aos codificadores híbridos. De forma geral, os codificadores de forma de onda são capazes de gerar voz com uma boa qualidade utilizando taxas de até 16 kilobits por segundo, porém seu uso é limitado para aplicações que exigem taxas inferiores. Por outro lado os vocoders podem reconstruir mensagens inteligíveis a taxas de 2,4 kilobits por segundo e até mesmo inferiores, porém sem prover a fala de naturalidade a nenhuma das taxas utilizadas.

Os codificadores híbridos, mais utilizados e bem-sucedidos atualmente, se baseiam técnicas no domínio do tempo conhecidas como análise por síntese. Eles utilizam o mesmo modelo de filtro de predição linear encontrado nos vocoders LPC. No entanto, ao invés de aplicar um modelo de dois estados, extremamente simplificado de vozeado ou não vozeado para gerar a excitação, a entrada para este filtro é escolhida em um ou mais dicionários em uma tentativa de aproximar o sinal reconstruído da forma de onda do sinal original.

Os codificadores utilizando análise por síntese foram introduzidos em 1982 por Atal e Remde com a técnica denominada MPE(*Multi-Pulse Excited*) ou codificador excitado por multiplos pulsos (ATAL; REMDE, 1982). Esse codificador determinava a melhor excitação para cada quadro de voz como um trem de pulsos de amplitude arbitrária escolhido, porém com espaçamento fixo entre eles. Assim adicionava aos vocoders mais liberdade na escolha da excitação, pois além de definir a freqüência fundamental, podia também variar sua amplitude ao longo do tempo. Esses codificadores foram logo suplantados pelo algoritmo proposto por Schroeder e Atal em 1985, cuja principal diferença era a introdução de um dicionário onde se buscaria o sinal de excitação. Em outras palavras, a escolha da entrada

do filtro de predição seria quantizada vetorialmente. Esta é a abordagem mais comum na codificação de voz a baixas taxas atualmente e originou a família de codificadores CELP(*Code Excited Linear Prediction*), que atingem taxas entre 4,8 e 16 kilobits por segundo com qualidade compatível com os codificadores da família PCM.

No entanto, em seu início, o trabalho dos codificadores CELP de analisar através de tentativas qual a melhor excitação em um dicionário era impráticavel para os computadores existentes. Uma CPU Cray-1 da época, levava aproximadamente 125 segundos para processar 1 segundo de sinal de voz. Desde então muito trabalho de otimização foi realizado e isso, aliado ao aumento no poder de tratar números dos novos computadores, tornou possível a ampla implementação dessa família de codificadores em dispositivos de baixo custo como os celulares atuais.

O desenvolvimento de novas estratégias de codificação de voz cada vez mais eficientes continua hoje em dia sendo motivado pelos mesmos fatores de mercado que impulsionaram seu desenvolvimento nas últimas seis décadas. O aproveitamento eficiente dos canais de transmissão disponíveis, aliados à vontade de se comunicar a distância de forma segura e confiável. Porém, essa busca foi muito fortalecida pela explosão na telefonia celular e mais recentemente pelo surgimento da telefonia sobre IP. As pesquisas atuais continuam buscando codificadores a baixas taxas, motivados pelas forças econômicas envolvidas, porém com um foco importante em aliar estes ganhos a melhorias na qualidade de voz percebida pelo usuário e a uma maior robustez a erros do canal. Abrindo espaço assim para o surgimento de novas estratégias de codificadores híbridos ou até mesmo de forma de onda que sejam eficientes e capazes de transmitir com uma maior riqueza as nuances de entonação e timbre da voz humana.

## 1.2 Algoritmo proposto

Assim, a proposta deste trabalho parte da busca de um novo conceito de codificação de voz, partindo do estudo da viabilidade em se aplicar um algoritmo desenvolvido originalmente no campo do processamento de imagens chamado de *Multiscale Multidimensional Parser* a este problema. De forma sucinta, a idéia por trás deste codificador consiste na aplicação de uma quantização vetorial multiescala, sem muitas suposições a priori sobre o modelo estatístico do sinal de entrada. O algoritmo tenta representar cada quadro de voz através de uma árvore binária composta de blocos de tamanho variável encontrados em um dicionário. Busca-se uma relação de compromisso entre o erro admitido no sinal e o

particionamento aceito na árvore, minizando assim a taxa de bits gerada pelo codificador para uma dada distorção.

O algoritmo se aprimora, aprendendo os padrões existentes no sinal, atualizando seus dicionários com as melhores representações encontradas até então. Dessa forma no futuro pode se representar quadros semelhantes utilizando uma árvore menor e necessitando assim de menos bits transmitidos. Valendo ressaltar que, como o dicionário é formado através da junção, expansão e contração de seus próprios vetores originais, não é necessário que ele seja transmitido explicitamente, podendo ser reconstruído durante o processo de decodificação. Como saída do codificador é gerado para cada quadro uma representação do particionamento desta árvore, bem como os índices do dicionário encontrados em cada nó.

Após a geração dos símbolos pelo codificador acima, a correlação entre eles ainda pode ser explorada através dos métodos convencionais de codificação por entropia, resultando assim em uma taxa menor de bits a ser transmitida sem impactar na qualidade do sinal resultante. Neste trabalho em particular foi proposto um codificador aritmético, utilizando diferentes modelos estatísticos para cada nível da árvore percorrida.

Desta forma, este trabalho consiste em implementar em linguagem C uma biblioteca que realize a codificação e a decodificação de um arquivo contendo um sinal de voz humana utilizando o algoritmo proposto acima e estudar suas possíveis adaptações e melhorias para explorar sua estatística em particular. Sua eficácia será avaliada utilizando métodos objetivos levando em conta parâmetros perceptuais. Em particular será utilizada a recomendação do ITU P.862 e buscando ainda facilitar sua comparação com outros trabalhos do genêro, será utilizado como conjunto de sinais para teste o banco de frases fornecido na implementação padrão deste avaliador.

## 1.3 Organização do projeto

Este trabalho está estruturado de maneira a possibilitar inicialmente o estudo comparativo das tecnologias de compressão e codificação existentes de uma maneira sucinta em seu Capítulo 2, e mais especificamente suas aplicações ao campo da voz. Incluindo também, uma visão geral do método de codificação aritmética utilizado neste trabalho.

Após esta análise inicial, o algoritmo do codificador por recorrência de padrões multiescala será explicado de maneira genérica no Capítulo 3. São descritos em detalhe os algoritmos de particionamento dos sinais de entrada, de atualização dos dicionários e

formação das árvores de codificação, bem como o caminho reverso utilizado na recuperação do sinal decodificado.

O Capítulo 4 deste trabalho trata das adaptações necessárias para o tratamento da voz e da implementação desenvolvida durante este projeto. No Capítulo 5 os resultados obtidos com o algoritmo proposto serão apresentados e discutidos, justificando as escolhas de projeto expostas anteriormente.

No Capítulo 6 temos as conclusões do trabalho, avaliando comparativamente esta técnica de codificação frente a outros métodos previamente estabelecidos e propondo futuras continuações à este trabalho.

# 2 Fundamentos de compressão de sinais

## 2.1 Introdução

A compressão de sinais ou codificação de fonte é o processo onde se codifica uma fonte de informação utilizando uma menor quantidade de símbolos do que uma representação direta utilizaria. Este objetivo é alcançado através de algoritmos específicos de codificação, que devem ser conhecidos tanto pela transmissor quanto pelo receptor para que a mensagem possa ser interpretada.

Este capítulo tem como finalidade fundamentar a base teórica envolvida neste processo. Na seção 2.2 há uma breve motivação para o estudo destes algoritmos, seguida na seção 2.3 pela apresentação das técnicas de compressão sem perdas, dando ênfase a codificação aritmética. Na seção 2.4, os algoritmos de compressão com perdas são abordados, especificamente a teoria envolvida nos processos que analisam a relação taxa-distorção. Por último, na seção 2.5, as medidas de desempenho de sistemas de compressão serão tratadas, entre elas as medidas pseudo-subjetivas de qualidade de voz utilizadas posteriormente durante a análise dos resultados.

## 2.2 Motivação

A importância da compressão reside no fato de que cada vez mais informações são geradas e utilizadas em formato digital, necessitando de espaço para armazenamento e de banda para sua transmissão. Estes são recursos finitos que, apesar de sua enorme evolução, não conseguem acompanhar o crescimento da demanda, se tornando assim economicamente dispendiosos. A compressão aparece como uma forma eficiente e econômica de se utilizar esses recursos, tomando proveito do fato de que a enorme maioria das fontes reais de dados são extremamente redundantes estatisticamente. Ou seja, quando apresentadas de uma forma interpretável para o ser humano, os sinais apresentam uma correlação estatística entre suas amostras que pode ser minimizada, gerando uma repre-

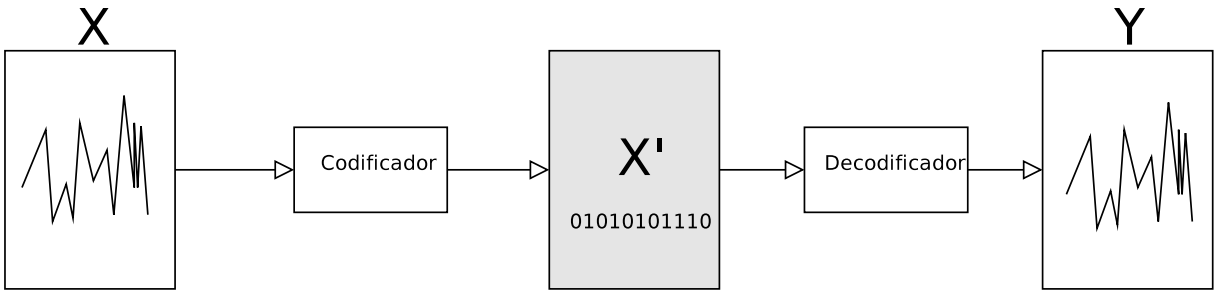


Figura 1: Esquema genérico de compressão.

sentação muito mais compacta da mesma informação. Essas formas compactas são criadas através da identificação e utilização de estruturas comuns ou repetitivas presentes nos dados originais, permitindo, por exemplo, sua reutilização, ou ainda predição, através de cálculos estatísticos.

Como exposto anteriormente, quando alguma técnica de compressão é analisada, trabalha-se com dois elementos diferentes: aquele que processa o dado de entrada X e gera uma representação X' que necessita de um número menor de bits para ser armazenada, chamado de codificador, e aquele que processa a informação comprimida X' e gera o dado reconstruído Y, chamado de decodificador como representado em 1.

Dependendo das necessidades de transmissão de informação e da taxa que se deseja alcançar, os algoritmos de compressão podem ser classificados em duas grandes classes: sem perdas e com perdas. Enquanto na primeira, o dado reconstruído Y é igual ao dado de entrada X, na segunda aceita-se uma diferença entre a reconstrução e a informação original. A compressão com perdas permite atingir taxas de dados muito inferiores aos outros métodos e é aplicada largamente a sinais de áudio e imagens, onde os sentidos humanos muitas vezes mascaram essas diferenças.

Assim a compressão é sempre uma relação de compromisso entre alguns fatores como os mencionados: taxa resultante e perdas inseridas. Outro ponto relevante é a quantidade de processamento necessário. Pois, se inicialmente ela é projetada para economizar em custos de banda e espaço de armazenamento, por outro lado ela requer poder de processamento, o que também se reverte em mais gastos. Dessa forma o projeto de esquemas de codificação deve considerar todos esses aspectos e relacioná-los com a necessidade do usuário, tentando equacionar a melhor solução possível.

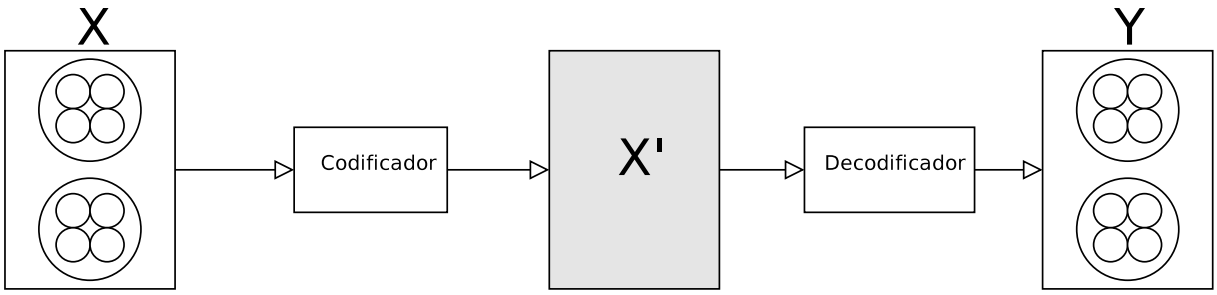


Figura 2: Compressão de sinais sem perdas.

## 2.3 Compressão de sinais sem perdas

Como dito anteriormente, a compressão de sinais sem perdas não provoca distorção na entrada X do codificador, ou seja, podemos recuperá-la completamente, sem perda alguma, quando efetuamos a reconstrução no decodificador como representado na figura 2. Uma característica importante é que não se consegue uma taxa de compressão alta quando se utiliza codificadores desta família, ou seja, a relação entre o tamanho de X e X' não possui um valor elevado.

Em muitos casos deve-se optar pelo uso de um algoritmo de compressão sem perdas devido a necessidades específicas do problemas enfrentado. O caso mais comum é a compressão de texto ou *software* em computadores. A alteração de apenas um bit que seja pode alterar completamente o significado do texto ou inutilizar a lógica implementada no *software*. Outro exemplo bastante comum é a aquisição de imagens em situações onde ela será intensamente manipulada e as perdas inseridas pela compressão em cada etapa de processamento não são aceitáveis. Nesses casos, para atender os requisitos e ainda economizar em espaço utilizado, as técnicas de compressão sem perdas são indicadas.

Existem técnicas bastante conhecidas para compressão sem perdas, dentre as quais podemos destacar a técnica desenvolvida por David Hufffman, chamado de código de Huffman, as técnicas baseadas em dicionários desenvolvidas por Abraham Lempel e Jacob Ziv, e o método de codificação aritmética criado por Peter Elias descrito em maior profundidade na sub-seção a seguir. Essas técnicas também são conhecidas como codificadores de entropia e buscam de uma maneira geral associar descrições curtas aos símbolos que aparecem freqüêntemente e descrições longas aos símbolos com baixa probabilidade de acontecer. Dessa maneira quanto mais se conhece da estatística de X, na média, mais X' se aproximará da entropia da fonte.

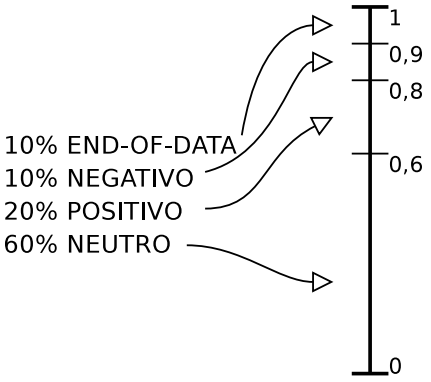


Figura 3: Atribuição de intervalos da faixa original proporcionais a probabilidade de ocorrência de cada símbolo.

### 2.3.1 Codificação aritmética

A codificação aritmética é um método de codificação por entropia que, diferente das outras técnicas que separam a mensagem de entrada em símbolos e os substituem por palavras, funciona codificando toda a mensagem em um único número n contido no intervalo de 0 a 1.

Codificadores aritméticos produzem como saída um resultado ótimo para um dado conjunto de símbolos e probabilidades. Algoritmos que utilizam esse tipo de codificação começam pela determinação de um modelo dos dados, basicamente uma predição dos padrões de símbolos que serão encontrados na mensagem. Quanto melhor essa predição, mais próximo do ótimo a saída gerada estará. A principal vantagem da codificação aritmética é a facilidade de se transformar seus modelos em adaptativos, dessa forma pode-se descobrir a distribuição probabilística de símbolos do sinal sem maiores informações a priori.

O codificador funciona atribuindo intervalos aos símbolos de entrada. Esses intervalos terão tamanhos proporcionais à probabilidade de ocorrência de cada símbolo associado como representado no exemplo da figura 3. No entanto não é necessário que o codificador transmita todo o intervalo, apenas uma fração que pertença a este intervalo. Na verdade, só é necessário transmitir dígitos suficientes (em qualquer base) dessa fração, de tal modo que todas as frações que comecem com esses dígitos pertençam ao mesmo intervalo representado. Ou seja, o codificador funciona alocando números dentro dos intervalos correspondentes aos símbolos que recebe e, subdividindo esses intervalos de acordo com o modelo dos dados a medida que cada nova codificação acontece.

Este método nada mais é do que uma generalização dos métodos tradicionais que

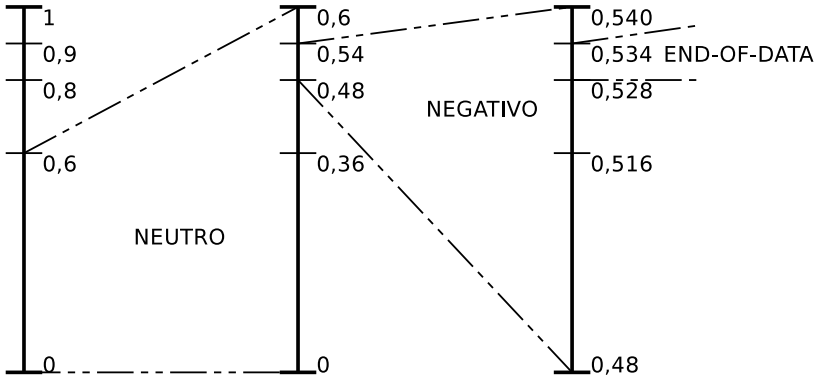


Figura 4: Processo de decodificação.

empregam códigos de comprimento variável. Os símbolos com grande probabilidade de ocorrer alocam poucos digítos, pois como a faixa utilizada para sua representação é larga, com pouca precisão pode-se encontrar uma fração que esteja contida em seu interior. Porém os símbolos raros, possuem baixa probabilidade e precisam de uma fração com muita precisão, ou em outras palavras, muitos dígitos para serem representados.

Na figura 4, um processo de decodificação será apresentado para ilustrar os passos envolvidos neste processo. Neste exemplo considera-se que a mensagem foi codificada utilizando um modelo fixo de quatro símbolos, com cada símbolo contendo uma distribuição de probabilidades com representada na Figura 4 e que a mensagem transmitida foi 0,538. Para efeitos didáticos, os números serão representados em decimal ao invés de binário. No começo, como o codificador, o decodificador inicia no intervalo de [0;1), e usando o mesmo modelo, o divide em quatro sub-intervalos. Como podemos ver, a fração 0.538 pertence ao sub-intervalo NEUTRO, [0; 0,6); isto indica que o primeiro símbolo no codificador foi NEUTRO.

Como próximo passo subdivide o intervalo [0; 0.6) em sub-intervalos e verifica que a fração recebida coincide com o intervalo [0,48;0,54); portanto o segundo símbolo da mensagem é NEGATIVO. Mais uma vez o codificador efetua a subdivisão desse intervalo e verifica que o próximo símbolo é o END-OF-DATA, representando que o processo de decodificação está completo e que a mensagem foi recuperada.

É importante notar que a mesma mensagem poderia ter sido codificada pelas frações 0,534, 0,535, 0,536, 0,537 ou 0,539, sugerindo que o uso da base decimal introduziu alguma ineficiência no processo. Essa hipotése está correta; enquanto o conteúdo de um número decimal de três digitos é aproximadamente 9,966 bits, a mesma mensagem poderia ter sido codificada como a fração binária 0,10001010 (0,5390625 em decimal) com um custo

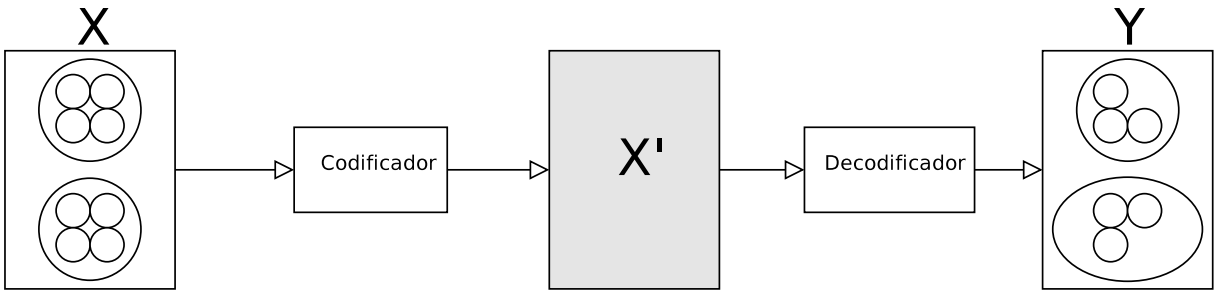


Figura 5: Compressão de sinais com perdas.

de somente 8 bits. Esse custo é ligeiramente superior à entropia, que é aproximadamente 7,381 bits e representa uma medida da quantidade de informação contida na mensagem original (HAYKIN, 2001).

## 2.4 Compressão de sinais com perdas

Neste tipo de compressão, existe diferença entre o resultado produzido pelo decodificador e a mensagem original, ou seja, Y e X não são iguais. Tal esquema é usado quando não existe a necessidade de integridade absoluta de reconstrução, ou seja, o usuário pode aceitar algum tipo de perda de informação durante o processo, como demonstrado na figura 5.

Nos esquemas sem perdas utiliza-se, como medida de desempenho e parâmetro de comparação, a taxa gerada R, porém para esquemas com perdas, conforme o nome implica, existe algum tipo de perda de informação inerente ao processo. Portanto torna-se necessário incluir outro tipo de medida para caracterizar essa diferença ocorrida entre X e Y, esta medida é chamada de distorção D. A relação entre a taxa R e a distorção D é dada pela função taxa-distorção R(D) detalhada brevemente na subseção seguinte.

De uma maneira geral, os métodos de compressão com perdas podem ser divididos em duas grandes classes: quantização e codificação por transformada. Ambas as famílias envolvem passos que são comuns a todos os codificadores. Sendo eles: a transformação, a quantização e a codificação.

Os codificadores por transformada são os únicos que introduzem o primeiro passo deste processo: a transformação da fonte de entrada. Essa transformação consiste em manipular matematicamente o sinal de entrada de tal forma que o seu resultado contenha a maior parte das informações concentrada em poucos elementos, ou seja, efetuando uma compactação da energia.

A quantização consiste no processo de representar uma grande quantidade de símbolos usando uma quantidade muito menor. Para isso o codificador escolhe em um dicionário qual de seus símbolos representa de maneira mais adequada o símbolo apresentado na mensagem original. Lembrando que essa é uma escolha por similaridade, visto que o dicionário não apresenta todos os símbolos possíveis, apenas uma representação significativa da estatistística do sinal de entrada. Existem basicamente dois tipos de quantizadores: o escalar e o vetorial. No primeiro cada símbolo é quantizado individualmente, enquanto no vetorial, os valores de entrada são agrupados em blocos ou vetores e estes que são mapeados em índices de um dicionário.

A codificação é a última parte e consiste em codificar os símbolos, fornecidos pelo quantizador, com alguma técnica de codificação sem perdas, como as citadas na seção anterior, de maneira a eliminar qualquer correlação que ainda venha a existir entre os símbolos gerados.

Um esquema de compressão de sinais com perdas fornece uma compressão maior que um esquema sem perdas.Um esquema sem perdas possui um limite teórica fundamental definido pela entropia da fonte, já no esquema com perdas o que temos é uma relação entre a taxa e a distorção. Dessa maneira, tem-se uma relação de compromisso, podendo obter taxas tão baixas quanto se queira se uma maior distorção no sinal reconstruído for aceitável.

O algoritmo desenvolvido neste trabalho pertence à classe dos algoritmos de compressão com perdas e une as etapas de transformação, quantização e codificação de entropia em um único passo, atuando de forma adaptativa na compressão dos dados. Apesar disso, vale ressaltar que o algoritmo também é capaz de comprimir dados sem perdas, desde que sua distorção alvo seja ajustada para zero.

### 2.4.1 Teoria taxa-distorção

A compactação obtida para um conjunto de dados, quando utilizam-se esquemas que inserem perdas, pode ser aumentada o quanto se queira, desde que seja aceita uma quantidade maior de distorção. Essa introdução de distorção, feita de forma controlada, dependendo da aplicação, pode fazer com que a taxa final atinja valores muito inferiores aos obtidos pelos codificadores sem compressão. Sinais de voz, são exemplos típicos onde uma fidelidade bit a bit não é necessária para que o resultado sonoro final seja semelhante ao som codificado originalmente.

Assim, introduz-se o conceito de função taxa-distorção exemplificada na Figura 6, numa tentativa de explorar de forma racional essas variáveis de projeto a fim de se obter a melhor relação custo-benefício para uma dada aplicação. Os casos extremos dessa função ocorrem em (2.1), quando a informação original é toda transimitida e a distorção é zero e em (2.2), quando a distorção é máxima e logo não há necessidade de se transmitir nenhuma informação. Vale ressaltar que o primeiro ponto coincide com os esquemas de compressão sem perdas, assim em uma análise teórica esses esquemas podem ser vistos como casos particulares dos codificadores com perdas.

$$(R_a, D_a) = (R_{max}, 0) \tag{2.1}$$

$$(R_b, D_b) = (0, D_{max}) \tag{2.2}$$

A função taxa-distorção, ou R(D), sempre especifica a menor taxa média R na qual a saída de uma fonte pode ser codificada, mantendo-se uma distorção média menor ou igual a D. Dessa maneira a função R(D) define o desempenho possível de ser atingido por qualquer código de compressão, ou seja, a desigualdade expressa em (2.3), é sempre satisfeita. Nem sempre consegue-se encontrar a função R(D) que represente uma dada fonte, e além disso, pode não ser possível obter-se um método de compressão viável que seja capaz de comprimir a saída da mesma satisfazendo determinado ponto da curva. Entretanto a função taxa-distorção serve para balizar o desenvolvimento de um codificador e demonstrar o quão próximo do ótimo ele está.

$$R_{codigo} \geq R(D) \tag{2.3}$$

## 2.5 Medidas de desempenho

Quando um novo método de compressão surge, é necessário posicioná-lo em relação a outros existentes de forma a inferir seu desempenho e guiar seu desenvolvimento. Dessa forma é necessário que sejam definidos critérios e métricas que permitam a avaliação destes trabalhos.

O primeiro parâmetro para se avaliar um codificador é a taxa de compressão que ele permite. Ela é definida como a relação entre o número de bits necessários para representar

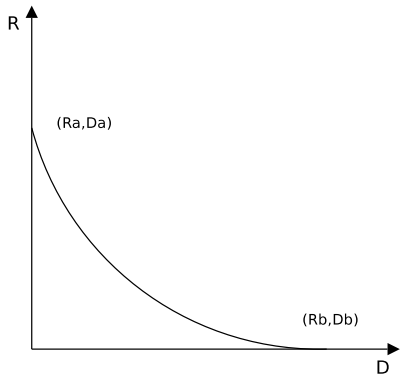


Figura 6: Função taxa-distorção.

a amostra de voz antes da compressão e o número de bits necessários para representar a mesma amostra após a compressão, ou seja:

$$CR = \frac{Tamanho\ Original}{Tamanho\ Atual} \tag{2.4}$$

Porém a forma mais comum de se representar a taxa de compressão é em termos do número médio de bits necessários para representar um segundo de amostra de voz, neste caso usa-se somente o termo taxa (R), descrita por:

$$R = \frac{[Tamanho\ Original] \cdot [Taxa\ Atual]}{Tamanho\ Atual} \tag{2.5}$$

Neste trabalho, os arquivos originais utilizados para teste estão sempre amostrados a 8 kHz utilizando 16 bits de precisão, portanto gerando incialmente uma taxa de 128 kbps.

Como o algoritmo apresentado realiza uma compressão com perdas, a próxima medida que precisa ser discutida é a distorção. Pois conforme citado anteriormente, nesse tipo de esquema a reconstrução do sinal não é perfeita e é necessário avaliar além da taxa, a qualidade do sinal reconstruído.

A maneira mais intuitiva de se inferir a distorção que foi criada no sinal, seria obter o erro médio quadrático entre a entrada e a saída do sistema avaliado. Porém esse tipo de medida é o que menos se aproxima da maneira como o ouvido humano percebe a diferença entre duas amostras de voz. Dessa maneira, um sinal decodificado pode ser extremamente fiel ao original quando avaliado por um humano e mesmo assim possuir um erro médio quadrático elevado.

Assim com o crescimento dos sistemas de comunicação e transmissão de voz nos anos

50 e o surgimento dos algoritmos de codificação digital nos anos 60, foi criado um campo de estudo na busca de se obter ferramentas que permitissem avaliar de maneira justa e qualitativa um sinal de voz. De uma maneira geral, os testes desenvolvidos desde então pertencem a duas categorias: objetivos e subjetivos. Medidas subjetivas são baseadas na comparação entre a amostra original e a codificada por um ouvinte ou por um grupo de ouvintes, que vão atribuir notas para a qualidade da voz dentro de uma escala pré-determinada. Já medidas objetivas são semelhantes ao exemplo citado anteriormente, ou seja, um valor numérico quantizando a distância entre as duas amostras através de um modelo matemático.

As medidas subjetivas são sempre preferidas na avaliação da qualidade de um sinal de voz, e entre elas, a mais famosa é conhecida por *Mean Opinion Score* ou simplesmente MOS. Este método é o mais utilizado para a comparação subjetiva de codificadores de voz. Como produto final da avaliação fornece uma nota que vai de 1 a 5, sendo a primeiro insatisfatório e a última excelente. Ele define também uma série de padrões de treinamento pelas quais os ouvintes devem ser submetidos previamente, de forma a eliminar alguma tendência que possa existir no julgamento, bem como requisitos que definem o número mínimo de participantes e as condições da audição.

Porém as medidas subjetivas, apesar de representarem a maneira como os humanos percebem a distorção introduzida no som, apresentam algumas dificuldades práticas. São medidas que envolvem custos financeiros, seja com a preparação do ambiente, com a disponibilidade de pessoas ou com o tempo necessário para sua realização. Dessa maneira, seria difícil julgar alterações incrementais em um algoritmo em desenvolvimento como o proposto neste trabalho. Além disso, é uma medida estatística, ou seja, possui uma margem de erro inerente. O que dentro de um processo restrito tornaria difícil a comparação de resultados entre trabalhos e a repetição dos resultados obtidos por outros pesquisadores. Assim neste trabalho optou-se por utilizar uma classe de medidas objetivas que buscam criar um modelo capaz de prever o efeito subjetivo das distorções encontradas nas amostras de voz conhecidas como medidas pseudo-subjetivas de qualidade.

### 2.5.1 Medidas pseudo-subjetivas de qualidade

As medidas pseudo-subjetivas buscam ir além da obtenção de uma quantização da distância ou medida de similaridade entre duas amostras e contornar esses problemas de ordem prática através de modelos que buscam aumentar a correlação entre os seus resultados e testes realizados com ouvintes reais. Dessa forma, fornecem resultados que

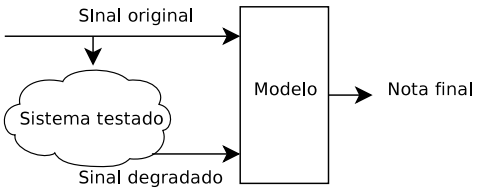


Figura 7: Medidas objetivas de qualidade de áudio.

podem ser comparados diretamente com seus equivalentes subjetivos com um alto grau de confiança.

Estes modelos, conhecidos de maneira geral por modelos de mascaramento perceptual, são utilizados para distinguir entre distorções que vão ser audíveis das que serão inaudíveis a nossos ouvidos e são comprovadamente o melhor método de prever o quão perturbador será o efeito de distorções complexas nas amostras analizadas. De uma maneira geral os testes objetivos, e entre eles os pseudo-subjetivos, funcionam conforme o esquema da Figura 7, comparando a amostra não processada com a saída do sistema analisado.

Em 1996, após um longo estudo internacional, os modelos perceptuais para avaliar a qualidade de amostras de voz foram padronizados e geraram a recomendação P.861 do ITU-T. O modelo foi resultado do trabalho de John Beerends, um pesquisador do KPN Research Lab e foi conhecido como PSQM.

Porém, após algum tempo, descobriram situações onde a correlação entre as medidas subjetivas e os resultados do algoritmo PSQM tinham um valor muito baixo. Estes estudos motivaram a padronização de um novo algoritmo de teste conhecido como PESQ, que tinha como intuito corrigir esses problemas encontrados. Como resultado o ITU-T homologou a recomendação P.862 em fevereiro de 2001, revogando a P.861.

### 2.5.2 PESQ (ITU-P.862)

O algoritmo PESQ, por ser um padrão mundial para a medida qualitativa de amostras de voz, foi utilizado durante a elaboração deste trabalho como critério de medida de distorção nas comparações realizadas com o codificador proposto. Sua escolha se justificou pela sua facilidade de uso e por sua alta correlação com os resultados dos tradicionais testes MOS.

O modelo é composto de diversos blocos conforme a estrutura representada na figura 8. Os estágios são :

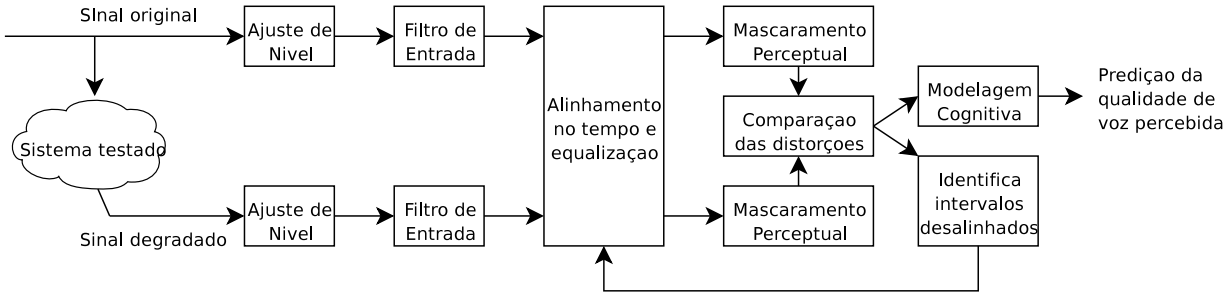


Figura 8: Estrutura do PESQ (ITU-P.862).

### Normalização

Como primeira etapa de processamento, temos o alinhamento de nível do sinal. Essa etapa é responsável por normalizar a potência do sinal de entrada para um valor de referência, normalmente o mesmo valor utilizado nos testes subjetivos.

### Filtro de entrada

O sinal é então filtrado para compensar distorções no espectro notóriamente introduzidas pela rede telefônica ou pelos *handsets* dos aparelhos telefônicos.

### Alinhamento temporal

O sistema testado pode incluir um atraso, muitas vezes variável durante o teste, como seria o exemplo de uma transmissão de voz por IP. Para compensar esse efeito, a amostra é particionado em blocos e o algoritmo tenta identificar similaridades entre eles e compensar esses atrasos.

### Mascaramento perceptual

Ambos os sinais passam por uma transformação que busca mapear as propriedades da audição humana. Os ruídos abaixo dos limiares de audição em um dado momento são eliminados, pois não contribuiriam para o resultado final em uma audição puramente subjetiva.

### Modelagem cognitiva

Calcula a distância entre os dois sinais considerando uma média não-linear entre duas classes: absoluto ou simétrico, que mede o erro audível absoluto e aditivo ou asimétrico, que mede erros audíveis muito mais intensos do que o referência.

Essa distância é convertida então para uma nota PESQ, que varia em uma faixa de -0,5 a 4,5. Neste trabalho foi utilizado também uma função de mapeamento para possibilitar a comparação direta entre a nota PESQ e a tabela de resultados MOS conforme normatizado

Tabela 1: Escala de qualidade

| Nota | Qualidade da voz |
|---|---|
| 5 | Excelente |
| 4 | Bom |
| 3 | Razoável |
| 2 | Ruim |
| 1 | Péssimo |

pelo ITU-T P.800. Essa função representada em (2.6), está contida na norma P.862.1 e converte a nota PESQ em uma nota PESQ-LQ que varia entre 1 e 5 conforme a tabela 1.

$$PESQ_{LQ} = 0.999 + \frac{4}{1 + e^{-1.4945PESQ+4.6607}} \tag{2.6}$$

## 2.6 Conclusão

Este capítulo buscou fundamentar a teoria que será utilizada durante o restante deste trabalho. Apresentou e definiu as famílias de compressão existentes, mais especificamente o método de codificação aritmética e a teoria taxa-distorção. Foram apresentados também os conceitos de medição de qualidade de sinais de voz que são utilizados para comparar os codificadores com perdas, entre eles os padrões de testes do ITU-T utilizados para avaliar esta implementação do MMP.

# 3 Método de recorrência de padrões multiescala

## 3.1 Introdução

Este capítulo fará uma descrição detalhado do método de recorrência de padrões multi-escala desenvolvido inicialmente em [(CARVALHO, 2001), (DUARTE, 2002)]. O algoritmo MMP, de uma maneira genérica, não parte de nenhuma premissa *a priori* sobre o sinal que será codificado, aprendendo sua estatística durante o processo. Deste modo, os passos e conceitos descritos abaixo são de um codificador universal, sem nenhuma especialização no problema abordado neste trabalho.

Na seção 3.2 encontram-se descritas as estruturas básicas presentes no algoritmo, em três subseções, analisando respectivamente a inicialização do dicionário, o particionamento e a codificação dos vetores e os métodos de atualização utilizados. Na seção 3.3, o conceito de análise de custo através do critério de taxa-distorção é introduzido e na seção 3.4 o algoritmo completo do MMP é detalhado, englobando tanto o processo de codificação como o de decodificação.

## 3.2 Algoritmo MMP

O MMP representa uma nova classe de algoritmos de compressão com perdas, baseado em um casamento aproximado de padrões recorrentes com escalas diferentes. O termo recorrente se deve à possibilidade de particionar o sinal de entrada em diferentes escalas, buscando sua melhor representação. Nesse sentido, o algoritmo pode ser visto como uma expansão do casamento de padrões comuns ou quantização vetorial, onde nesta nova proposta, existe a possibilidade de se casar vetores de diferentes tamanhos. O MMP pode funcionar com sinais unidimensionais ou de mais dimensões de maneira transparente, como o escopo desse trabalho é codificar voz, sua abordagem estará restrita ao primeiro caso.

Para realizar este casamento entre vetores de tamanhos diferentes o MMP utiliza transformações de escalas, que podem ser a união ou expansão de vetores menores ou a contração no caso de vetores de tamanho maior. Dessa forma, o dicionário inicial utilizado pelo MMP para representar o sinal é atualizado ao longo do processamento com novos vetores. Estes novos vetores são aproximações melhores dos padrões presentes no sinal já processado e representam melhor a estatística encontrada na entrada. Assim, pode-se dizer que o processo de quantização realizado no algoritmo é adaptativo, pois busca aprender a dinâmica do sinal durante seu funcionamento.

O objetivo deste processo é que o codificador consiga atingir determinado critério de distorção determinado pelo usuário utilizando a menor quantidade de bits possível. Para isso, o codificador tenta representar o sinal de entrada através dos maiores vetores existentes em seus dicionários. Porém nem sempre é possível encontrar um casamento que satisfaça a distorção exigida, então, tenta-se representar o sinal de entrada através de sinais menores, até que se satisfaça algum critério de parada. Como essas subdivisões melhoram a qualidade da reconstrução mas aumentam a taxa de bits final o algoritmo utiliza um critério de taxa e distorção [(CARVALHO; SILVA; FINAMORE, 2001)] para otimizar o ponto de parada, garantindo a melhor representação possível para um determinado conjunto de requisitos.

### 3.2.1 Inicialização do dicionário

O dicionário utilizado no processo de codificação é composto na verdade por vários subdicionários $S_n$ com dimensões específicas. O número de subdicionários é limitado e definido em função da largura inicial do bloco que será processado, que é um parâmetro fixo, definido previamente pelo usuário. Como as divisões realizadas pelo codificador particionam os blocos na metade, os subdicionários são equivalentes aos subníveis de uma árvore binária, como ilustrado na Figura 9. Podemos relacionar o número de dicionários $K$ ao comprimento do bloco $C_b$ através da relação:

$$K = \log_2 (C_b) + 1 \tag{3.1}$$

Assim construímos um dicionário inicial com vetores cujas amplitudes representam a faixa dinâmica do sinal desejado. Lembrando que esta escolha pode implicar em uma distorção implícita ao algoritmo, pois pode-se optar por alterar o passo de quantização do sinal de entrada escolhendo um número menor de vetores iniciais do dicionário. Um ponto

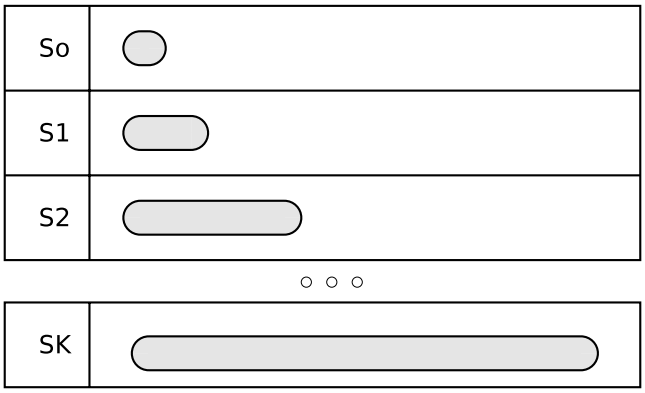


Figura 9: Estrutura de subdivisão do dicionário em níveis de diferentes escalas.

interessante do algoritmo, é que os processos de atualização do dicionário, mostrados mais adiante, podem gerar níveis que não existiam previamente no dicionário e dessa maneira, ele pode se especializar para uma determinada faixa dinâmica de um sinal ocasional que não tinha sido prevista no projeto do codificador.

Para definirmos a estrutura completa do dicionário, é preciso definir além do comprimento do bloco $C_b$, o valor máximo, mínimo dos vetores e o passo de quantização desejado considerando os critérios expostos acima. O número de vetores existentes inicialmente no dicionário pode ser obtido pela relação a seguir:

$$N_{vetores} = \frac{max\,(X) - min\,(X)}{passo} + 1 \tag{3.2}$$

Os vetores iniciais serão obtidos através da relação abaixo, onde $U(t)$ é um degrau unitário, $k$ é o nível do dicionário, $K$ é o número de níveis e $n$ é o índice do vetor.

$$S_{k,n} = [U(t) - U(t - 2^k)] \cdot n \cdot passo\ ,\ \text{para}\ n = (0, 1, ..., N_{vetores} - 1)\ \text{e}\ k = (0, 1, ..., K) \tag{3.3}$$

### 3.2.2 Particionamento e codificação

Após a definição dos parâmetros da estrutura do dicionário e de sua inicialização, o codificador está preparado para receber o sinal de entrada e começar a codificação propriamente dita. Quando utilizamos o MMP, o codificador considera individualmente cada bloco de tamanha $C_b$ da entrada como na Figura 10. A codificação deste bloco é realizada usando-se a melhor aproximação deste bloco no dicionário de maior dimensão, neste caso o de nível 0, como representado na Figura 11.

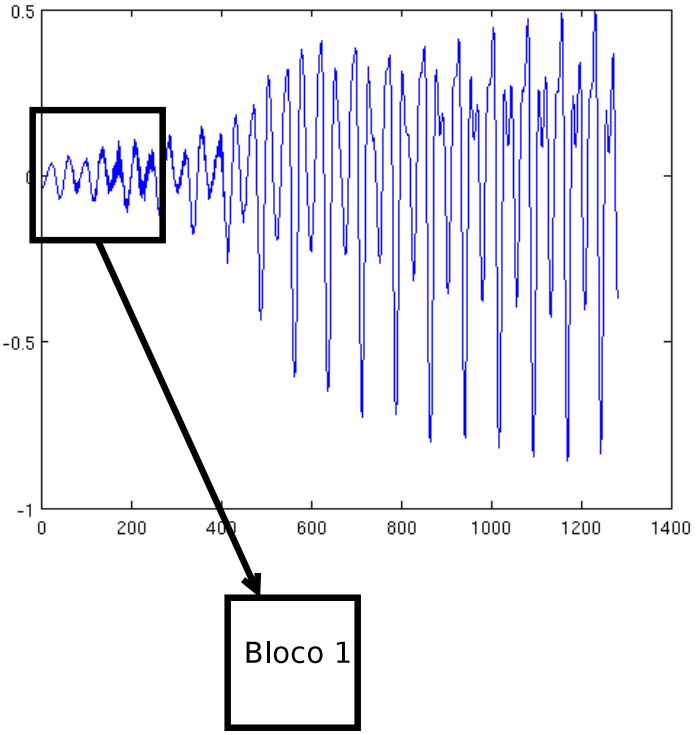


Figura 10: Exemplo de particionamento do sinal de entrada em blocos.

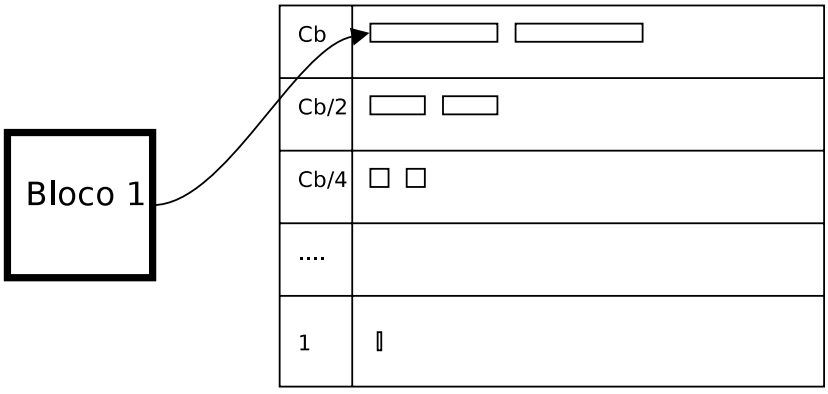


Figura 11: Determinação da melhor aproximação disponível no dicionário.

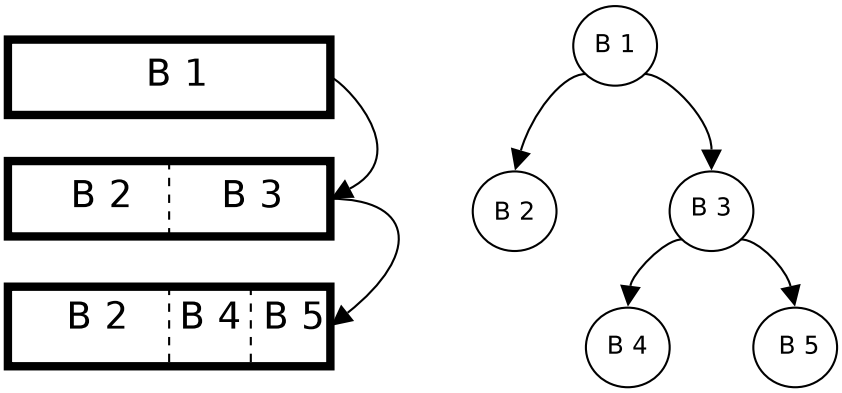


Figura 12: Subdivisões realizadas pelo codificador no bloco inicial.

Essa tentativa de codificação gera uma medida de distorção que será comparada em relação a um critério de parada. Caso ela o satisfaça, o codificador indica isso para o *stream* de saída com um *flag* 1. Em seguida gera na saída o índice do bloco escolhido no dicionário de nível correspondente. Vale ressaltar que para o nível de menor dimensão do dicionário, pelo fato de ser o último nível, não é necessário a indicação que se trata de um nó terminal da árvore. Dessa forma, o codificador pode economizar alguns bits com os *flags* e representar diretamente o índice do vetor escolhido.

Quando o critério de controle não é satisfeito, ocorre o particionamento do bloco. O codificador precisa indicar esse evento no *stream* de saída com um *flag* 0, de maneira que o decodificador possa repetir seus passos no processo inverso. Dois blocos de tamanho $\frac{C_b}{2}$ são gerados e o processo descrito anteriormente é repetido, considerando desta vez os novos blocos separadamente e comparando-os ao nível inferior do dicionário.

Essa divisão em subblocos com a metade do tamanho pode ser associada com a geração de uma árvore binária que estará representada pelos *flags* 0s e 1s no *stream* de saída do codificador. Esta árvore representada na figura 12, será reconstruída pelo decodificador para saber de qual nível do dicionário corresponde cada índice que ele recuperará e o permita reconstruir o sinal transmitido.

### 3.2.3 Atualização do dicionário

Quando a codificação de um determinado subbloco satisfaz o critério de distorção estipulado pelo usuário, o codificador além de gerar os símbolos necessários na saída (*flag* de parada e índice do dicionário), atualiza o dicionário. Essa atualização implica na inserção de um novo vetor no dicionário de nível imediatamente superior. Considerando a

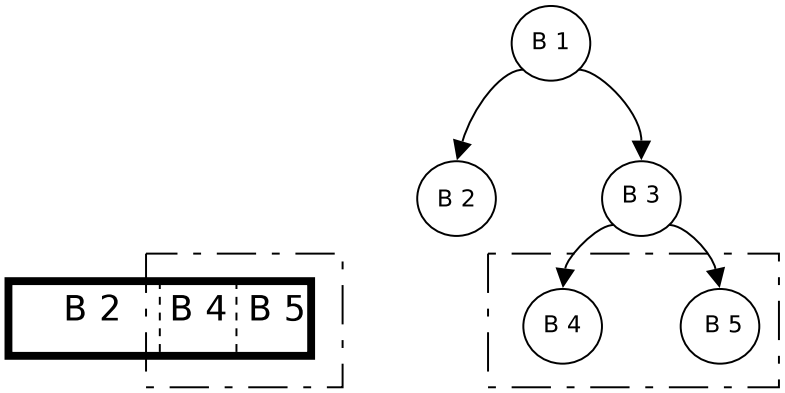


Figura 13: Primeira etapa de atualização do dicionário, inserindo um novo vetor no nível intermediário.

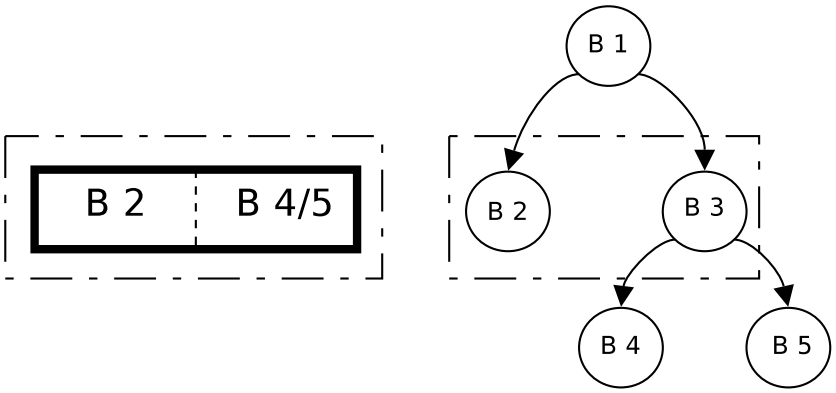


Figura 14: Segunda etapa de atualização onde o codificador insere um novo vetor no nível superior.

estrutura de árvore binária, pode-se dizer que a atualização ocorre quando dois nós-filhos do mesmo nó pai tiverem sido codificados pelo MMP (nós disponíveis).

A atualização é então realizada com a concatenação dos blocos já codificados correspondentes ao nós disponíveis e sua inserção no nível do nó-pai. Esse bloco concatenado sofre também transformações e é inserido em todos os outros níveis do dicionário.

Na figura 13 está representada a concatenação dos nós $B_4$ e $B_5$ já codificados formando um novo vetor $B_{4/5}$ que será inserido no dicionário. Em uma segunda etapa quando o outro ramo de codificação for concluído, o nó $B_2$ estará disponível e ele será concatenado a seu irmão e formará um novo vetor $B_{2/4/5}$ que será inserido no primeiro nível do dicionário, como representado na figura 14.

Dessa forma, quando outro bloco semelhante aparecer no sinal de entrada, o codificador será capaz de realizar a codificação utilizando os índices dos novos vetores sem ter que recorrer a subdivisões. Essa nova codificação gerará uma redução na quantidade de

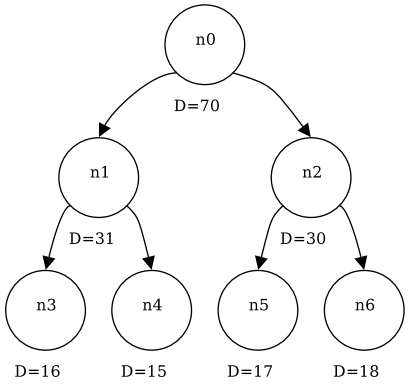


Figura 15: Exemplo de árvore binário contendo os valores de distorção associados.

símbolos utilizados na saída mantendo a distorção dentro dos níveis exigidos pelo usuário.

## 3.3 MMP com otimização taxa-distorção

O codificador precisa utilizar um critério de distorção para decidir se um bloco será quantizado ou se deve ser particionado. Como exemplo, será considerada a árvore $A(n_0)$ representada na figura 15. Na primeira hipótese, o critério de parada é que o valor de distorção seja menor que 29. Esse critério é local e analiza apenas o bloco sendo tratado no momento. Ele é o método que apresenta a menor complexidade possível, porém pode gerar alguns problemas.

Segundo o método proposto, a árvore $A(n_0)$, seria completamente codificada, pois somente os nós terminais possuem distorção abaixo de 29, o que geraria 3 *flags* indicando as divisões na árvore e 4 índices com uma distorção total de 66. O que este algoritmo não considera, é que a divisão de $n_1$ aumentou a taxa sem proporcionar melhoria na distorção e que a divisão do nó $n_2$ foi ainda pior e aumentou a distorção final.

Portanto, analisando esse tipo de exemplo, fica evidente o surgimento de duas necessidades: um critério de decisão global e a inserção da taxa na formação do custo de cada nó. Considerando esses dois critérios a divisão da árvore passaria da representada na Figura 16 para a Figura 17 e, o resultado final passaria a gerar 3 *flags* e 2 índices apenas, com uma distorção associada de 61.

O critério global proposto no MMP é baseado no custo lagrangeano. A otimização pelo método de Lagrange define uma relação entre a taxa e a distorção em um determinado nó como:

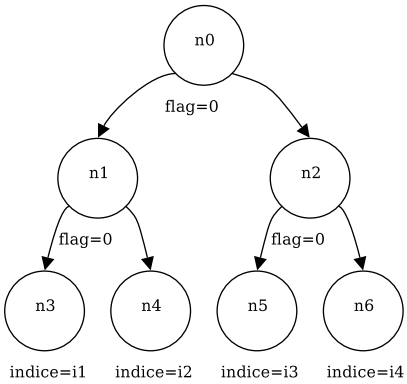


Figura 16: Exemplo de utilização do critério local de divisão.

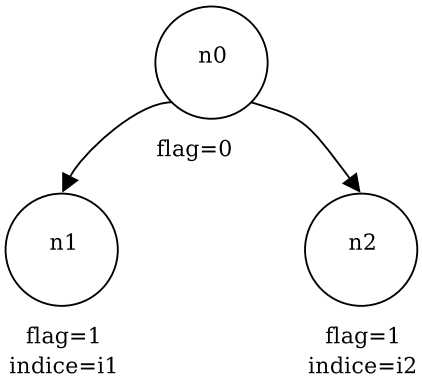


Figura 17: Exemplo de utilização do critério global de divisão

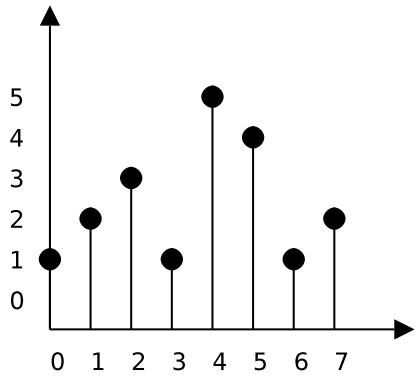


Figura 18: Exemplo de sinal de entrada.

$$J_(n_x) = D(n_x) + \lambda R(n_x) \tag{3.4}$$

Onde $n_x$ é o nó em questão e $J(n_x)$ é o custo total. $D(n_x)$ é a distorção entre o bloco $B_x$ associado a este nó e a melhor representação $s_i$ encontrada no subdicionário $S$. $R(n_x)$ é a quantidade de bits associada a codificação do nó e $\lambda$ é um parâmetro que será ajustado pelo usuário e definirá se o algoritmo deve priorizar taxa, distorção ou algum ponto intermediário nesta curva. Se $\lambda$ for zero, o MMP funcionará como um codificador sem perdas, pois independente da taxa, ele buscará a distorção nula.

Utilizando o custo $J(n_x)$ exposto em 3.4, o codificador pode definir a *poda* ou *descarte* dos nós de maneira global comparando seu custo com o de seus filhos. Em outras palavras, um determinado nó $n(p)$ só será subdividido em $n_e$ e $n_d$, se a soma de seus custos for menor que a original.

## 3.4 Algoritmo MMP

Nesta seção os passos detalhados nas seções anteriores serão ordenados e o funcionamento do MMP será detalhado. Para descrever o algoritmo completo será apresentado um exemplo e os passos do codificador serão ilustrados considerando o bloco representado na Figura 18.

Por esta figura pode-se inferir que o bloco máximo processado neste exemplo contém oito amostras e sua faixa dinâmica varia de 0 a 5. Para este exemplo o passo do dicionário será escolhido como 1, dessa forma os parâmetros para a construção e inicialização do dicionário já estão definidos. Os vetores iniciais do dicionário são escolhidos com o tamanho

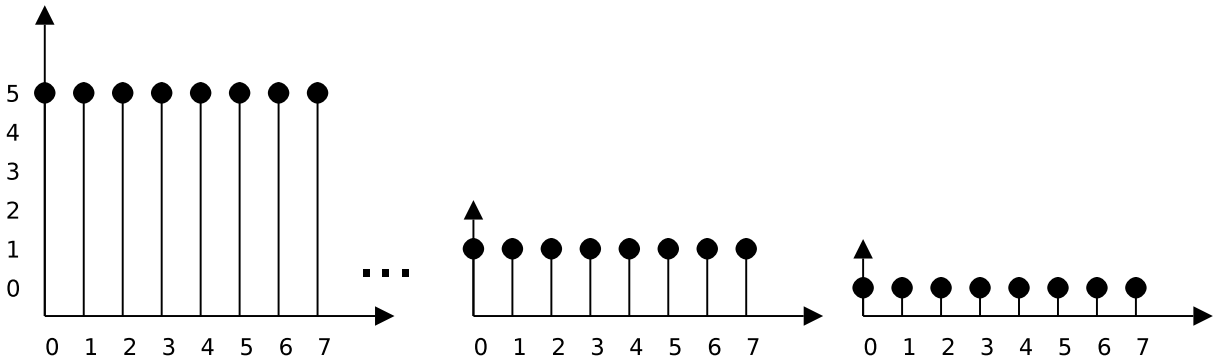


Figura 19: Vetores do nível superior do dicionário inicial.

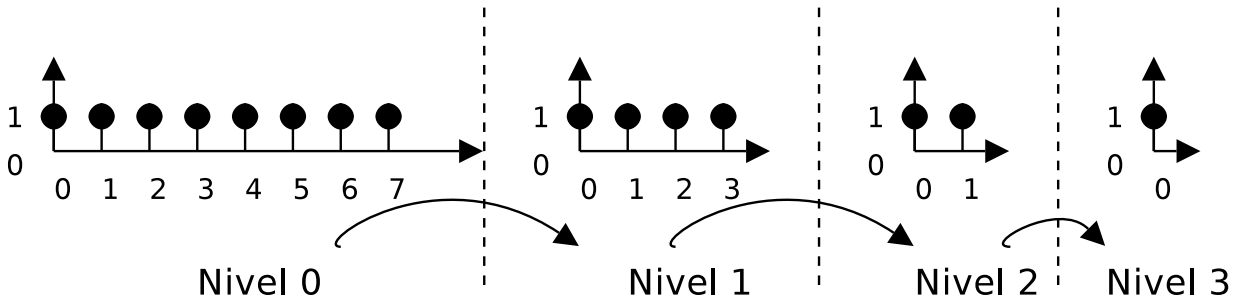


Figura 20: Divisões dos vetores do nível superior do dicionário.

máximo do bloco e de forma a conter toda a faixa dinâmica do sinal, com os vetores distanciados pelo valor do passo, como representado na Figura 19.

Após popular o nível superior do dicionário inicial, os demais níveis serão preenchidos com a subdivisão desses vetores como exemplificado na Figura 20. Essas divisões são realizadas até que o nível terminal seja preenchido com um vetor de somente uma amostra. Lembrando que cada nível do dicionário possui apenas vetores do mesmo tamanho, assim construímos um dicionário de múltiplas escalas. Cada vetor inserido no dicionário recebe um índice que o identifica dentro daquele nível.

### 3.4.1 Codificação

O MMP inicia o processo de codificação tentando aproximar o bloco do sinal de entrada por um vetor de mesmo tamanho. Para isso ele procura no dicionário superior qual vetor o representa com a menor distorção. Diversas técnicas podem ser propostas para medir a distância entre esses vetores considerando características psico-acústicas, porém neste exemplo utilizou-se uma medida de distância euclidiana entre os dois vetores

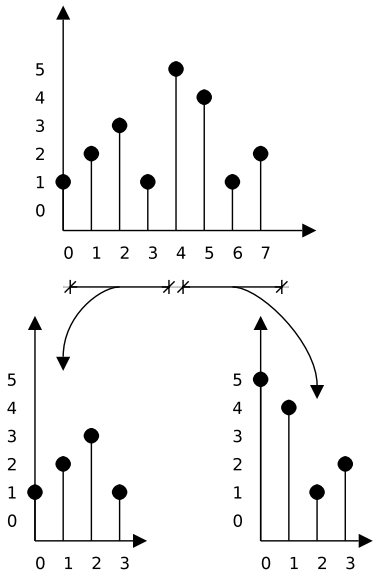


Figura 21: Primeiro particionamento do vetor de entrada.

comparados.

Para o exemplo em questão, a melhor representação no nível 0 seria o vetor que contém todos os elementos iguais a 2, o que geraria um erro médio quadrático associado de 19. O codificador em seguida particiona este bloco em dois subblocos de 4 amostras como representado na Figura 21. O procedimento de busca é refeito, desta vez no dicionário que possui vetores de 4 amostras, para cada um dos subblocos. Nesta etapa os melhores casamento obtidos são com os vetores que possuem os elementos iguais a 2 para o primeiro subbloco e 3 para o segundo, com um erro médio quadrático de 3 e 10 respectivamente.

Neste ponto o codificador define através da otimização taxa-distorção apresentada na seção anterior na equação (3.4) se deve prosseguir. O codificador verificará se o custo associado ao nó pai é superior a soma dos custos dos nós filhos como representado na Figura 22. Esse teste é feito através do critério definido abaixo:

$$J_{pai} \leq J_{filhoesquerda} + J_{filhodireita} \tag{3.5}$$

Caso o teste seja verdadeiro o codificador decide que o melhor seria interromper a codificação sem mais divisões. Porém para este exemplo, assume-se que o $\lambda$ escolhido pelo usuário foi pequeno, ou seja, o codificador deve privilegiar a redução da distorção e o algoritmo prossegue de maneira recursiva, descendo de nível e codificando os dois subblocos independentemente. O codificador deve indicar essa decisão no *stream* de saída através de um *flag* 0, sinalizando que está trabalhando agora no nível inferior da árvore.

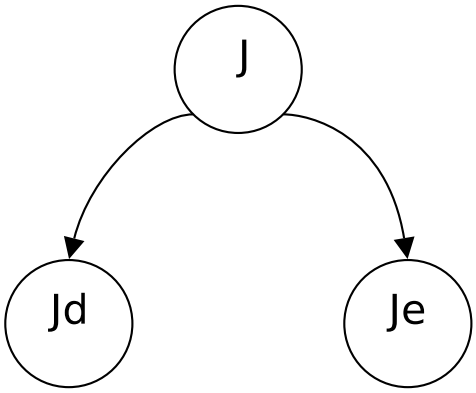


Figura 22: Análise do custo de cada nó da árvore para decidir o ponto de parada.

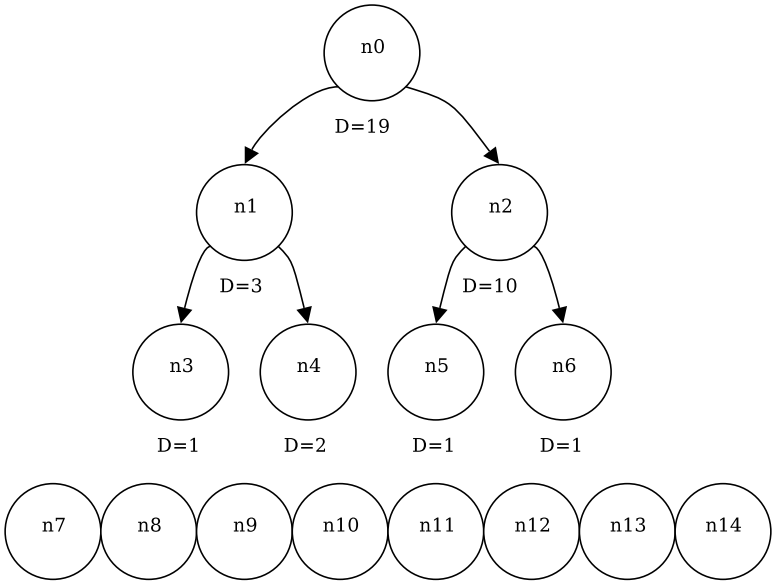


Figura 23: Estrutura de divisões do exemplo com suas respectivas distorções.

A árvore de divisões representada na Figura 23 é percorrida sempre começando pelos filhos à esquerda, em outras palavras, considerando apenas os três primeiros níveis teríamos por exemplo, a seguinte seqüência: 0, 1, 3, 4, 2, 5 e 6. Neste ponto o codificador já possui a informação sobre o melhor casamento e o custo associado para estes dois subblocos ($n_1$ e $n_2$) de 4 amostras. Então ele vai subdividir os mesmos em dois vetores de 2 amostras como em 24 e verificar qual distorção seria encontrada para essas representações. O mesmo teste descrito em 3.5 será refeito para esses nós e para este exemplo assume-se que o resultado foi novamente falso. Mais uma vez o codificador indica a decisão de dividir o nó $n_1$ com um *flag 0*.

A análise se repete de maneira recursiva, e neste nível o codificador comparará os subblocos de 2 amostras com suas respectivas divisões em vetores de 1 amostra, como

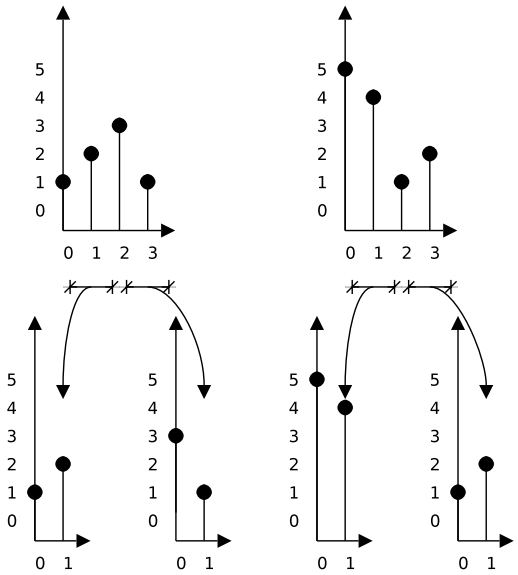


Figura 24: Divisão do sinal exemplificado em blocos de 2 amostras.

representado na figura 25. O codificador começa a analisar o nó $n_3$ , e encontra um casamento com distorção nula se representá-lo através das divisões $n_7$ e $n_8$, pois o dicionário contém amostras de toda a faixa dinâmica do sinal. Assim a divisão é novamente indicada através de um *flag* 0. Neste último nível, o codificador se limita a encontrar o melhor casamento para cada nó e indicar o índice do dicionário no *stream* de saída, já que não há a necessidade de se indicar explicitamente que não houve divisão.

Toda vez que a codificação de dois nós filhos é concluída, o codificador ao retornar ao nó pai atualiza o dicionário. No exemplo, ao retornar ao nó $n_3$ o dicionário é atualizado com a concatenação dos vetores utilizados em $n_7$ e $n_8$ no nível de 2 amostras e também com suas expansões, através de interpolação linear, nos outros níveis.

O processo continua de maneira similar percorrendo a árvore da maneira descrita anteriormente. Porém ao chegar no nó $n_6$ o codificador encontrará no dicionário de 2 amostras o vetor que foi inserido na primeira atualização. Assim ao comparar o custo associado a representação através do nó $n_6$ com a dos nós filhos $n_{13}$ e $n_{14}$, o codificador optará pela primeira. Essa opção de não dividir os nós precisa ser representada explicitamente para o decodificador através de um *flag* 1, indicando que $n_6$ é uma folha da árvore. Em seguida o codificador indica o índice referente ao casamento encontrado no dicionário e o processo de codificação deste bloco está concluido e a seguinte representação foi obtida:

$$0 \;\; 0 \;\; 0 \;\; i_7 \;\; i_8 \;\; 0 \;\; i_9 \;\; i_{10} \;\; 0 \;\; 0 \;\; i_{11} \;\; i_{12} \;\; 1 \;\; i_6 \tag{3.6}$$

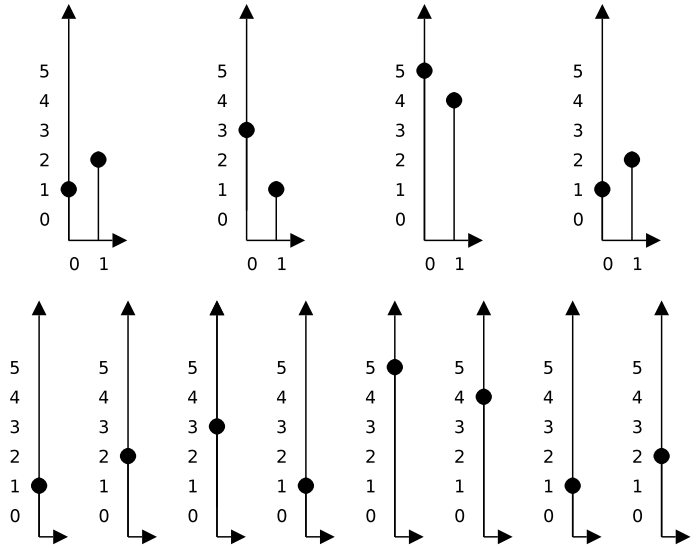

Figura 25: Divisão do sinal em blocos de 1 amostra.

Pode-se perceber que para este exemplo específico, como o dicionário estava em seu estado inicial, o codificador encontrou uma representação pouco eficiente do bloco recebido. Porém para os blocos seguintes, com a população crescente e inteligente do dicionário, casamentos de vetores maiores se tornarão cada vez mais freqüentes e conseqüentemente menos divisões serão necessárias para representar o sinal com a mesma distorção.

O algoritmo descrito acima pode também ser representado de maneira genérica através do pseudo-código a seguir:

**Codificação**

1: **loop**
2: BlocoPai ← bloco a ser codificado
3: Profundidade = Maxima
4: CodigoPai ← melhor representacao no nivel Profundidade para BlocoPai
5: Calcula CustoPai
6: CodificacaoRecursiva(BlocoPai, CustoPai, CodigoPai, Profundidade)
7: **end loop**

**Codificação Recursiva**

1: **if** Profundidade=0 **then**
2: Escreve codigo e sai
3: **end if**
4: Calcula custo de codificação do pai

5: BlocoEsquerdo ← BlocoPai[1 : $\frac{Tamanho}{2}$]
6: BlocoDireito ← BlocoPai[$\frac{Tamanho}{2} + 1 : Tamanho$]
7: CodigoEsquerdo ← melhor representacao no nivel Profundidade para BlocoEsquerdo
8: Calcula CustoEsquerdo
9: CodigoDireito ← melhor representacao no nivel Profundidade para BlocoDireito
10: Calcula CustoDireito
11: Calcula custo de codificação dos filhos
12: **if** CustoFilhos ≥ CustoPai **then**
13: Escreve flag de nó terminal
14: Escreve código
15: Atualiza estatística, dicionários e sai
16: **else**
17: Escreve flag de divisão
18: CodificacaoRecursiva(BlocoEsquerdo, CustoEsquerdo, CodigoEsquerdo, Profundidade - 1)
19: CodificacaoRecursiva(BlocoDireito, CustoDireito, CodigoDireito, Profundidade - 1)
20: **end if**

### 3.4.2 Decodificação

O decodificador MMP receberá o *stream* de dados gerados pelo codificador, que no exemplo acima está ilustrado em (3.6). Ao iniciar a decodificação é importante que os parâmetros do dicionário sejam os mesmos utilizados no codificador, para que a reconstrução de seu dicionário através das atualizações ocorra de maneira idêntica.

Ao receber os três primeiros *flags* zeros o decodificador percorre a árvore da Figura 23 até atingir o nó $n_7$, sabendo que este é o último nó da árvore ele espera receber um índice de dicionário. Encontra $i_7$ e a primeira amostra do sinal é recuperada. Continuando o processamento ainda no mesmo nível ele recupera o índice $i_8$ e sobe de nível ao nó $n_3$. Neste momento a atualização do dicionário é feita de maneira análoga à realizada no codificador, fazendo com que os dicionários de ambos permaneçam sempre idênticos.

O decodificador sobe ao nó $n_1$ e passa a analisar o ramo direito dessa sub-árvore. Como recebe novamente um *flag* 0 indicando que houve divisão, retorna ao nível inferior da árvore e recupera $i_9$ e $i_{10}$. O algoritmo continua, percorrendo a árvore da mesma maneira até chegar ao nó $n_6$. Neste nó, de maneira diferente dos outros o decodificador recebe explicitamente uma sinalização que deve interromper o processo de divisão através

de um *flag* 1. Ele recupera o índice $i_6$ e a decodificação do bloco está completa.

O processo de decodificação completo pode ser estruturado de maneira mais clara no pseudo-código abaixo:

**Decodificação**

```
1: Profundidade = Maxima
2: loop
3:    DecodificacaoRecursiva(Profundidade)
4: end loop
```

**Decodificação Recursiva**

```
1: if Profundidade=0 then
2:    Recupera código
3:    Escreve vetor
4: end if
5: Recupera Flag
6: if Flag=0 then
7:    Recupera código
8:    Escreve vetor
9: else if Flag=EOF then
10:   Fim do arquivo
11: else if Flag=1 then
12:   DecodificacaoRecursiva(Profundidade - 1)
13:   DecodificacaoRecursiva(Profundidade - 1)
14: end if
15: Atualiza estatística, dicionários e sai
```

## 3.5 Conclusão

Este capítulo apresentou os algoritmos envolvidos no método de recorrência de padrões multiescala. Os passos e conceitos de um codificador universal foram descritos, sem detalhar nenhuma especialização particular do problema abordado neste trabalho. Forma descritas as estruturas básicas presentes no algoritmo, incluindo a análise da inicialização do dicionário, do particionamento e da codificação dos vetores e dos métodos de atualização utilizados. Introduziu-se o conceito de análise de custo através do critério de taxa-distorção e, após essas definições, o algoritmo completo do MMP foi detalhado.

# 4 MMP aplicado a voz

## 4.1 Introdução

O algoritmo MMP apresentado no último capítulo foi desenvolvido inicialmente com o intuito de codificar imagens e foi apresentado no trabalho (CARVALHO, 2001). Porém seus conceitos são genéricos o suficiente, para permitir que com simples alterações o algoritmo passe a trabalhar com sinais unidimensionais. Isto, aliado ao fato de não utilizar necessariamente informações *a priori* sobre os sinais codificados, torna trivial sua adaptação inicial a codificação de sinais de voz proposta no trabalho. Ressaltando que, para atingir os níveis de excelência dos algoritmos considerados estado da arte, outros módulos específicos precisam ser trabalhados.

Neste trabalho buscou-se implementar o algoritmo básico do MMP proposto e, de maneira incremental, adicionar alguns desses blocos que permitissem uma maior adaptação as características dos sinais de voz. Possibilitando assim que se buscasse maiores taxas de compressão para um determinado nível de distorção obtido.

Este capítulo apresenta na seção 4.2 uma motivação ao seu desenvolvimento, seguida pela descrição da implementação do código do codificador na seção 4.3. A seção 4.4 contém detalhes sobre as escolhas realizadas no dicionário e a seção 4.5 descreve o codificador aritmético utilizado. Por último, na seção 4.6 a técnica de pós-filtragem adaptativa para redução da blocagem é detalhada.

## 4.2 Motivação

Os codificadores da família CELP e similares, que combinam vocoders com quantização da forma de onda, são provenientes de um conceito amplamente estudado. Evoluíram de maneira gradativa até atingir o estágio atual de compressão, qualidade e complexidade computacional. Para determinados requisitos, principalmente os que residem em um

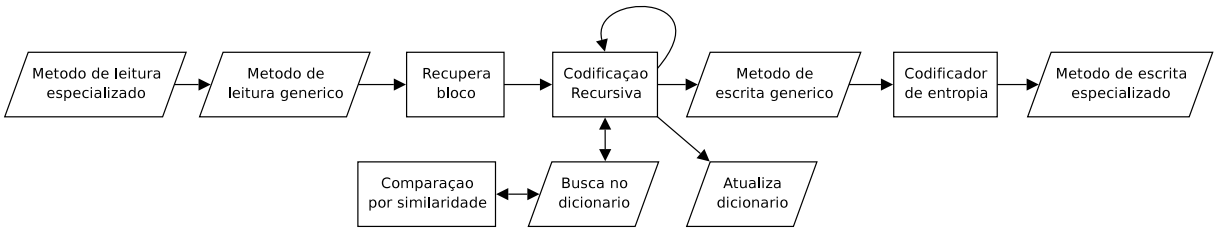


Figura 26: Esquemático de implementação do codificador

estágio intermediário entre compressão e qualidade de voz, essa família apresenta uma excelente relação de compromisso.

Para as aplicações extremamente rigorosas com a taxa de bits final, novas linhas de vocoders que funcionam basicamente através de reconhecimento, transcrição e síntese são propostos. Este campo apesar de promissor, quando analisado sobre a ótica de taxas simplesmente, não é capaz de apresentar uma reprodução fiel das entonações originais presentes na voz humana. Portanto sua aplicação se limitaria atualmente ao campo militar ou de extrema mobilidade.

A aplicação deste trabalho é realizar um estudo de viabilidade de uma família de codificadores com uma abordagem diferente das atuais. É uma tentativa de se obter uma codificação adaptada e eficiente para voz, porém que possa processar outras fontes de maneira transparente, gerando uma conversa mais natural em ambientes não-convencionais onde a contaminação da voz é inevitável. Permitindo também que seu funcionamento sejá escalonável linearmente para situações onde as limitações de taxa não sejam tão restritivas.

## 4.3 Implementação

A implementação prática do algoritmo do codificador exposto anteriormente esta representado na figura 26 e a do decodificador na figura 27. O código foi implementado em C/ANSI puro, utilizando a biblioteca GNU Scientific Library para realizar de forma otimizada os cálculos vetoriais.

A implementação foi estruturada utilizando os conceitos de orientação à objeto, com todos os parâmetros do codificador pertencentes à uma estrutura central e com funções específicas para recebê-la. A mesma abordagem foi utilizada para o decoficador, sendo que parte da biblioteca de funções foi implementada de maneira comum aos dois.

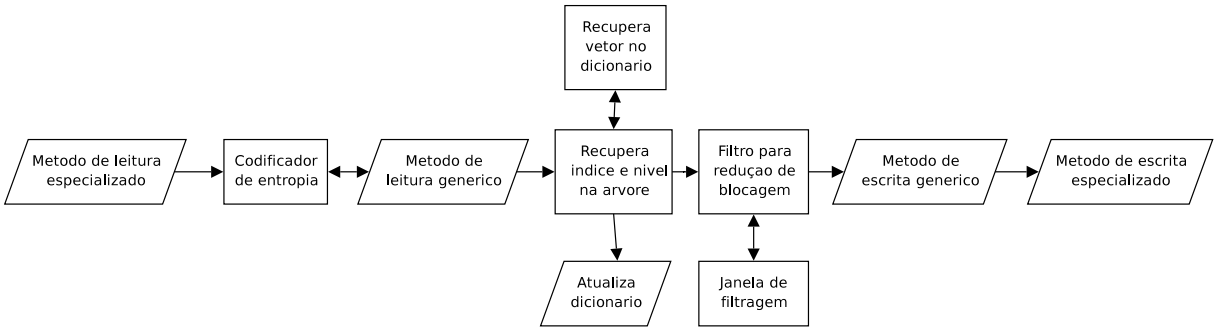


Figura 27: Esquemático de implementação do decodificador

Com essa abordagem modular, as alterações no código ficam extremamente segmentadas e simples de serem realizadas. Permitindo que novas experiências sejam realizadas facilmente, como a alteração do codificador de entropia ou do método de atualização do dicionário, por exemplo. Para atingir esse fim, as funções foram projetadas para permitir sua troca sem alteração na sua forma de chamada em outros módulos, como pode ser visto nos esquemáticos. Entre as facilidades proporcionadas, o codificador pode, por exemplo, ser adaptado para ler de arquivos de diferentes formatos, *streaming* de VoIP ou outras mídias apenas alterando os métodos especializados de leitura e escrita.

## 4.4 Dicionário

Grande parte do trabalho na especialização do algoritmo MMP para realizar codificação de voz foi no estudo dos parâmetros que definem o dicionário utilizado. Os testes efetuados durante a escolha destes parâmetros estão expostos no capítulo seguinte, nesta seção apenas estão descritos a estrutura e métodos utilizados na versão final deste trabalho bem como suas implicações.

### 4.4.1 Estrutura

A escolha do tamanho do bloco utilizado afeta diretamente duas características do codificador. A taxa de codificação que ele consegue alcançar e a complexidade da codificação. A taxa pode ser impactada de duas maneiras distintas e de efeitos contrários. Ela pode ser reduzida com o aumento dos blocos, pois é possível que o codificador encontre casamentos de grandes vetores, diminuindo o número de índices necessários na saída. Porém por outro lado, caso esses casamentos sejam muito raros, pela natureza muito dinâmica do sinal ou por que o tamanho escolhido foi muito elevado, o codificador terá

que adicionar no *stream* de saída diversos *flags* para pular níveis quase nunca utilizados impactando de maneira negativa na taxa.

Já a complexidade é atingida de uma maneira mais previsível. Quando se opta por um aumento do número de amostras em um bloco, a profundidade da árvore binária percorrida aumenta, e o número de recursividades da função de análise também. Assim impacta diretamente no aumento da complexidade necessária para realizar a codificação.

Devido à característica de formação e análise do algoritmo utilizando uma árvore binária, poucos tamanhos de blocos foram considerados viáveis para serem utilizados, pois apenas as potências de 2 poderiam ser particionados totalmente. Assim, com alguns testes e levando em consideração os pontos supra-citados, o tamanho de bloco utilizado no trabalho foi escolhido como 128 amostras, o que para áudio amostrado a 8 kHz corresponde a 16 ms de voz. Este número corresponde também ao período aproximado de estacionariedade da voz considerado em diversos estudos da área, sendo valores próximos a este amplamente utilizados em outros trabalhos similares.

A partir dessa informação uma estrutura com oito subdicionários está definida, bem como o tamanho dos vetores em cada subdicionário. Um ponto interessante, é que o algoritmo não limita o número máximo de vetores que um dado nível do dicionário pode conter. Em teoria, seria até melhor que ele crescesse indefinidamente, pois a probabilidade de um casamento bem-sucedido seria cada vez maior. Porém, para não tornar o processo de busca muito custoso, neste trabalho o número máximo de vetores que o dicionário pode conter é limitado e igual a 8192. Lembrando que, caso a codificação do índice do dicionário não utilizasse um modelo baseado na entropia, haveria uma justificativa em se reduzir o número de vetores para limitar a quantidade de bits necessários para representar o índice e não gerar uma taxa elevada.

Com a estrutura do dicionário definida, o algoritmo pode preênche-lo com sua população inicial. Aproveitou-se essa disposição inicial dos vetores para realizar, de maneira implícita ao algoritmo, uma redução da quantidade de bits por amostra e uma compressão prévia na taxa do sinal. Os sinais de teste utilizados foram amostrados a 16 bits, porém testes perceptuais mostram que, para a chamada voz telefônica, sinais com 12 bits por amostra são considerados de boa qualidade. Dessa maneira, o dicionário foi preenchido inicialmente com vetores de 0 a 65536 com um passo de quantização de 16, perfazendo um total de 4096 vetores iniciais.

O dicionário utilizado no codificador implementado está representado na figura 28. Ele é formado por um conjunto de 7 matrizes. Cada matriz representa um nível $N$, com

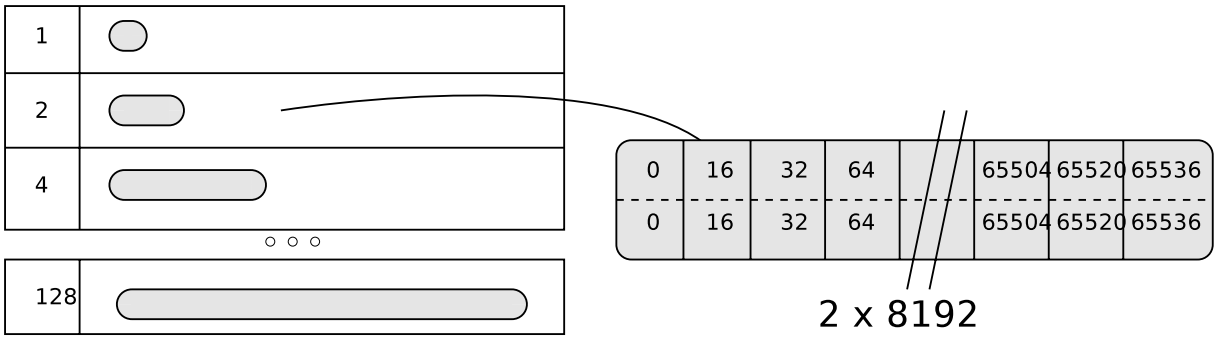


Figura 28: Representação do vetor de matrizes que compõe a implementação prática do dicionário.

$N$ variando de 0 a 7 e seu tamanho é de $2^N$ linhas por 8192 colunas. Cada coluna dessa matriz corresponde a um vetor armazenado neste nível do dicionário.

### 4.4.2 Atualização

Uma parte fundamental do MMP é sua capacidade de se adaptar as características do sinal codificado, ou seja, seus dicionários aprendem os padrões que recebem. Esse processo ocorre através da atualização do dicionário como explicado no capítulo anterior. Nesta implementação optou-se por dois tipos de incremento do dicionário: concatenação dos blocos e transformação de escala.

A concatenação dos blocos codificados acontece quando o algoritmo retorna a um determinado nó após a codificação de seus dois filhos. O algoritmo recebe o índice que foi utilizado em cada codificação, os recupera no dicionário de nível inferior, concatena os dois vetores e insere no dicionário do nível que se encontra. Esta operação está descrita na Figura 29.

A operação de trasnformação ocorre logo após a inserção e realiza uma interpolação ou uma decimação do vetor. Ela pode servir para contrair ou dilatar o vetor inserido no dicionário e replicar sua informação por todas os níveis do dicionário. O efeito dessa transformação na atualização do dicionário está representada na Figura 30 e nesta implementação utilizou-se interpolação linear para realizar as transformações de escala.

Como dito anteriormente, o número de vetores contido em cada nível do dicionário é limitado na implementação por motivos práticos. Isso gera um problema imediato quando se atualiza o dicionário com as concatenações e transformações. O codificador precisa incialmente manter um registro de que vetores estão utilizados no dicionário, já

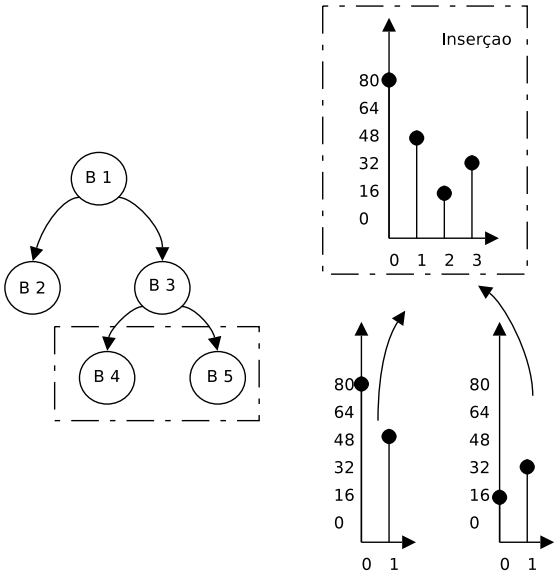


Figura 29: Concatenação dos nós codificados para atualização.

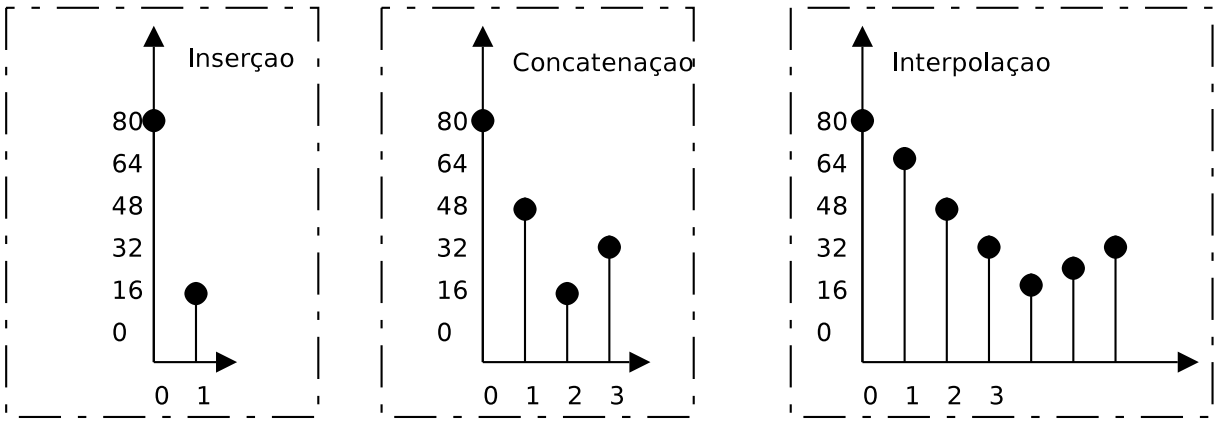


Figura 30: Transformação de escala das atualizações.

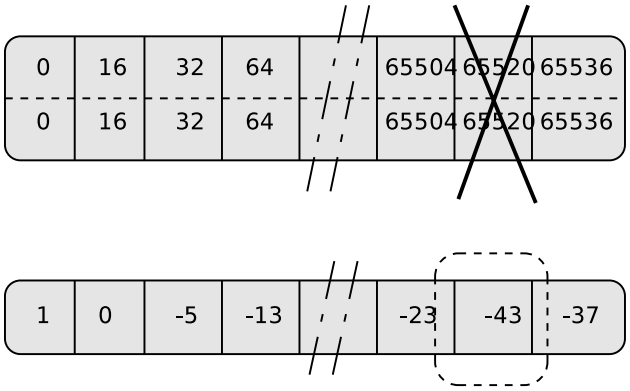


Figura 31: Estatística para substituição do dicionário.

que sua população original não ocupa todos os espaços disponíveis. Com essa informação o codificador pode escolher aonde inserir os novos vetores, porém após a codificação de alguns blocos, não haverá mais espaços disponíveis para inserções. Nesse momento o codificador precisa optar por discartar algum vetor que não é utilizado há muito tempo.

Essa técnica de atualização foi implementada através de um vetor auxiliar para cada nível do dicionário, contendo uma pontuação que representa a utilização de cada elemento. Essa vetor contendo a estatística de substituição do dicionário começa com todos os elementos valendo 1 ponto. A cada codificação as palavras não usadas perdem 1 ponto, pois elas estão envelhecendo no dicionário e podem não representar mais a estatística atual do sinal de entrada. Porém, quanda a palavra é usada novamente, sua pontuação retorna a 1. Assim, utilizando essa vetor auxiliar, o codificador escolhe o elemento com a menor pontuação e o substitui pela nova inserção como pode ser visto na figura 31, realizando a renovação do dicionário.

## 4.5 Codificador aritmético

A saída do codificador MMP, como representado na figura 26, ainda foi submetida a um codificador de entropia. Nesta implementação utilizou-se um codificador binário aritmético. Esta técnica foi escolhida devido a sua facilidade de adaptação e eficácia em aproximar a entropia da fonte. Essa estratégia possibilitou a redução da taxa de compressão sem inserir perdas adicionais no sinal, funcionando como um complemento ao algoritmo.

A utilização do codificador aritmética para representar os símbolos de saída do codificador permitiu também que o dicionário crescesse de tamanho sem afetar negativamente

a taxa. Pois nos esquemas de codificação que utilizam códigos de comprimento variável o que importa para a taxa final é a entropia do sinal, sendo pouco influenciada pelo número total de símbolos existentes.

O codificador aritmético, como explicado anteriormente, utiliza um modelo probabilístico para determinar o tamanho do segmento de cada símbolo. Este modelo, é um vetor contendo o número de ocorrências de cada símbolo no *stream* de saída. Neste trabalho, o codificador aritmético implementado utilizou modelos probabilísticos diferentes para representar a estatística de cada nível do dicionário. Pois os índices codificados em um determinado nível, apesar de serem representados pelo mesmo símbolo, referenciam vetores diferentes no dicionário, com probabilidades diferentes de aparecerem na saída. Utilizou-se também um modelo separado para representar os símbolos de divisão da árvore binária bem como um símbolo de fim de arquivo.

A implementação do codificador aritmético utilizado neste projeto foi baseado no trabalho desenvolvido em (WITTEN; NEAL; CLEARY, 1987). A codificação funciona na prática atribuindo uma faixa de inteiros ao intervalo entre 0 e 1 descrito no modelo teórico. É utilizado um vetor que armazena o número de vezes que determinado símbolo foi recebido e constitui o modelo de probabilidades do codificador. Para auxiliar a execução do algoritmo, outro vetor é definido, nele são armazenados os somatórios do primeiro vetor, variando de 0 até a posição $n$. Este procedimento de inicialização está representado no pseudo-código abaixo:

**Inicialização**

```
1: Recebe as probabilidades e preenche vetor Freq
2: FreqAcumulada[NumeroDeSimbolos] ← Freq[NumeroDeSimbolos]
3: for i=NumeroDeSimbolos - 1 to 0 do
4:    FreqAcumulada[i] = Freq[i] + FreqAcumulada[i+1]
5: end for
6: Superior ← FFFFh
7: Inferior ← 0000h
8: UnderflowBits ← 0
```

De posse dessas informações iniciais ele obtém dois números inteiros que correspondem a faixa inicial. No processo teórico, esse intervalo seria subdividido em faixas proporcionais a probabilidade de cada símbolo recebido e no final se obteria um único número real contendo todos os símbolos codificados. Porém isso geraria problemas imediatos com a precisão necessária para representá-lo em um computador real. Uma forma de contornar

esse fato, que foi empregada neste trabalho é, a medida que o bit mais significativo dos dois limites se torna igual, diz-se que o intervalo estabilizou. Neste momento este bit mais significativo pode ser enviado para a saída e o valor contido nos limites é deslocado para a esquerda. Dessa maneira um número infinitamente longo pode ser representado, pois apenas parte dele permanece armazenada nas variáveis de trabalho.

Este processo de codificação está representado no pseudo-código a seguir, onde considerou-se 16 bits para definição da faixa de trabalho:

**Codificação Aritmética**

1: Faixa = (Superior - Inferior) + 1
2: Superior = Inferior + (Faixa $\cdot \frac{\text{Freq[Simbolo]}}{\text{FreqAcumulada[Simbolo]}}$)-1
3: **loop**
4:  **if** Msb de Superior = Msb de Inferior **then**
5:  Envia msb de Inferior $\rightarrow$ saída
6:  **while** UnderflowBits $\geq 0$ **do**
7:  Envia (not(msb de Inferior) $\rightarrow$ saída)
8:  $Underflow Bits$ -=1
9:  **end while**
10:  **else if** Segundo msb de Superior = 1 e Segundo msd de Inferior = 0 **then**
11:  $Underflow Bits$ += 1
12:  Inferior = Inferior - 4000h
13:  Superior = Superior - 4000h
14:  **else**
15:  Break
16:  **end if**
17:  Desloca Inferior para a esquerda 1 bit
18:  Desloca Superior para a esquerda 1 bit
19:  Superior += 1
20: **end loop**

O processo prático de decodificação acontece de maneira análoga, e pode ser compreendido através dos passos a seguir, onde Valor é o número recuperado inicialmente do *stream* codificado.

**Decodificação Aritmética**

1: Faixa = (Superior - Inferior) + 1
2: Acumulado = $\frac{(\text{Valor}-\text{Inferior}+1)\cdot \text{FreqAcumulada}[0]-1)}{\text{Faixa}}$

3: Verifica que símbolo corresponde à variável acumulado varrendo o vetor FreqAcumulada
4: Superior = Inferior + Faixa $\cdot \frac{\text{FrequenciaAcumulada[Simbolo]}}{\text{FrequenciaAcumulada[0]}-1}$
5: Inferior = Inferior + Faixa $\cdot \frac{\text{FrequenciaAcumulada[Simbolo+1]}}{\text{FrequenciaAcumulada[0]}}$
6: **loop**
7: **if** Msb de Superior = Msb de Inferior **then**
8: Desloca Inferior para a esquerda 1 bit
9: Desloca Superior para a esquerda 1 bit
10: **else if** Segundo msb de Superior = 1 e Segundo msd de Inferior = 0 **then**
11: Valor = Valor - 4000h
12: Inferior = Inferior - 4000h
13: Superior = Superior - 4000h
14: Desloca Inferior para a esquerda 1 bit
15: Desloca Superior para a esquerda 1 bit
16: **else**
17: Break
18: Valor ← mais 1 bit do stream
19: **end if**
20: **end loop**

## 4.6 Filtro para redução de blocagem

O efeito de blocagem é um fenômeno comum a todos os codificadores em blocos como o MMP. Essa nomenclatura é muito comum no domínio do processamento de imagens, onde esse efeito aparece como bordas entre os blocos e é facilmente detectado pelo olho humano. Na voz, o efeito não é reconhecido de imediato e pode ser atribuído a uma quantização mais agressiva. Gera nas amostras codificadas ruído de alta freqüência causado por descontinuidades artificiais inseridas pelo codificador nas fronteiras dos blocos. Esses erros ocorrem porque o codificador analisa cada bloco de maneira independente e se acentua principalmente quando utilizado a taxas baixas, onde maiores afastamentos dos valores originais dos blocos são tolerados, favorecendo o aparecimento destas descontinuidades.

O procedimento de segmentação utilizado no MMP divide o vetor de entrada $X$ em $L$ segmentos: $X = ( X^{l_0} \ ... \ X^{l_{L-1}} )$. Cada segmento é independentemente aproximado usando vetores pertencentes a um dicionário de diversas escalas. Não existe em príncipio nenhuma restrição à escolha dos vetores de reprodução que leve em conta a continuidade

do ponto de segmentação. Uma forma de controlar este efeito de blocagem é alterar o cálculo do custo utilizado no procedimento de busca de modo a considerar explicitamente a descontinuidade no ponto de segmentação. Uma outra possibilidade é utilizar blocos com superposição. Nesta abordagem, a soma dos comprimentos dos $L$ segmentos é maior que o comprimento total do vetor de entrada $N$. Dois segmentos adjacentes não são apenas concatenados, mas somados com superposição de alguns componentes.

Contudo, quando se utiliza superposição no codificador, a seleção da melhor árvore de codificação do MMP seria dificultada. Neste caso, a distorção associada a um nó passaria a ser dependente dos vetores de reprodução associados aos segmentos adjacentes, o que seria complexo computacionalmente. Devido a este problema, a solução alternativa proposta em (CARVALHO, 2001) foi implementada neste trabalho. Uma técnica de pós-filtragem adaptativa foi implementada no codificador e adiciona a superposição dos vetores de reprodução apenas no decodificador.

Para evitar os problemas supra-citados, a técnica utilizada utiliza vetores sem sobreposição para realizar a análise do sinal no codificado e vetores com sobreposição para realizar a reconstrução do lado do decodificador. Exemplificando, considera-se um vetor de entrada $X$, codificado pelo MMP em quatro segmentos de mesmo comprimento representado por: $X = ( X^4 \ X^5 \ X^6 \ X^7 )$. Supondo ainda que a aproximação de cada segmento foi realizada por vetores contendo componentes de mesma amplitude temos: $X' = ( a_0 1_{\frac{n}{4}} \ a_1 1_{\frac{n}{4}} \ a_2 1_{\frac{n}{4}} \ a_3 1_{\frac{n}{4}} )$. O que é análogo a recuperar uma aproximação de $X$ a partir de uma versão decimada por 4, $X_d = ( a_0 \ a_1 \ a_2 \ a_3 )$. Pela teoria de bancos de filtros multi-taxas (DINIZ; SILVA; NETTO, 2004), pode-se recuperar uma versão passa-baixa de $X$ a partir de sua versão decimada utilizando-se um filtro de síntese diferente do filtro de análise.

Neste trabalho utilizou-se um filtro adaptativo FIR para modificar o formato dos vetores de síntese no decodificador, este filtro para redução de blocagem é aplicado após a reconstrução do bloco. É um filtro de média móvel com o comprimento ajustada a cada amostra de maneira a coincidir com o tamanho do vetor sendo processado, como exemplificado na figura 32. Nesta figura, o bloco de saída é composto de três segmentos. O componente sendo filtrado é indicado pela seta e, como pode ser visto, o comprimento do filtro FIR é proporcional ao tamanho original do vetor de reconstrução. A vantagem da modificação do tamanho do filtro de maneira a acompanhar o tamanho do vetor de síntese sendo processado é que a redução das freqüências altas do sinal é minimizada. Pois os filtros apenas atuam de maneira mais agressiva nos blocos onde a informação original

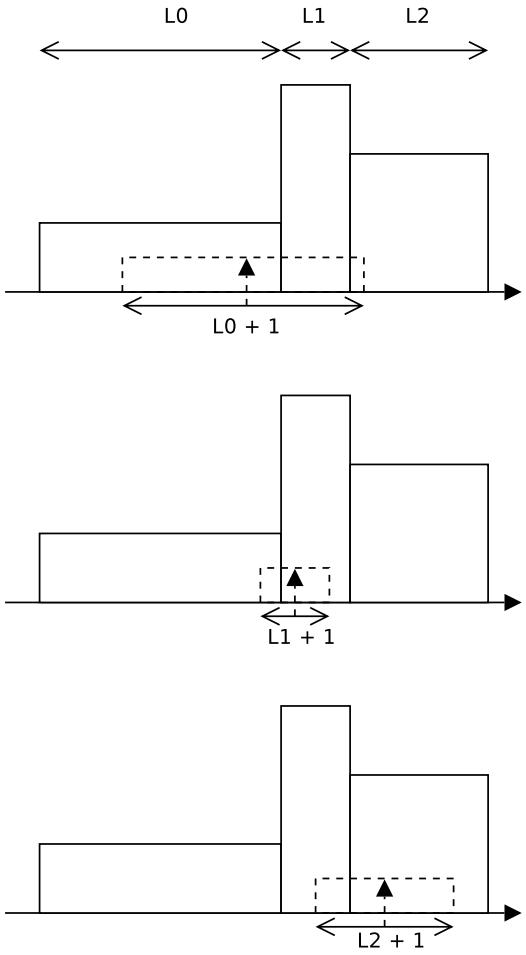


Figura 32: Variação de tamanho do filtro adaptativo para redução do efeito blocagem.

já havia sido descartada no codificador.

O tamanho máximo do filtro utilizado foi de 128 amostras. Porém realizar a suavização entre as bordas do bloco atual com o bloco futuro o filtro deve ser não-causal. Em outras palavras, caso o filtro seja de tamanho $N$, a resposta ao impulso do filtro deve ser não-nula de $-\frac{N}{2}$ à $\frac{N}{2}$. Assim, a maneira encontrada para torná-lo implementável, foi a adição de um atraso fixo de 64 amostras no sinal de saída do decodificador para considerar sempre o pior caso existente.

## 4.7 Conclusão

Este capítulo apresentou uma descrição da implementação prática do código do codificador MMP, incluindo os detalhes específicos envolvidos na adaptação para a aplicação em tela. Detalhou as escolhas realizadas na definição da estrutura do dicionário e des-

creveu as características e o algoritmo do codificador aritmético utilizado. Por último, cobriu também a técnica de pós-filtragem adaptativa utilizada como forma de amenizar os artefatos de blocagem presentes nos sinais decodificados.

# 5 Análises de resultados

## 5.1 Introdução

A implementação do algoritmo MMP discutida no capítulo anterior é a proposta final obtida durante este trabalho. Porém, durante seu curso, muitos parâmetros ou blocos foram acrescentados, influenciando na obtenção deste resultado final. Assim, nesta seção serão apresentados os resultados de testes objetivos que foram realizados com o sistema proposto e suas variantes, pois é imprescindível que a influência de cada uma dessas peças seja analisada para que a implementação das melhorias seja feita de maneira racional.

A seção 5.2 contém o detalhamento da metodologia empregada nos testes. Na seção 5.3 as alterações referentes ao dicionário serão discutidas, entre elas o tamanho dos blocos, seu tamanho máximo e métodos de atualização. A seção 5.4 analizará a eficácia do método de redução de blocagem através da pós-filtragem adaptativa e na seção 5.5 será analizado a influência do codificador aritmético na taxa final. A seção 5.6 estudará a habilidade do algoritmo de se adaptar a estatística dos sinais de voz, comparando os resultados quando há variação na duração das amostras. Por fim, na seção 5.7 há uma comparação entre os resultados do MMP com alguns algoritmos de codificação de voz atuais.

## 5.2 Metodologia de testes

Os testes foram realizados com os arquivo de amostra contidos na norma P.862 do ITU-T. São sessenta e seis arquivos de áudio de 8 segundos de duração cada, amostrados a 8 kHz com precisão de 16 bits. Os resultados de taxa estão expressos em bits por segundo e foram medidos através do tamanho total do arquivo comprimido. Lembrando que o algoritmo é de taxa variável e que este valor equivale à taxa média.

A qualidade do sinal reconstruído foi comparada com a da amostra original utilizando o programa de referência contido na norma P.862 do ITU-T que define o método PESQ.

Esta nota foi mapeada para uma escala MOS através da fórmula contida na norma P.862.1 do mesmo órgão.

Considerou-se como codificador de referência o MMP com bloco de 128 amostras, dicionário de 8192 vetores, pós-filtragem com atraso, codificador aritmético binário e atualização com interpolação dos vetores. Os testes foram realizados com a concatenação de 6 dos vetores de amostra disponíveis, sendo 3 com voz feminina e 3 com voz masculina, totalizando 48 segundos de amostra de áudio por teste. Para cada comparação, apenas a característica estudada foi alterada em relação ao codificador de referência.

## 5.3 Alterações nos dicionários

A escolha dos três parâmetros de definição do dicionário foram avaliados nesta seção através da comparação entre diferentes codificações. Os parâmetros estudados foram o tamanho do bloco, tamanho do dicionário e o método de atualização utilizado.

### 5.3.1 Tamanho do bloco

Neste teste foram utilizados blocos de 256 amostras com o intuito de verificar a utilização destes possíveis níveis pelo codificador. Para a obtenção destes dados, o codificador foi alterado de forma a exibir em sua saída qual nível do dicionário estava sendo utilizado para compor a saída. De posse desta informação, pode-se ver no histograma da Figura33 que os níveis mais utilizados são os intermediários.

Os níveis mais baixos do dicionário, são evitados pelo codificador durante a análise, pois quando efetua o cálculo de custo estas opções comprometem a taxa resultante. Já nos níveis mais altos, os casamentos são mais difíceis e por isso ocorrem com menos freqüência. Se o bloco do dicionário crescer indefinidamente, o codificador terá que sinalizar diversas subdivisões para passar por estes níveis não utilizados, impactando na taxa final. A partir dessa análise, optou-se por utilizar 7 níveis no dicionário de codificação, o que equivale a um bloco de análise de 128 amostras de voz.

### 5.3.2 Tamanho do dicionário

Para avaliar a influência do tamanho do dicionário na capacidade de compressão do algoritmo foram realizados testes com dicionários de 4096, 6144 e 8192 vetores por nível. Na Figura 34 apresentam-se os resultados obtidos para 6 níveis diferentes de compressão.

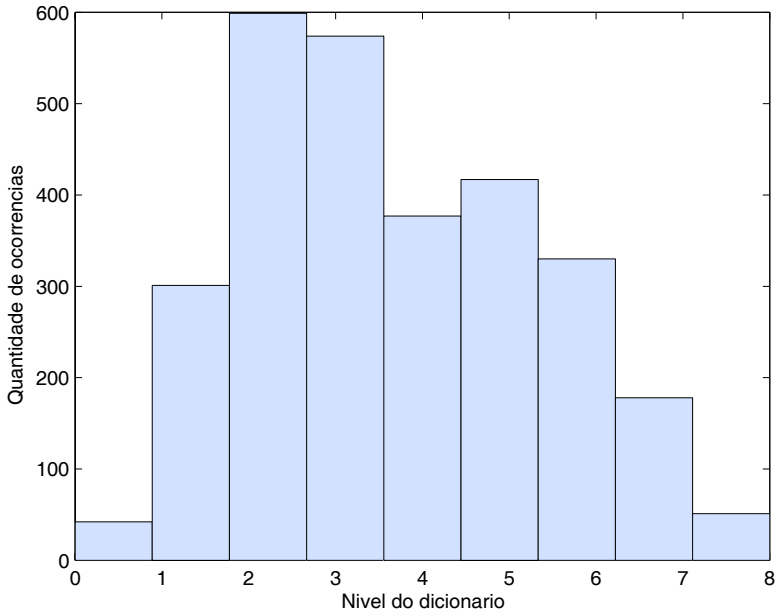


Figura 33: Histograma de utilização dos níveis do dicionário

Tabela 2: Avaliação do impacto da variação do número de vetores do dicionário.

| **4096 palavras** | | **6144 palavras** | | **8192 palavras** | |
|---|---|---|---|---|---|
| MOS | Taxa | MOS | Taxa | MOS | Taxa |
| 3,29 | 18978 | 3,29 | 17328 | 3,44 | 16503 |
| 3,10 | 13781 | 3,08 | 12288 | 3,18 | 11484 |
| 2,85 | 8263 | 2,83 | 7751 | 2,94 | 7312 |
| 2,67 | 4980 | 2,60 | 5549 | 2,75 | 5388 |
| 2,51 | 4843 | 2,49 | 4670 | 2,75 | 4323 |
| 2,42 | 4148 | 2,36 | 3652 | 5,52 | 3545 |

As retas conectando os pontos foram obtidas através de interpolação linear e servem apenas para ilustrar o comportamento esperado do codificador dentro desses intervalos.

Pode-se observar que com o crescimento do número de vetores disponível para a realização de casamentos o algoritmo se torna mais eficiente. Concluindo-se que o método de aprendizagem adiciona padrões úteis à base de dados, permitindo que se encontrem representações mais eficientes dos padrões futuros.

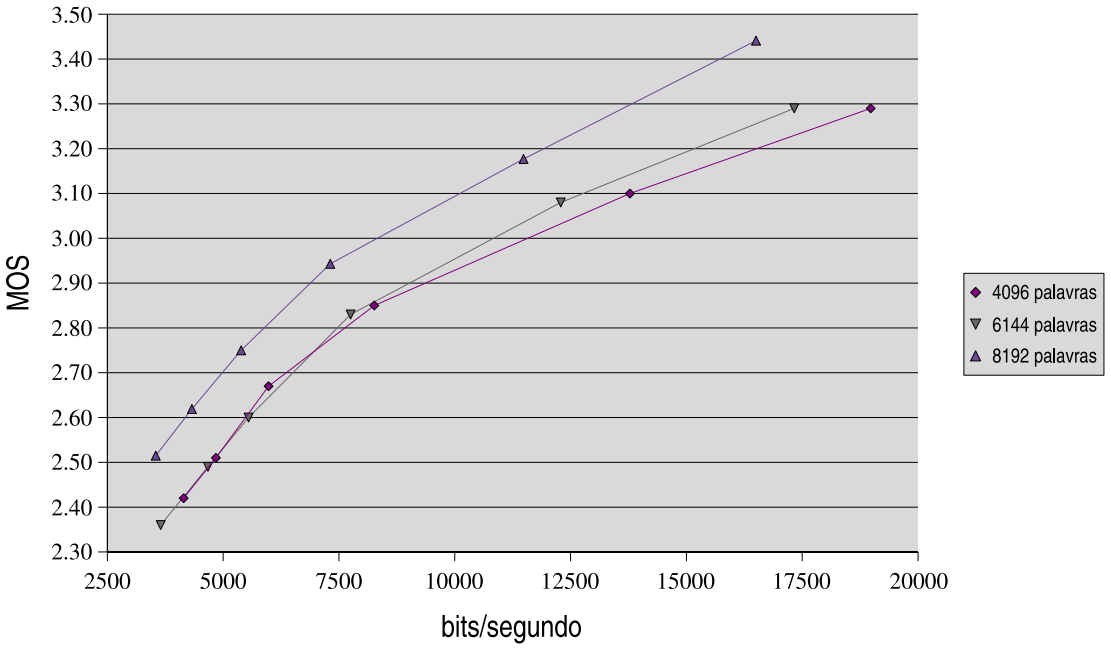


Figura 34: Avaliação do impacto da variação do número de vetores do dicionário.

### 5.3.3 Métodos de atualização

Nesta seção buscou-se avaliar os ganhos obtidos ao se inserir um novo método de atualização do dicionário. Neste trabalho incialmente considerou-se para a atualização do dicionário em um determinado nível apenas a concatenação dos vetores escolhidos no nível inferior. Porém posteriormente optou-se pela adição da interpolação e decimação desses vetores concatenados nos demais níveis do dicionário.

Esses vetores adicionais que passaram a ser inseridos aumentariam a diversidade do dicionário e possibilitariam o surgimento de novos níveis de quantização diferente dos originais, permitindo que o algoritmo se especialize na faixa dinâmica do sinal de entrada. Na Figura 35 tem-se a comparação entre essas duas propostas através da análise de taxa *versus* qualidade de reconstrução.

Conclui-se que o aumento da diversidade populacional do dicionário contribui para que o algoritmo se torne mais eficiente através de uma adaptação mais rápida aos padrões existentes no sinal. Observou-se através do gráfico um ganho tanto na taxa obtida, quanto na qualidade do sinal reconstruído.

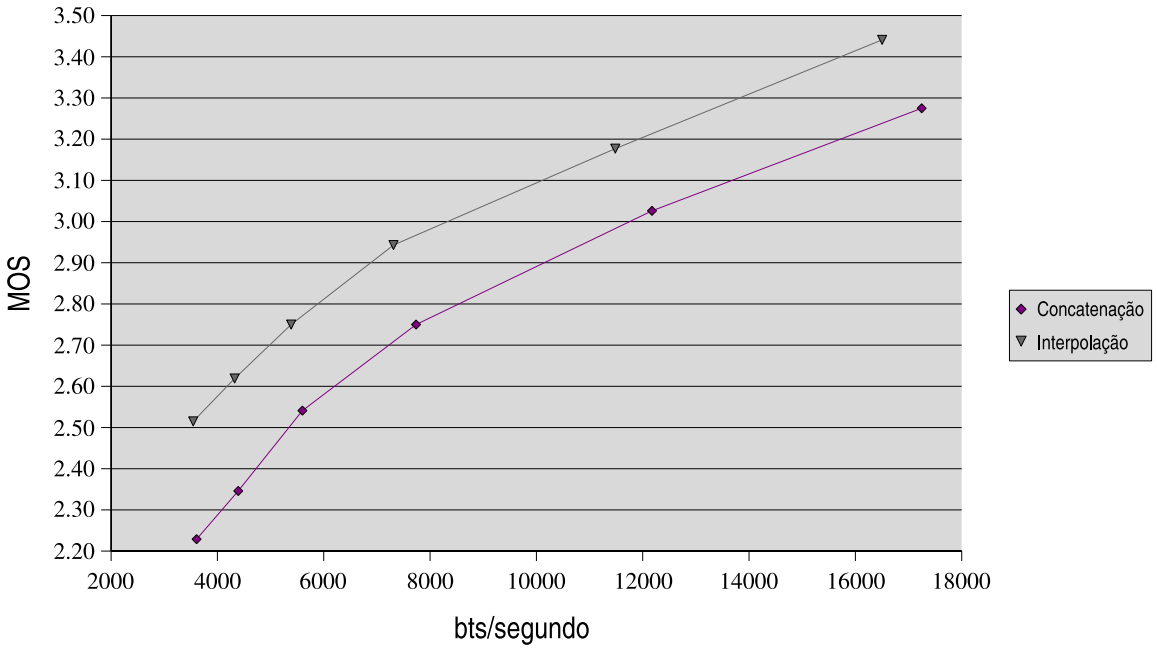


Figura 35: Avaliação dos métodos de atualização do dicionário

Tabela 3: Avaliação dos métodos de atualização do dicionário

| **Concatenação** | | **Interpolação** | |
|---|---|---|---|
| MOS | Taxa | MOS | Taxa |
| 3,28 | 17248 | 3,44 | 16503 |
| 3,03 | 12174 | 3,18 | 11484 |
| 2,75 | 7737 | 2,94 | 7312 |
| 2,54 | 5599 | 2,75 | 5388 |
| 2,35 | 4391 | 2,62 | 4323 |
| 2,23 | 3609 | 2,52 | 3545 |

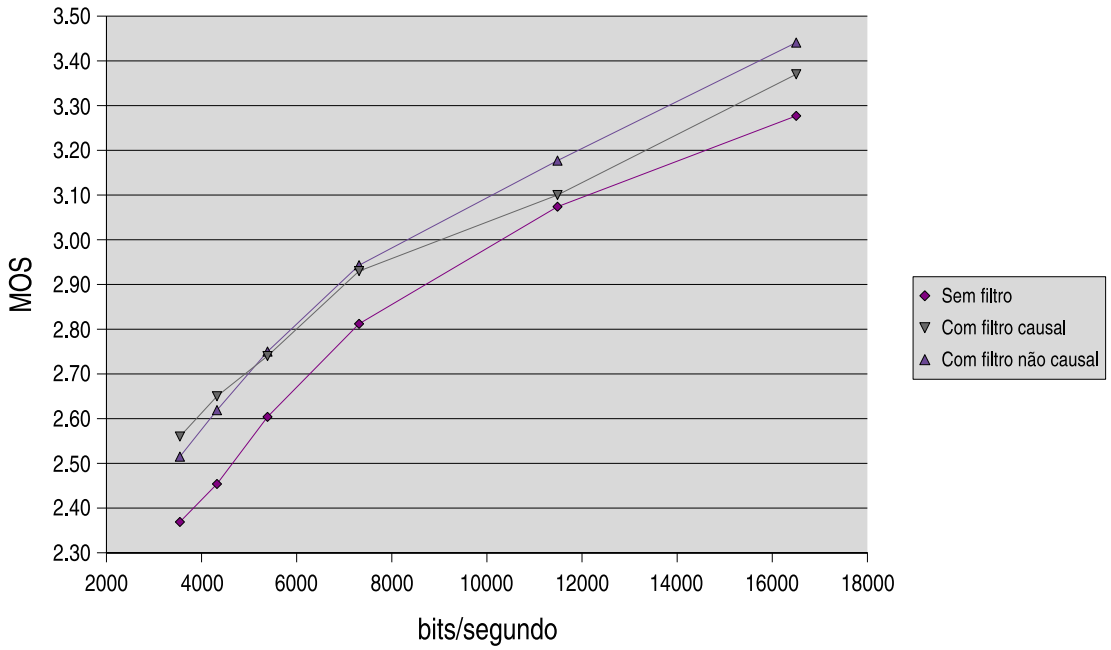


Figura 36: Avaliação do filtro para redução de blocagem

## 5.4 Filtro de redução de blocagem

O problema do surgimento de descontinuidades entre os blocos codificados, conforme explicado no capítulo anterior, é um problema inerente à estrutura empregada neste codificador. Como tentativa de se minimizar o aparecimento desses artefatos no sinal reconstruído empregou-se um método de pós-filtragem adaptativa, já descrito previamente.

Nesta seção tem-se a comparação entre a reconstrução utilizando a concatenção dos vetores recuperados do dicionário e a reconstrução utilizando a pós-filtragem desses vetores de acordo com o tamanho do bloco. Dois métodos de pós-filtragem foram considerados, o primeiro e que obteve melhores resultados, foi chamado neste trabalho de "não-causal". Este nome se justifica, pois ele considera na saída da amostra atual uma combinação de amostras futuras e passadas. Assim, para implementar este método adicionou-se ao codificador um atraso fixo de metade do tamanho do bloco. No segundo método apenas as amostras passadas foram consideradas e, apesar de se obter um resultado ligeiramente inferior, a redução do atraso total intrínseco ao algoritmo pode ser crucial para algumas aplicações.

Os resultados obtidos estão expostos na Figura 36.

Pode-se observar que para todas as taxas codificadas a adição da pós-filtragem pro-

Tabela 4: Avaliação do filtro para redução de blocagem

| **Sem filtro** | | **Com filtro** | | **Com filtro NC** | |
|---|---|---|---|---|---|
| MOS | Taxa | MOS | Taxa | MOS | Taxa |
| 3,28 | 16503 | 3,37 | 16503 | 3,44 | 16503 |
| 3,07 | 11484 | 3,10 | 11484 | 3,18 | 11484 |
| 2,81 | 7312 | 2,93 | 7312 | 2,94 | 7312 |
| 2,60 | 5388 | 2,74 | 5388 | 2,75 | 5388 |
| 2,45 | 4324 | 2,65 | 4324 | 2,62 | 4323 |
| 2,37 | 3545 | 2,56 | 3545 | 2,52 | 3545 |

porcionou uma melhoria na qualidade do sinal decodificado. Para baixas taxas não há diferenças perceptíveis entre as duas implementações do filtro. Porém no outro extremo da escala a implementação "não-causal"apresentou uma ligeira vantagem.

## 5.5 Codificador aritmético

A adição de um codificador de entropia como bloco de saída do algoritmo foi proposta considerando as possíveis vantagens obtidas com a redução na taxa sem influenciar a qualidade de reconstrução do sinal. Nesta seção analisou-se os ganhos obtidos através da adaptação do codificador aritmético ao modelo estatístico de utilização dos vetores de cada nível do dicionário. Proporcionando uma representação mais eficiente dos índices na saída.

No teste do codificador aritmético, foi utilizado o codificador MMP de referência e um codificador aritmético binário com um modelo estatístico para cada nível do dicionário, bem como um modelo para símbolos de divisão. Para fins de comparação, utilizou-se o codificador MMP configurado de maneira similar utilizando como bloco de saída um código de comprimento fixo para cada símbolo, utilizando apenas o número de bits necessários. Em outras palavras, 13 bits para os índices e 1 bit para os códigos de divisão.

Conforme representado na Figura 37, o codificador aritmético possibilitou, como esperado, uma redução de taxa sem alterar a qualidade de reconstrução do sinal. Essa redução alcançou o valor médio de 21% nos testes realizados.

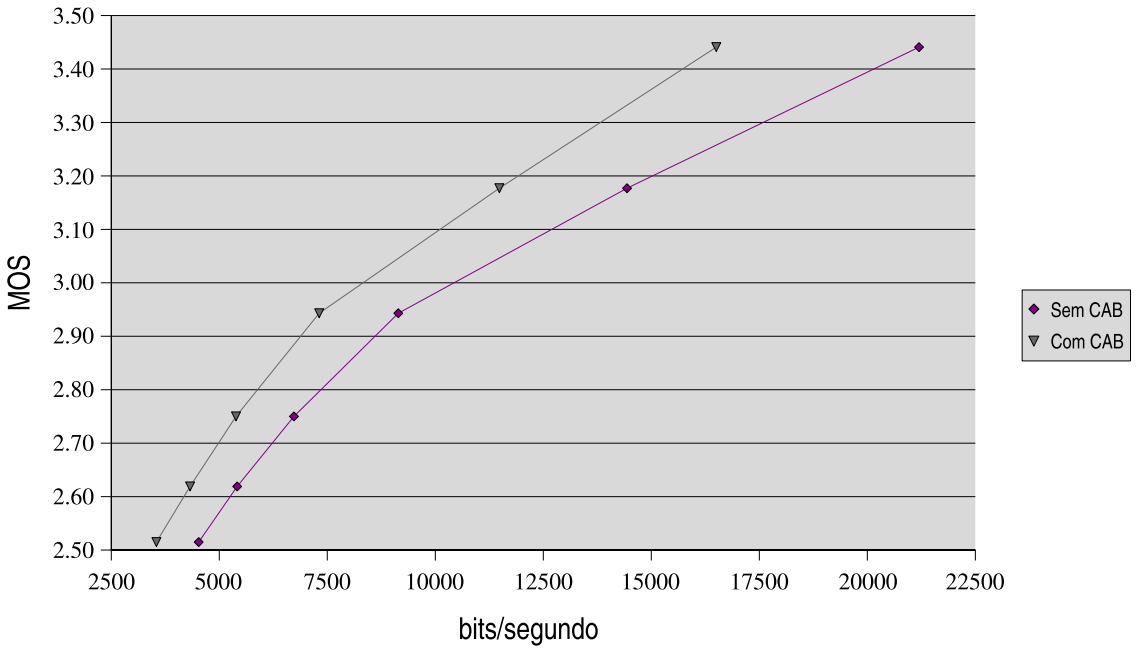


Figura 37: Avaliação do codificador aritmético

Tabela 5: Avaliação do codificador aritmético

| **Com CAB** | | **Sem CAB** | |
|---|---|---|---|
| MOS | Taxa | MOS | Taxa |
| 3,44 | 21297 | 3,44 | 16503 |
| 3,18 | 14439 | 3,18 | 11484 |
| 2,94 | 9144 | 2,94 | 7312 |
| 2,75 | 6728 | 2,75 | 5388 |
| 2,62 | 5414 | 2,62 | 4323 |
| 2,52 | 4525 | 2,52 | 3545 |

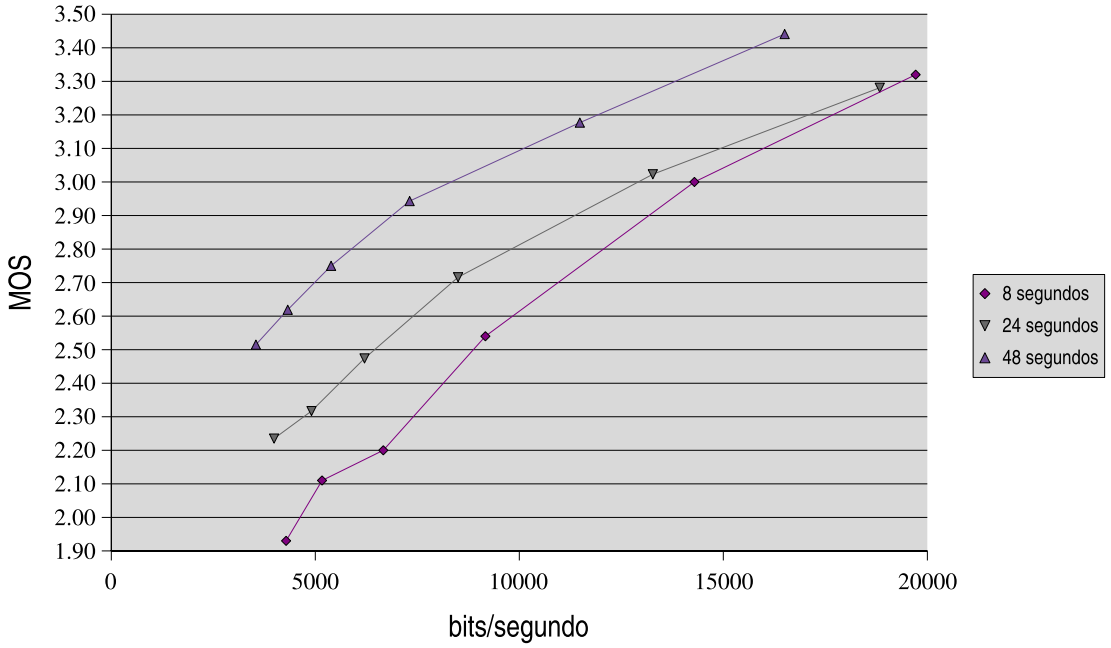


Figura 38: Avaliação da capacidade de adaptação do codificador

Tabela 6: Avaliação da capacidade de adaptação do codificador

| **8 segundos** | | **24 segundos** | | **48 segundos** | |
|---|---|---|---|---|---|
| MOS | Taxa | MOS | Taxa | MOS | Taxa |
| 3,32 | 19713 | 3,28 | 18835 | 3,44 | 16503 |
| 3,00 | 14291 | 3,02 | 13272 | 3,18 | 11484 |
| 2,54 | 9168 | 2,72 | 8500 | 2,94 | 7312 |
| 2,20 | 6667 | 2,47 | 6207 | 2,75 | 5388 |
| 2,11 | 5166 | 2,32 | 4906 | 2,62 | 4323 |
| 1,93 | 4284 | 2,24 | 3989 | 2,52 | 3545 |

## 5.6 Capacidade de adaptação

Um ponto importante para o algoritmo MMP é sua capacidade de aprender, durante o processo de codificação, a estatística do sinal de entrada. Esta habilidade se deve a atualização dos dicionários e dos modelos de codificação. Com o intuito de comprovar que estes procedimentos adaptativos são eficazes para obter uma taxa média menor, nesta seção foi realizada uma comparação entre as taxas obtidas com amostras de voz de comprimentos diferentes. Foram utilizadas amostras de 8, 24 e 48 segundos, codificadas de maneira independente, ou seja, com os dicionários e modelos sendo reinicializados.

Como pode-se observar na Figura 38, o algoritmo apresenta um desempenho notavel-

mente superior quando a duração das amostras codificadas é maior. Demonstrando assim, que sua capacidade adaptativa é um diferencial no processo de compressão. No entanto, para amostras ainda maiores, não se notou uma queda nas taxas obtidas. Verificou-se assim que o codificador atingiu um patamar na sua capacidade de adaptação para esse dado número de vetores disponíveis em seu dicionário e a ordem dos modelos de predição escolhidos para esses testes.

## 5.7 Comparação com outros codificadores

Nesta seção comparou-se o algoritmo MMP proposto neste trabalho com outros codificadores de voz já padronizados [(BELTRAO, 2003)(MAIA, 2000)]. Para esta finalidade foram escolhidos alguns padrões da indústria, entre eles:

- PCM (G.711)
- ADPCM (G.721)
- LPC-10 (FS-1015)
- DoD-CELP (FS-1016)
- LD-CELP (G.728)
- CS-ACELP (G.729)
- VSELP (IS54)

Os resultados podem ser vistos na Figura 39. Nela observa-se que os métodos de forma de onde, PCM e ADPCM oferecem, como esperado, uma excelente qualidade. Porém ao custo de uma alta relação de bits/segundo. No outro extremo, temos os vocoders, aqui representados pelo LPC-10. Essa família atende as finalidades onde a banda é extremamente restrita e a qualidade não é prioridade.

Como exposto anteriormente no meio desse caminho e apresentando a melhor relação custo-benefício atual, estão os codificadores CELP e suas variações. Mostrando um bom compromisso entre taxa e qualidade de reconstrução da voz.

Pode-se observar no gráfico algumas características diferenciais do MMP. Entre elas destaca-se a facilidade em se alterar, de maneira contínua, o ponto onde se deseja trabalhar na curva de taxa *versus* distorção. Esta variação não é simples nos demais algoritmos e

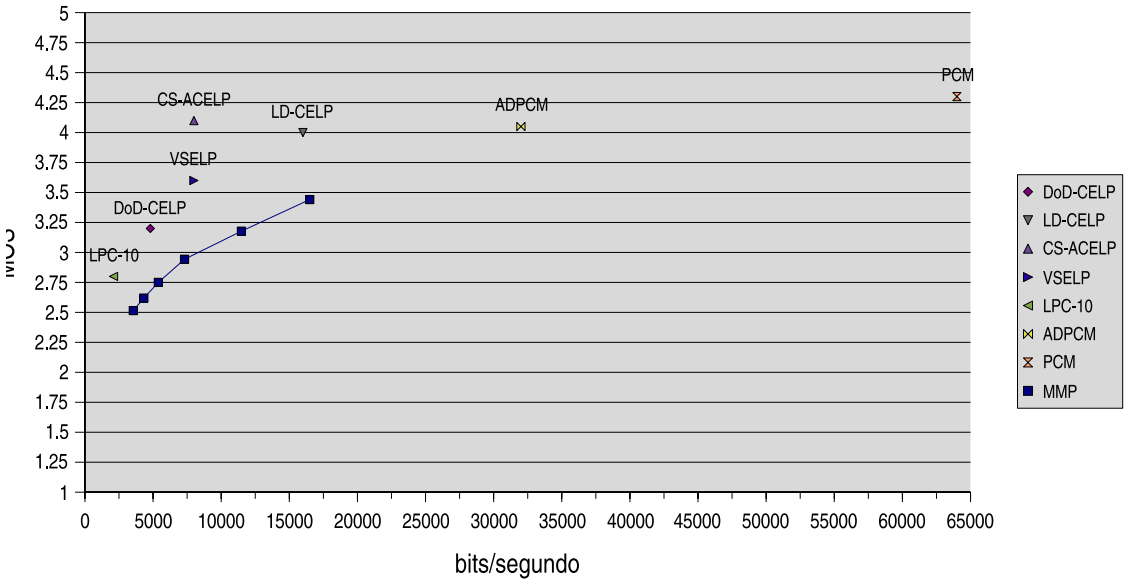


Figura 39: Comparativo entre padrões de codificação de voz

pode permitir o desenvolvimento de uma nova família de codificadores inteligentes que realizem um ajuste extremamente preciso em seus parâmetros para se adequar a rede onde estejam operando. Outro ponto interessante é sua capacidade de gerar um *stream* de dados com taxa variável. Em outras palavras, a taxa gerada só atinge o nível necessário para representar a complexidade que determinado trecho do sinal de entrada demanda.

Nota-se também que o algoritmo MMP apresentou um resultado ligeiramente inferior aos padrões da indústria. Porém deve-se levar em consideração o estágio inicial de pesquisa em que se encontra este codificador e o espaço existente para futuras adaptações. Assim, estes gráficos comparativos, servem primordialmente para situar o trabalho frente ao cenário atual e demonstrar a viabilidade no estudo dessa família de algoritmos em trabalhos aplicados à codificação de voz.

# 6 Conclusões

A proposta deste trabalho foi a de realizar um estudo da viabilidade em se aplicar um algoritmo desenvolvido originalmente no campo do processamento de imagens chamado de *Multiscale Multidimensional Parser* ao problema de codificação de voz. Além de implementar o algoritmo original detalhado em (CARVALHO, 2001), este trabalho consistia em propor adaptações e alterações para permitir que com uma maior especialização nas características da voz humana o sistema obtivesse resultados melhores em sua tarefa de compressão.

Seus objetivos foram alcançados, resultando na implementação prática em linguagem C de uma biblioteca que realiza a codificação e a decodificação de um arquivo contendo um sinal de voz PCM amostrado a 8 kHz e quantizado com 16 bits. O programa foi desenvolvido utilizando conceitos de orientação a objeto e projetado de forma a ser multiplataforma, podendo atender a diversas arquiteturas e finalidades.

Como exposto, o projeto envolveu a implementação de um sistema completo de codificação a partir do zero, permitindo o estudo e entendimento de cada etapa deste processo. Os parâmetros de ajuste do algoritmo, entre eles, a estrutura do dicionário, tamanho de bloco e técnicas de atualização, bem como as adições propostas foram justificados através de simulações criteriosas utilizando como métrica padrões de referência da área. Em particular foram utilizadas as recomendações do ITU P.862 e P.862.1, que são normas de medidas pseudo-subjetivas de qualidade.

Os resultados apresentados no capítulo de testes demonstraram a eficácia do algoritmo em realizar compressão com perdas de sinais de voz. Porém, em uma comparação frente aos padrões de codificação existentes na indústria o codificador demonstrou uma taxa ligeiramente maior para uma mesma qualidade de voz. Ainda assim, o sistema permite o processamento de voz com qualidade razoável à taxas em torno de 8 kilobits/segundo, rivalizando com os codificadores comerciais atuais.

Cabe notar que o algoritmo apresenta alguns pontos diferenciais, entre eles, a geração

de *streams* com taxa variável e a trivialidade em se alterar o ponto desejado na curva taxa-distorção de maneira contínua. Estes são itens atualmente desejados em sistemas de transmissão de voz via redes de computadores, pois possibilitam que um sistema de gestão possa gerenciar facilmente o impacto do tráfego gerado. Diferente do que ocorre na maioria sistemas de voz sobre IP que exigem artifícios como a comutação entre diversos codificadores com configurações diferentes funcionando em paralelo.

## 6.1 Propostas para trabalhos futuros

Alguns itens, por estarem fora de seu escopo original, não puderam ser abordados neste trabalho. Entre os pontos do codificador passíveis de melhorias ou merecedores de estudo destacam-se :

- O codificador apresenta uma complexidade elevada, como forma de reduzir esse problema, pode-se propor uma forma estruturada de dicionário. Essa estrutura permitiria uma busca semi-ótima porém a um custo reduzido;
- Propor um método para realizar os casamentos que leve em consideração parâmetros psicoacústicos;
- Propor um método que utilize técnicas de predição para atualizar o dicionário, aumentando a probabilidade de casamentos futuros;
- Estudar a viabilidade de codificar voz em banda larga;
- Estudar a viabilidade de se realizar a codificação em subbandas, atribuindo-se diferentes taxas a cada uma;
- Implementar cabeçalhos e protocolo de arquivo para possibilitar a detecção de erros no *stream*;
- Propor um método para realizar a análise dos blocos considerando superposição com seus vizinhos para reduzir efeito de blocagem;
- Propor um dicionário pré-inicializado para acelerar o processo de adaptação.

# Referências

ATAL, B. S.; REMDE, J. R. A new model of lpc excitation for producing natural-sounding speech at low bit rates. *Proc. Int. Conf. on Acous., Speech, and Sig. Processing*, v. 1, Maio 1982.

BELTRAO, F. C. da C. *Implementação de um codificador de voz CELP em tempo real.* Escola Politécnica/UFRJ, Maio 2003.

BENNETT, W. R. Secret telephony as a historical example of spread-spectrum communications. *IEEE Transactions on Communications*, COM-31, n. 1, Janeiro 1983.

CARVALHO, M. B. de. *Multidimensional Multiscale Parser - MMP.* Tese (Doutorado) — Programa de Engenharia Elétrica, COPPE/UFRJ, Abril 2001.

CARVALHO, M. B. de; SILVA, E. A. B. da; FINAMORE, W. A. Rate distortion optimized adaptive multiscale vector quantization. *ICIP*, II, n. 1, p. 439–442, Outubro 2001.

DELLER, J. R.; PROAKIS, J. G.; HANSEN, J. H. *Discrete-time processing of speech signals.* 2nd.. ed. New York: Macmillan, 1993.

DINIZ, P. S. R.; SILVA, E. A. B. da; NETTO, S. L. *Processamento Digital de Sinais.* 1a. ed. Porto Alegre: Bookman, 2004.

DUARTE, M. H. *Codificação de Imagens Estéreo Usando Recorrência de Padrões.* Dissertação (Mestrado) — Programa de Engenharia Elétrica, COPPE/UFRJ, Agosto 2002.

HAYKIN, S. *Communication Systems.* 4nd.. ed. New York: John Wiley Inc., 2001.

MAIA, R. da S. *Codificação CELP e análise espectral de voz.* Dissertação (Mestrado) — Programa de Engenharia Elétrica, COPPE/UFRJ, Março 2000.

WITTEN, I. H.; NEAL, R. M.; CLEARY, J. G. Arithmetic coding for data compression. *ACM*, v. 30, n. 6, Junho 1987.

# Anexo

# Anexo A - Códigos

## libac.h

```
#ifndef AC_HEADER
#define AC_HEADER

#include <stdio.h>

typedef struct {
  FILE *fp;
  long low;
  long high;
  long fbits;
  int buffer;
  int bits_to_go;
  long total_bits;
} ac_encoder;

typedef struct {
  FILE *fp;
  long value;
  long low;
  long high;
  int buffer;
  int bits_to_go;
  int garbage_bits;
} ac_decoder;

typedef struct {
  int nsym;
  int *freq;
  int *cfreq;
  int adapt;
} ac_model;

void ac_encoder_init (ac_encoder *, const char *);
void ac_encoder_done (ac_encoder *);
void ac_decoder_init (ac_decoder *, const char *);
void ac_decoder_done (ac_decoder *);
void ac_model_init (ac_model *, int, int *, int);
void ac_model_done (ac_model *);
long ac_encoder_bits (ac_encoder *);
void ac_encode_symbol (ac_encoder *, ac_model *, int);
```

```
41  int ac_decode_symbol (ac_decoder *, ac_model *);
42
43  #endif
```

## libac.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #include "libac.h"
5
6   #define Code_value_bits 16
7
8   #define Top_value (((long)1<<Code_value_bits)-1)
9   #define First_qtr (Top_value/4+1)
10  #define Half          (2*First_qtr)
11  #define Third_qtr (3*First_qtr)
12  #define Max_frequency 25000//16383
13
14  static void output_bit (ac_encoder *, int);
15  static void bit_plus_follow (ac_encoder *, int);
16  static int input_bit (ac_decoder *);
17  static void update_model (ac_model *, int);
18
19  #define error(m)                                              \
20  do  {                                                         \
21    fflush (stdout);                                            \
22    fprintf (stderr, "%s:%d: error: ", __FILE__, __LINE__);     \
23    fprintf (stderr, m);                                        \
24    fprintf (stderr, "\n");                                     \
25    exit (1);                                                   \
26  }  while (0)
27
28  #define check(b,m)                                            \
29  do  {                                                         \
30    if (b)                                                      \
31      error (m);                                                \
32  }  while (0)
33
34  static void
35  output_bit (ac_encoder *ace, int bit)
36  {
37    ace->buffer >>= 1;
38    if (bit)
39      ace->buffer |= 0x80;
40    ace->bits_to_go -= 1;
41    ace->total_bits += 1;
42    if (ace->bits_to_go==0)  {
43      if (ace->fp)
44        putc (ace->buffer, ace->fp);
45      ace->bits_to_go = 8;
46    }
47
```

```
48    return;
49  }
50
51  static void
52  bit_plus_follow (ac_encoder *ace, int bit)
53  {
54    output_bit (ace, bit);
55    while (ace->fbits > 0)  {
56      output_bit (ace, !bit);
57      ace->fbits -= 1;
58    }
59
60    return;
61  }
62
63  static int
64  input_bit (ac_decoder *acd)
65  {
66    int t;
67
68    if (acd->bits_to_go==0)  {
69      acd->buffer = getc(acd->fp);
70      if (acd->buffer==EOF)  {
71        acd->garbage_bits += 1;
72        if (acd->garbage_bits>Code_value_bits-2)
73          error ("arithmetic decoder bad input file");
74      }
75      acd->bits_to_go = 8;
76    }
77
78    t = acd->buffer&1;
79    acd->buffer >>= 1;
80    acd->bits_to_go -= 1;
81
82    return t;
83  }
84
85  static void
86  update_model (ac_model *acm, int sym)
87  {
88    int i;
89
90    if (acm->cfreq[0]==Max_frequency)  {
91      int cum = 0;
92      acm->cfreq[acm->nsym] = 0;
93      for (i = acm->nsym-1; i>=0; i--)  {
94        acm->freq[i] = (acm->freq[i] + 1) / 2;
95        cum += acm->freq[i];
96        acm->cfreq[i] = cum;
97      }
98    }
99
100   acm->freq[sym] += 1;
101   for (i=sym; i>=0; i--)
```

```
102       acm->cfreq[i] += 1;
103
104     return;
105   }
106
107   void
108   ac_encoder_init (ac_encoder *ace, const char *fn)
109   {
110
111     if (fn)  {
112       ace->fp = fopen (fn, "wb"); /* open in binary mode */
113       check (!ace->fp, "arithmetic encoder could not open file");
114     }  else  {
115       ace->fp = NULL;
116     }
117
118     ace->bits_to_go = 8;
119
120     ace->low = 0;
121     ace->high = Top_value;
122     ace->fbits = 0;
123     ace->buffer = 0;
124
125     ace->total_bits = 0;
126
127     return;
128   }
129
130   void
131   ac_encoder_done (ac_encoder *ace)
132   {
133     ace->fbits += 1;
134     if (ace->low < First_qtr)
135       bit_plus_follow (ace, 0);
136     else
137       bit_plus_follow (ace, 1);
138     if (ace->fp)
139       putc (ace->buffer >> ace->bits_to_go, ace->fp);
140
141     if (ace->fp)
142       fclose (ace->fp);
143
144     return;
145   }
146
147   void
148   ac_decoder_init (ac_decoder *acd, const char *fn)
149   {
150     int i;
151
152     acd->fp = fopen (fn, "rb"); /* open in binary mode */
153     check (!acd->fp, "arithmetic decoder could not open file");
154
155     acd->bits_to_go = 0;
```

```
156   acd->garbage_bits = 0;
157
158   acd->value = 0;
159   for (i=1; i<=Code_value_bits; i++)  {
160     acd->value = 2*acd->value + input_bit(acd);
161   }
162   acd->low = 0;
163   acd->high = Top_value;
164
165   return;
166 }
167
168 void
169 ac_decoder_done (ac_decoder *acd)
170 {
171   fclose (acd->fp);
172
173   return;
174 }
175
176 void
177 ac_model_init (ac_model *acm, int nsym, int *ifreq, int adapt)
178 {
179   int i;
180
181   acm->nsym = nsym;
182   acm->freq = (int *) (void *) calloc (nsym, sizeof (int));
183   check (!acm->freq, "arithmetic coder model allocation failure");
184   acm->cfreq = (int *) (void *) calloc (nsym+1, sizeof (int));
185   check (!acm->cfreq, "arithmetic coder model allocation failure");
186   acm->adapt = adapt;
187
188   if (ifreq)  {
189     acm->cfreq[acm->nsym] = 0;
190     for (i=acm->nsym-1; i>=0; i--)  {
191       acm->freq[i] = ifreq[i];
192       acm->cfreq[i] = acm->cfreq[i+1] + acm->freq[i];
193     }
194     if (acm->cfreq[0] > Max_frequency)
195       error ("arithmetic coder model max frequency exceeded");
196   }  else  {
197     for (i=0; i<acm->nsym; i++) {
198       acm->freq[i] = 1;
199       acm->cfreq[i] = acm->nsym - i;
200     }
201     acm->cfreq[acm->nsym] = 0;
202   }
203
204   return;
205 }
206
207 void
208 ac_model_done (ac_model *acm)
209 {
```

```
210   acm->nsym = 0;
211   free (acm->freq);
212   acm->freq = NULL;
213   free (acm->cfreq);
214   acm->cfreq = NULL;
215
216   return;
217 }
218
219 long
220 ac_encoder_bits (ac_encoder *ace)
221 {
222   return ace->total_bits;
223 }
224
225 void
226 ac_encode_symbol (ac_encoder *ace, ac_model *acm, int sym)
227 {
228   long range;
229
230   check (sym<0||sym>=acm->nsym, "symbol out of range");
231
232   range = (long)(ace->high-ace->low)+1;
233   ace->high = ace->low + (range*acm->cfreq[sym])/acm->cfreq[0]-1;
234   ace->low = ace->low + (range*acm->cfreq[sym+1])/acm->cfreq[0];
235
236   for (;;)  {
237     if (ace->high<Half)  {
238       bit_plus_follow (ace, 0);
239     }  else if (ace->low>=Half)  {
240       bit_plus_follow (ace, 1);
241       ace->low -= Half;
242       ace->high -= Half;
243     }  else if (ace->low>=First_qtr && ace->high<Third_qtr)  {
244       ace->fbits += 1;
245       ace->low -= First_qtr;
246       ace->high -= First_qtr;
247     }  else
248       break;
249     ace->low = 2*ace->low;
250     ace->high = 2*ace->high+1;
251   }
252
253   if (acm->adapt)
254     update_model (acm, sym);
255
256   return;
257 }
258
259 int
260 ac_decode_symbol (ac_decoder *acd, ac_model *acm)
261 {
262   long range;
263   int cum;
```

```
264   int sym;
265
266   range = (long)(acd->high-acd->low)+1;
267   cum = (((long)(acd->value-acd->low)+1)*acm->cfreq[0]-1)/range;
268
269   for (sym = 0; acm->cfreq[sym+1]>cum; sym++)
270     /* do nothing */ ;
271
272   check (sym<0||sym>=acm->nsym, "symbol out of range");
273
274   acd->high = acd->low + (range*acm->cfreq[sym])/acm->cfreq[0]-1;
275   acd->low = acd->low +  (range*acm->cfreq[sym+1])/acm->cfreq[0];
276
277   for (;;)  {
278     if (acd->high<Half)  {
279       /* do nothing */
280     }  else if (acd->low>=Half)  {
281       acd->value -= Half;
282       acd->low -= Half;
283       acd->high -= Half;
284     }  else if (acd->low>=First_qtr && acd->high<Third_qtr)  {
285       acd->value -= First_qtr;
286       acd->low -= First_qtr;
287       acd->high -= First_qtr;
288     }  else
289       break;
290     acd->low = 2*acd->low;
291     acd->high = 2*acd->high+1;
292     acd->value = 2*acd->value + input_bit(acd);
293   }
294
295   if (acm->adapt)
296     update_model (acm, sym);
297
298   return sym;
299 }
```

# libmmp.h

```
1  #ifndef _MMP_H_
2  #define _MMP_H_
3
4  #include <math.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <gsl/gsl_matrix.h>
8  #include <gsl/gsl_blas.h>
9  #include <stdint.h>
10 #include "libac.h"
11
12 #define BLOCKSIZE 128
13 #define NUMLEVEL 8
14 #define CBSIZE 4096
```

```
15
16  struct _MMP_CTX
17  {
18          FILE *instream;
19          FILE *outstream;
20          gsl_matrix *codebook[NUMLEVEL];
21          int cb_limit[NUMLEVEL];
22          gsl_vector *temp[NUMLEVEL];
23          gsl_vector *filter_levels;
24          gsl_vector *filter_memory;
25          gsl_matrix *codebook_stats;
26          int minrange;
27          int maxrange;
28          int step;
29          double lambda;
30          int code;
31          int bit_size;
32          uint32_t bit_buffer;
33          ac_encoder ac_enc;
34          ac_decoder ac_dec;
35          ac_model ac_codebook_model[NUMLEVEL];
36          ac_model ac_division_model;
37  };
38
39  typedef struct _MMP_CTX MMP_CTX;
40
41  int mmp_create_codebook (MMP_CTX *mmp);
42  MMP_CTX *mmp_encoder_init (double lambda, char *infile, char *outfile);
43  MMP_CTX *mmp_decoder_init (char *infile, char *outfile);
44  int mmp_vq (MMP_CTX *mmp, gsl_vector *block, int depth, int *code,
45                  double *distance);
46  int mmp_recursive_encode (MMP_CTX *mmp, gsl_vector *block, int *final_node,
47                  int *code, double *distance);
48  int mmp_write_final_node (MMP_CTX *mmp);
49  int mmp_write_code (MMP_CTX *mmp, int code, int depth);
50  int mmp_write_division (MMP_CTX *mmp);
51  int mmp_write_eof (MMP_CTX *mmp);
52  int mmp_update_codebook (MMP_CTX *mmp, int left_code, int right_code,
53                  int depth, int *new_code);
54  int mmp_partition (gsl_vector *block, gsl_vector *left_block,
55                  gsl_vector *right_block);
56  int mmp_update_stats (MMP_CTX *mmp, int code, int depth);
57  int mmp_decoder_destroy (MMP_CTX *mmp);
58  int mmp_encoder_destroy (MMP_CTX *mmp);
59  int mmp_encode (MMP_CTX *mmp);
60  int mmp_decode (MMP_CTX *mmp);
61  int mmp_get_raw_data (MMP_CTX *mmp, double *data);
62
63  #endif
```

# libbase.c

```
1  #include "libmmp.h"
2
3  int mmp_create_codebook(MMP_CTX *mmp)
4  {
5          int i, j, l, index, depth;
6
7          mmp->codebook_stats = gsl_matrix_calloc (NUMLEVEL, CBSIZE);
8          for (i=0 ; i<NUMLEVEL ; i++)
9          {
10                 depth = pow(2,i);
11                 mmp->codebook[i] = gsl_matrix_calloc (depth, CBSIZE);
12                 mmp->temp[i] = gsl_vector_calloc(depth);
13                 index = 0;
14                 for (j = mmp->minrange ; j <= mmp->maxrange ; j = j+mmp->step)
15                 {
16                         for (l = 0; l < depth; l++)
17                         {
18                                 gsl_matrix_set(mmp->codebook[i], l, index, j);
19                         }
20                         gsl_matrix_set(mmp->codebook_stats, i, index, 1);
21                         index++;
22                 }
23         }
24         return 0;
25 }
26
27 int mmp_update_stats (MMP_CTX *mmp, int code, int depth)
28 {
29         gsl_vector_view temp;
30
31         gsl_matrix_set (mmp->codebook_stats, depth, code, 2);
32         temp = gsl_matrix_row (mmp->codebook_stats, depth);
33         gsl_vector_add_constant(&temp.vector, -1);
34
35         if (gsl_vector_min (&temp.vector) < 16383)
36                 gsl_vector_scale (&temp.vector, 0.5);
37
38         return 0;
39 };
40
41 int mmp_get_cb_index (MMP_CTX *mmp, int depth)
42 {
43         gsl_vector_view col;
44         int min_code;
45
46         col = gsl_matrix_row (mmp->codebook_stats, depth);
47         min_code = gsl_vector_min_index (&col.vector);
48
49         if (min_code > mmp->cb_limit[depth])
50                 mmp->cb_limit[depth] = min_code;
51
52         return min_code;
```

```
53  }
54
55  void mmp_insert_expansion (MMP_CTX *mmp, int depth, int source, int dest)
56  {
57          double buf1, buf2;
58          int lim, i;
59
60          lim = pow(2,depth);
61          for (i = 0; i < lim-1; i++)
62          {
63                  buf1 = gsl_matrix_get (mmp->codebook[depth], i, source);
64                  buf2 = gsl_matrix_get (mmp->codebook[depth], i+1, source);
65                  gsl_matrix_set (mmp->codebook[depth + 1], 2*i, dest, buf1);
66                  gsl_matrix_set (mmp->codebook[depth + 1], 2*i+1, dest, fabs((buf1+buf2)/2));
67          }
68          buf1 = gsl_matrix_get (mmp->codebook[depth], lim-1, source);
69          gsl_matrix_set (mmp->codebook[depth + 1], 2*(lim-1), dest, buf1);
70          gsl_matrix_set (mmp->codebook[depth + 1], 2*(lim-1)+1, dest, buf1);
71  }
72
73  void mmp_insert_contraction (MMP_CTX *mmp, int depth, int source, int dest)
74  {
75          double buf1, buf2;
76          int lim,i;
77
78          lim = pow(2,depth-1);
79          for (i = 0; i < (lim-1); i++)
80          {
81                  buf1 = gsl_matrix_get (mmp->codebook[depth], 2*i, source);
82                  buf2 = gsl_matrix_get (mmp->codebook[depth], 2*i+1, source);
83                  gsl_matrix_set (mmp->codebook[depth - 1], i, dest, fabs((buf1+buf2)/2));
84          }
85          buf1 = gsl_matrix_get (mmp->codebook[depth], 2*(lim-1), source);
86          gsl_matrix_set (mmp->codebook[depth - 1], lim-1, dest, buf1);
87  }
88
89  int mmp_update_codebook (MMP_CTX *mmp, int left_code, int right_code, int depth,
90                  int *new_code)
91  {
92          double temp;
93          int i, min_code, min_code2, lim, depth_temp;
94
95          // Procura indice com menor pontuacao nas estatisticas para substituicao
96          min_code = mmp_get_cb_index (mmp, depth);
97          // Limpa estatistica da palavra a ser inserida
98          gsl_matrix_set (mmp->codebook_stats, depth, min_code, 1);
99
100         // Tamanho das palavras originais
101         lim = pow(2,depth - 1);
102         for (i = 0; i < lim; i++)
103         {
104                 // Junta duas palavras do nivel inferior e insere no atual
105                 temp = gsl_matrix_get (mmp->codebook[depth - 1], i, left_code);
106                 gsl_matrix_set (mmp->codebook[depth], i, min_code, temp);
```

```
107                     temp = gsl_matrix_get (mmp->codebook[depth - 1], i, right_code);
108                     gsl_matrix_set (mmp->codebook[depth], i + lim, min_code, temp);
109             }
110
111             // Retorna o codigo da nova palavra
112             *new_code = min_code;
113
114             depth_temp = depth;
115             // Insere nos niveis superiores versoes expandidas do sinal
116             while (depth < NUMLEVEL - 1)
117             {
118                     // Posicao onde vou inserir
119                     min_code2 = mmp_get_cb_index (mmp, depth + 1);
120                     // Limpa estatistica da palavra a ser inserida
121                     gsl_matrix_set (mmp->codebook_stats, depth + 1, min_code2, 1);
122                     // Insere expansao
123                     mmp_insert_expansion (mmp, depth, min_code, min_code2);
124                     // Sobe um nivel
125                     depth++;
126                     min_code = min_code2;
127             }
128
129             depth = depth_temp;
130             min_code = *new_code;
131             // Insere nos niveis inferiores versoes contraidas do sinal
132             while (depth > 1)
133             {
134                     // Posicao onde vou inserir
135                     min_code2 = mmp_get_cb_index (mmp, depth -1);
136                     // Limpa estatistica da palavra a ser inserida
137                     gsl_matrix_set (mmp->codebook_stats, depth - 1, min_code2, 1);
138                     // Insere contracao
139                     mmp_insert_contraction (mmp, depth, min_code, min_code2);
140                     // Desce mais um nivel
141                     depth--;
142                     min_code = min_code2;
143             }
144
145             return 0;
146     };
147
148     int mmp_get_raw_data (MMP_CTX *mmp, double *data)
149     {
150             unsigned short temp;
151             int ret;
152
153             ret = fread(&temp, sizeof(short), 1, mmp->instream);
154             *data = (double) temp;
155
156             return ret;
157     }
```

# libencode.c

```
1  #include "libmmp.h"
2
3  int mmp_vq (MMP_CTX *mmp, gsl_vector *block, int depth, int *code, double *distance)
4  {
5          double acum, min = HUGE_VAL;
6          int i;
7            *code = 0;
8          for (i = 0; i< mmp->cb_limit[depth]; i++)
9          {
10                 gsl_matrix_get_col(mmp->temp[depth], mmp->codebook[depth], i);
11                 gsl_vector_sub(mmp->temp[depth], block);
12                 acum = gsl_blas_dnrm2(mmp->temp[depth]);
13                 if (acum < min)
14                 {
15                         min = acum;
16                         *code = i;
17                 }
18         }
19         *distance = min;
20         return 0;
21 }
22
23 int mmp_recursive_encode (MMP_CTX *mmp, gsl_vector *block, int *final_node,
24                 int *parent_code, double *parent_distance)
25 {
26         gsl_vector_view left_block, right_block;
27         double parent_cost, child_cost, left_distance, right_distance;
28         int depth, left_code, right_code;
29         int final_left, final_right;
30
31         depth = (int) rint(log(block->size)/log(2));
32         *final_node = 0;
33
34         if (depth == 0)
35         {
36                 mmp_write_code(mmp, *parent_code, depth);
37                 *final_node = 1;
38                 return 0;
39         }
40
41         parent_cost = *parent_distance + mmp->lambda * (NUMLEVEL - depth);
42
43         left_block  = gsl_vector_subvector (block, 0, block->size/2);
44         right_block = gsl_vector_subvector (block, block->size/2, block->size/2);
45
46         mmp_vq(mmp, &left_block.vector, depth - 1, &left_code, &left_distance);
47         mmp_vq(mmp, &right_block.vector, depth - 1, &right_code, &right_distance);
48         child_cost = (left_distance + right_distance + 2 * mmp->lambda *
49                         (NUMLEVEL - depth + 1))/2;
50
51         if (parent_cost <= child_cost)
52         {
```

```
53                  mmp_write_final_node(mmp);
54                  mmp_write_code(mmp, *parent_code, depth);
55                  mmp_update_stats(mmp, *parent_code, depth);
56                  *final_node = 1;
57          }
58          else
59          {
60                  mmp_write_division(mmp);
61                  mmp_recursive_encode(mmp, &left_block.vector, &final_left,
62                                  &left_code, &left_distance);
63                  mmp_recursive_encode(mmp, &right_block.vector, &final_right,
64                                  &right_code, &right_distance);
65                  if ((final_left == 1) && (final_right == 1))
66                  {
67                          mmp_update_codebook(mmp, left_code, right_code, depth, parent_code);
68                          *final_node = 1;
69                  }
70          }
71
72          return 0;
73  }
74
75  int mmp_write_final_node (MMP_CTX *mmp)
76  {
77          ac_encode_symbol (&mmp->ac_enc, &mmp->ac_division_model, 0);
78          return 0;
79  };
80
81  int mmp_write_code (MMP_CTX *mmp, int code, int depth)
82  {
83          ac_encode_symbol (&mmp->ac_enc, &mmp->ac_codebook_model[depth], code);
84          return 0;
85  };
86
87  int mmp_write_division (MMP_CTX *mmp)
88  {
89          ac_encode_symbol (&mmp->ac_enc, &mmp->ac_division_model, 1);
90          return 0;
91  };
92
93  int mmp_write_eof (MMP_CTX *mmp)
94  {
95          ac_encode_symbol (&mmp->ac_enc, &mmp->ac_division_model, 2);
96          return 0;
97  }
98
99  int mmp_encode (MMP_CTX *mmp)
100 {
101         int fn, code, bl = 0, i = 0;
102         double temp, distance;
103
104         gsl_vector *block = gsl_vector_calloc (BLOCKSIZE);
105
106         // Enquanto houver dados na entrada
```

```
107         while (mmp_get_raw_data (mmp, &temp))
108         {
109                 // Preenche um vetor com o dado novo e incrementa o contador
110                 gsl_vector_set(block, i, temp);
111                 i++;
112
113                 // Quando o vetor contiver um bloco completo
114                 if (i == BLOCKSIZE)
115                 {
116
117                         // Codifica o bloco
118                         mmp_vq(mmp, block, NUMLEVEL-1 , &code, &distance);
119                         mmp_recursive_encode (mmp, block, &fn, &code, &distance);
120
121                         // Atualiza os contadores e imprime mensagem na tela
122                         // com numero de blocos processados
123                         i = 0;
124                         bl++;
125                         printf("\r%i",bl);
126                         fflush(stdout);
127                 }
128         }
129         // Envia um simbolo de fim de arquivo para a saida
130         mmp_write_eof (mmp);
131
132         // Libera o vetor temporario
133         gsl_vector_free(block);
134
135         printf("\n");
136         return 0;
137 }
138
139 // Inicializa a estrutura utilizada pelo codificador
140 MMP_CTX *mmp_encoder_init(double lambda, char *infile, char *outfile)
141 {
142         int i;
143         MMP_CTX *ret;
144
145         // Aloca espaco para a estrutura
146         ret = calloc(1, sizeof(MMP_CTX));
147
148         // Preenche o valor da constante lambda
149         ret->lambda = lambda;
150
151         // Preenche os valores de quantizacao
152         ret->minrange = 0;
153         ret->maxrange = 65535;
154         ret->step = 16;
155
156         // Inicializa a estrutura do codificador aritmetico e abre
157         //        o stream de saida
158         ac_encoder_init (&ret->ac_enc, outfile);
159
160         // Incializa um modelo de codificacao para cada nivel do dicionario
```

```
161         for (i = 0; i< NUMLEVEL; i++)
162         {
163                 ac_model_init (&ret->ac_codebook_model[i], CBSIZE, NULL, 1);
164                 ret->cb_limit[i] = 4096;
165         }
166
167         // Inicializa um modelo de codificacao para os simbolos de divisao e EOF
168         ac_model_init (&ret->ac_division_model, 3, NULL, 1);
169
170         // Abre o stream de entrada
171         ret->instream = fopen (infile, "r");
172
173         // Gera o dicionario
174         mmp_create_codebook (ret);
175
176         return ret;
177 }
178
179 int mmp_encoder_destroy (MMP_CTX *mmp)
180 {
181         int i;
182
183         if (mmp->instream != NULL)
184                 fclose (mmp->instream);
185
186         gsl_matrix_free(mmp->codebook_stats);
187
188         for (i = 0; i < NUMLEVEL; i++)
189         {
190                 gsl_matrix_free(mmp->codebook[i]);
191                 gsl_vector_free(mmp->temp[i]);
192         }
193
194         ac_encoder_done (&mmp->ac_enc);
195
196         for (i = 0; i< NUMLEVEL; i++)
197                 ac_model_done (&mmp->ac_codebook_model[i]);
198
199         ac_model_done (&mmp->ac_division_model);
200
201         free(mmp);
202         return 0;
203 }
```

## libdecode.c

```
1  #include "libmmp.h"
2
3  // Funcao "wrap" que retorna um simbolo de divisao
4  int mmp_get_division (MMP_CTX *mmp)
5  {
6          mmp->code = ac_decode_symbol (&mmp->ac_dec, &mmp->ac_division_model);
7          return 1;
```

```
8  }
9
10 // Funcao "wrap" que retorna um simbolo referente a uma palavra do dicionario
11 int mmp_get_code (MMP_CTX *mmp, int depth)
12 {
13         mmp->code = ac_decode_symbol (&mmp->ac_dec, &mmp->ac_codebook_model[depth]);
14         return 1;
15 }
16
17 /*gsl_vector *hamming(int lenght)
18 {
19         gsl_vector *ret;
20         int i;
21         double val;
22
23         ret = gsl_vector_alloc(lenght+1);
24         for (i= -lenght/2; i < lenght/2+1; i++)
25         {
26                 val = 0.54 + 0.46 * cos (2*M_PI*i/lenght);
27                 gsl_vector_set (ret, i+lenght/2, val);
28         }
29
30         return ret;
31 }*/
32
33 // Filtro para reducao de blocagem
34 void mmp_block_filter (MMP_CTX *mmp, unsigned short *data, int level)
35 {
36         int i, filter_size;
37         int acum = 0;
38
39         //Calcula o tamanho do filtro no ponto atual
40         filter_size = pow(2,level - 1);
41         if (filter_size < 2)
42                 filter_size = 2;
43
44         //Desloca os pontos da memoria do filtro
45         for (i = mmp->filter_memory->size - 1; i > 0; i--)
46         {
47                 gsl_vector_swap_elements (mmp->filter_memory, i, i-1);
48                 gsl_vector_swap_elements (mmp->filter_levels, i, i-1);
49         }
50
51         //Adiciona os novos valores a memoria
52         gsl_vector_set (mmp->filter_memory, 0, *data);
53         gsl_vector_set (mmp->filter_levels, 0, filter_size);
54
55         // Recupero o valor correspondente ao tamanho do filtro armazenado
56         // na metade da memoria
57         filter_size = gsl_vector_get (mmp->filter_levels, mmp->filter_levels->size/2);
58
59         // Aplico o filtro aos dados contidos na metade da memoria
60         for (i = BLOCKSIZE/2 - filter_size/2 ; i < BLOCKSIZE/2 + filter_size/2 + 1; i++)
61                 acum += gsl_vector_get (mmp->filter_memory, i);
```

```
62
63          // Dados filtrados com atraso de BLOCKSIZE/2
64          *data = (unsigned short) (acum / (filter_size + 1));
65  }
66
67  // Recupera um vetor do dicionario e envia para a saida
68  int mmp_output_message (MMP_CTX *mmp, int depth)
69  {
70          int i;
71          unsigned short data;
72
73          // Recupera o vetor do dicionario e armazena em um vetor temporario que
74          // esta alocado na estrutura geral
75          gsl_matrix_get_col(mmp->temp[depth], mmp->codebook[depth], mmp->code);
76
77          for (i = 0; i < mmp->temp[depth]->size; i++)
78          {
79                  // Recupera o dado
80                  data = (unsigned short) gsl_vector_get (mmp->temp[depth], i);
81
82                  // Aplica o filtro de reducao de blocagem
83                  //mmp_block_filter (mmp, &data, depth);
84
85                  fwrite (&data, sizeof(short), 1, mmp->outstream);
86          }
87          return 0;
88  }
89
90  int mmp_recursive_decode(MMP_CTX *mmp, int depth, int *parent_code)
91  {
92          int left_code, right_code;
93
94          *parent_code = -1;
95          left_code = -1;
96          right_code = -1;
97
98          if (depth == 0)
99          {
100                 if (!mmp_get_code (mmp, depth))
101                         return 0;
102
103                 *parent_code = mmp->code;
104                 mmp_output_message (mmp, depth);
105                 return 0;
106         }
107
108         if (!mmp_get_division (mmp))
109                 return 0;
110
111         if (mmp->code == 0)
112         {
113                 if (!mmp_get_code (mmp, depth))
114                         return 0;
115
```

```
116                     *parent_code = mmp->code;
117                     mmp_output_message (mmp, depth);
118                     mmp_update_stats (mmp, mmp->code, depth);
119             }
120             else
121             {
122                     if (mmp->code == 2)
123                             return 1;
124
125                     if (mmp->code == 1)
126                     {
127                             mmp_recursive_decode(mmp, depth - 1, &left_code);
128                             mmp_recursive_decode(mmp, depth - 1, &right_code);
129                     }
130
131                     if ((left_code != -1) && (right_code != -1))
132                             mmp_update_codebook (mmp, left_code, right_code, depth, parent_code);
133             }
134             return 0;
135     }
136
137     int mmp_decode(MMP_CTX *mmp)
138     {
139             int parent_code;
140             int bl = 0;
141
142             // Loop de decodificação, retorna verdadeiro quando consegue
143             // terminar um bloco e enviar para a saida
144             while (mmp_recursive_decode (mmp, NUMLEVEL-1, &parent_code) == 0)
145             {
146                     // Imprime na tela numero de blocos decodificados
147                     bl++;
148                     printf("\r%i",bl);
149                     fflush(stdout);
150             }
151
152             printf("\n");
153
154             return 0;
155     }
156
157     MMP_CTX *mmp_decoder_init(char *infile, char *outfile)
158     {
159             int i;
160             MMP_CTX *ret;
161
162             ret = calloc(1, sizeof (MMP_CTX));
163             ret->minrange = 0;
164             ret->maxrange = 65535;
165             ret->step = 16;
166
167             ret->filter_memory = gsl_vector_alloc (BLOCKSIZE + 1);
168             gsl_vector_set_all (ret->filter_memory, 32768);
169             ret->filter_levels = gsl_vector_alloc (BLOCKSIZE + 1);
```

```
170         gsl_vector_set_all (ret->filter_levels, 2);
171
172         ac_decoder_init (&ret->ac_dec, infile);
173
174         for (i = 0; i< NUMLEVEL; i++)
175         {
176                 ac_model_init (&ret->ac_codebook_model[i], CBSIZE, NULL, 1);
177                 ret->cb_limit[i] = 4096;
178         }
179
180         ac_model_init (&ret->ac_division_model, 3, NULL, 1);
181
182         ret->outstream = fopen (outfile, "w");
183
184         mmp_create_codebook (ret);
185
186         return ret;
187 }
188
189 int mmp_decoder_destroy (MMP_CTX *mmp)
190 {
191         int i;
192
193         if (mmp->outstream != NULL)
194                 fclose (mmp->outstream);
195
196         gsl_matrix_free(mmp->codebook_stats);
197
198         for (i = 0; i < NUMLEVEL; i++)
199         {
200                 gsl_matrix_free(mmp->codebook[i]);
201                 gsl_vector_free(mmp->temp[i]);
202         }
203
204         ac_decoder_done (&mmp->ac_dec);
205
206         for (i = 0; i< NUMLEVEL; i++)
207                 ac_model_done (&mmp->ac_codebook_model[i]);
208
209         ac_model_done (&mmp->ac_division_model);
210
211         free(mmp);
212         return 0;
213 }
```

## mmpencode.c

```
1  #include "libmmp.h"
2
3  void print_help ()
4  {
5          printf("Usage: ./mmpencode lambda inputfile outputfile\n");
6  }
```

```
7
8  int main (int argc, char **argv)
9  {
10         MMP_CTX *mmp;
11
12         if (argc != 4)
13         {
14                 print_help();
15                 exit(0);
16         }
17
18         // Inicializa a estrutura do codificador com os argumentos
19         // recebidos da linha de comando
20         mmp = mmp_encoder_init (atof(argv[1]), argv[2], argv[3]);
21
22         // Comeca o loop de codificacao
23         mmp_encode (mmp);
24
25         // Limpa as referencias alocadas durante o processo
26         mmp_encoder_destroy (mmp);
27
28         exit(0);
29 }
```

## mmpdecode.c

```
1  #include "libmmp.h"
2
3  void print_help ()
4  {
5          printf("Usage: ./mmpdecode inputfile outputfile\n");
6  }
7
8  int main (int argc, char **argv)
9  {
10         MMP_CTX *mmp;
11
12         if (argc != 3)
13         {
14                 print_help();
15                 exit(0);
16         }
17
18         // Inicializa a estrutura do decodificador com os argumentos
19         // recebidos da linha de comando
20         mmp = mmp_decoder_init (argv[1], argv[2]);
21
22         // Comeca o loop de codificacao
23         mmp_decode (mmp);
24
25         // Limpa as referencias alocadas durante o processo
26         mmp_decoder_destroy (mmp);
27
```

```
28            exit(0);
29    }
```