

Voice Coder

Nuno Morgadinho <l13591@alunos.uevora.pt>
Cláudio Fernandes <l14103@alunos.uevora.pt>

21 de Julho de 2003

Resumo

Implementação de um codificador/descodificador de voz, o qual, baseado no método de análise/síntese de voz LPC e utilizando o algoritmo de Levinson-Durbin, permita a compressão/descompressão de voz.

KEYWORDS: Linear Prediction, Linear Prediction Coding, LPC, Levinson-Durbin, Compressão de Voz



Conteúdo

1	Introdução	3
2	Explicação do funcionamento	4
2.1	Voz Humana: Como é produzida?	4
2.2	Teorema da Amostragem	5
2.3	Modelo LPC	6
2.3.1	Codificação, Compressão ou Análise LPC	6
2.3.2	Descodificação, Descompressão ou Síntese LPC	7
3	Parte Teórica	8
3.1	Codificação, Compressão ou Análise LPC	9
3.1.1	Coeficientes LPC	9
3.1.2	Cálculo do Pitch e Classificação Vozeado/Não Vozeado	11
3.1.3	Cálculo do Ganho e Detecção de Início/Fim da Fala	11
3.2	Descodificação, Descompressão ou Síntese LPC	12
4	Construção de Exemplos	13
4.1	Compressão Obtida	14
5	Conclusão e Trabalho Futuro	16
6	Apêndice 1: Como visualizar a onda de som de um ficheiro de audio no octave	18
7	Apêndice 2: Código	19

1 Introdução

A compressão ou codificação de voz é normalmente realizada usando voice coders ou vocoders.

A compressão de voz é muitas vezes designada de codificação de voz, visto que define um método para reduzir a quantidade de informação necessária para representar um sinal de voz. Normalmente, estes métodos são baseados em algoritmos com perdas (lossy compression) mas são aceitáveis porque a perda de qualidade não é perceptível ao ouvido humano.

Mas como comprimir voz? O que é a voz? Como é produzida? Como pode ser representada?

Por exemplo, uma técnica de compressão trivial e economizadora de largura de banda é não transmitir o silêncio visto que segundo observações efectuadas, em conversação normal %50 da conversa é silêncio.

Mas esta não é suficiente para as necessidades exigidas pelas telecomunicações. Cada vez mais, se quer transmitir voz em alta qualidade, ocupando o mínimo necessário em termos de canal de transmissão, nomeadamente em largura de banda.

Linear Predictive Coding (LPC) é um método de compressão que modela (cria uma maquete da realidade) do processo da produção de voz. Especificamente, o método LPC modela este processo usando um filtro digital e calculando os parâmetros da produção de voz.

Uma explicação mais simples será dizer que no método LPC transmitem-se estes parâmetros do processo de produção de voz em vez de transmitir a voz em si. Como tal, nunca irá ser reproduzido o sinal original de voz, independentemente da ocorrência de erros ou não.

Foi proposto pela primeira vez como um método para codificar voz humana pelo Departamento de Defesa dos Estados Unidos em 1984, através do federal-standard 1015/LPC-10.

Todos os vocoders têm 4 atributos essenciais: bit rate, complexidade, qualidade e o delay. Qualquer vocoder, independentemente do algoritmo que use terá de efectuar trade-offs entre estes 4 atributos.

O primeiro atributo, a bit rate, é usada para determinar o grau de compressão que o vocoder consegue atingir. Um sinal de voz sem ser comprimido é normalmente transmitido a uma taxa de 64 kbps fazendo o sampling a 8 kHz com samples de 8 bits. O LPC vocoder transmite a uma bit rate de 2.4 kbps.

O segundo atributo, a complexidade do algoritmo afecta mutuamente o custo e a sua eficácia. O método LPC devido à sua alta taxa de compressão envolve milhares de computações por segundo.

O terceiro atributo é a qualidade. A qualidade é uma medida subjectiva e depende de ouvinte para ouvinte. Um dos testes mais comuns para medir a qualidade é o 'absolute category rating' (ACR) em que um conjunto de perguntas são feitas a um conjunto de ouvintes que de seguida classificam a qualidade com uma escala do tipo Muito boa, Boa,

Suficiente, Má, Muito Má.

Outro atributo também importante é o delay com que o sinal de voz chega. Geralmente, um delay maior que 300ms é considerado inaceitável.

LPC coders sacrificam a qualidade por uma bit rate menor e como resultado disso apresentam uma voz que soa sintetizada (como um robot a falar).

O algoritmo consiste numa parte de análise ou codificação e numa parte de síntese ou decodificação.

A análise e a síntese de voz através do LPC explora a natureza previsível dos sinais de voz. A autocorrelação e a autocovariância fornecem as ferramentas matemáticas para determinar essa previsibilidade.

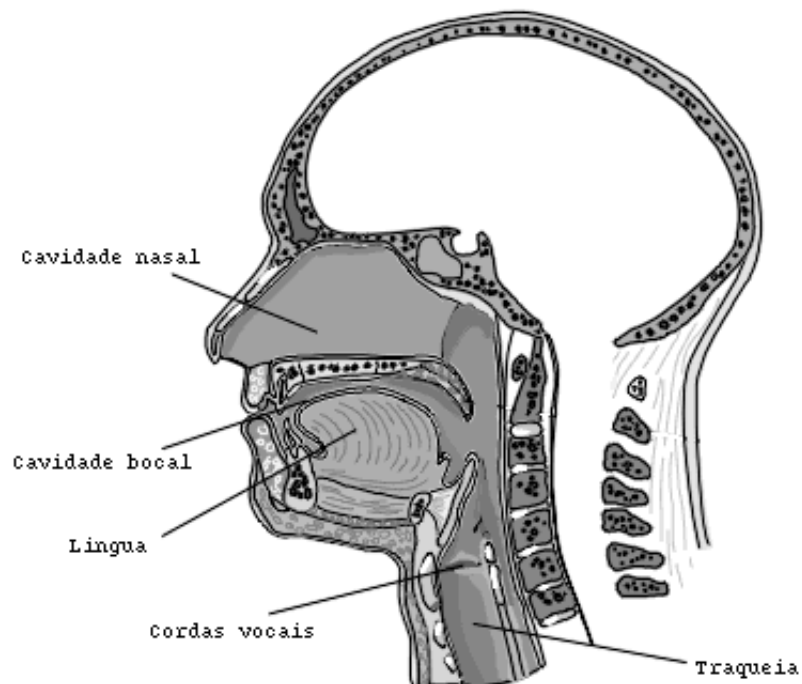
Se soubermos a autocorrelação de um sinal de voz, podemos usar o algoritmo de Levinson-Durbin para encontrar uma solução eficiente para o problema de otimização dos coeficientes de previsibilidade e usar a solução para comprimir ou re-sintetizar a voz.

2 Explicação do funcionamento

2.1 Voz Humana: Como é produzida?

Quando se fala, ar é puxado dos pulmões, passa pela garganta e pelas cordas vocais, sai pela boca e é produzida voz.

Ao falar, o tracto vocal muda de forma, produzindo diferentes sons.



Este conjunto limitado de sons é classificado em dois tipos de sons. Os sons sonoros (ou vozeados) que representam o vibrar das cordas e os sons surdos (ou não vozeados), para os quais as cordas vocais não vibram, apenas permanecem abertas.

É importante referir que, a classificação nestes dois tipos é essencial no processo de análise/síntese de um sinal de voz, se queremos conseguir reproduzir o original, uma vez que este, é constituído de sons vozeados e não vozeados.

Exemplos de sons vozeados são os sons produzidos na pronúncia das vogais “a”, “e”, “i”, “o”, “u”. Por outro lado, a pronúncia de letras como o “f” e o “s” no meio de palavras são exemplos de sons não vozeados.

Quanto à vibração das cordas vocais, esta é outro dos factores chave na produção de diferentes sons e acontece com uma certa frequência (ou taxa) que é designada de ‘pitch’ e que varia com a voz e de pessoa para pessoa. As mulheres e as crianças têm normalmente uma frequência maior (vibração rápida) que os homens adultos (vibração lenta).

A quantidade de ar vinda dos pulmões determina a altura (ou volume) da voz.

2.2 Teorema da Amostragem

A voz produzida é captada por um microfone.

O microfone encarrega-se de transformar as vibrações do ar criadas pelas ondas de som (movimentos acústicos) em vibrações eléctricas (ou corrente eléctrica). Esta conversão é relativamente directa e as vibrações eléctricas podem então ser amplificadas, gravadas ou transmitidas.

Ao serem gravadas, transforma-se a voz da forma de sinal analógico (de natureza contínua) para a forma digital (sinal baseado em impulsos, discreto).

Para converter um sinal analógico para a forma digital, primeiro tem de se passar o sinal para um que tenha largura de banda limitada e depois fazer a amostragem do mesmo.

Um sinal diz-se que tem largura de banda limitada se é composto por sinusóides com frequências que vão até uma dada frequência máxima F_{max} , i.e., não tem sinusóides de frequência maior que F_{max} .

O teorema de Nyquist (também conhecido como teorema da amostragem) diz que se amostrarmos um sinal contínuo com largura de banda F_{max} a uma frequência maior ou igual a duas vezes F_{max} , então o sinal amostrado contém toda a informação do sinal contínuo e consegue-se recuperar exactamente o sinal original a partir das amostras.

Isto significa que se for usada uma frequência de amostragem de 8000 Hz consegue-se ter toda a informação de um sinal que tenha largura de banda 4000 Hz.

Como o sinal de fala está praticamente todo até aos 4 KHz, pode usar-se em aplicações telefónicas (como esta) frequências de amostragem de 8000 Hz.

Os parâmetros de gravação são importantes: 8/16 bits (tamanho de cada amostra/sample), 11/22/44 KHz (frequência de amostragem/sampling rate). Em particular, a frequência

de amostragem vai influenciar a qualidade de som e conseqüentemente o tamanho.

O que fica gravado em suporte digital são as amostras e normalmente os parâmetros da gravação (que especificam como a gravação foi feita).

2.3 Modelo LPC

Pode-se então pensar nos pulmões como sendo a origem da voz (ou fonte), e o tracto vocal como um filtro que produz os vários tipos de som que compõem a fala (fonemas).

Esta ideia, baseada na separação da fonte do filtro, é o chamado modelo fonte-filtro e é o modelo usado na análise e síntese do LPC, em que o tracto vocal, representado como um tubo de diâmetro variável, é aproximado matematicamente.

Note-se que os vocoders baseados em modelos não reproduzem o sinal de voz original porque baseiam-se em encontrar, codificar e transmitir parâmetros da produção de voz e não a voz em si.

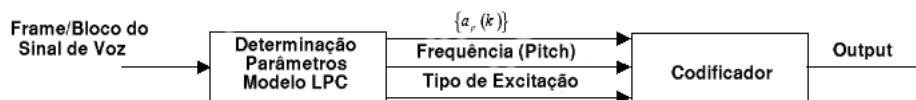
Uma vez que os sinais de voz variam com o tempo, este processo é feito em pequeno pedaços do sinal, designados de frames. Estes parâmetros mudam sensivelmente a cada 20 msec, pelo que considera-se o sinal de voz dividido em blocos/frames deste tamanho.

Sendo assim, existem 50 frames por segundo. (20 msec x 50 = 1 segundo).

A uma sampling rate de 8000 samples por segundo, 20 msec é equivalente a 160 samples, pelo que cada frame contém 160 samples de 8 bits.

2.3.1 Codificação, Compressão ou Análise LPC

Para cada frame, são calculados os parâmetros da produção de voz que são: os coeficientes do LPC, o pitch (frequência), a classificação vozeado/não vozeado e o ganho (volume).



Historicamente, o sampling de sinais de voz é feito a 8000 samples/segundo. Cada sample é normalmente de 8 ou 16 bits.

Isto corresponde a uma taxa de transmissão sem compressão de 64 ou 128 kbps. Ao transmitir os parâmetros de produção de voz por cada frame em vez das 160 samples, esta técnica de compressão, classificada de compressão com perdas (lossy), consegue reduzir a taxa de transmissão para 8 kbps, teoricamente sem perda de qualidade perceptível ao ouvido humano.

O compromisso da compressão é a qualidade de som. Quanto mais compressão se quiser ter, mais qualidade de som se terá de abdicar. O que se faz normalmente é tentar abdicar da qualidade de som que não é perceptível ao ouvido humano.

Os vocoders baseados no Modelo LPC sacrificam então a qualidade de voz por uma taxa de transmissão menor (e consequentemente mais rápida) e como resultado disso apresentam uma voz que soa sintetizada (como um robot a falar).

2.3.2 Descodificação, Descompressão ou Síntese LPC

Cada bloco/frame é decodificado individualmente e a sequência dos blocos/frames de voz, quando reproduzida (ou sintetizada), irá assemelhar-se ao sinal de voz dado como input.

O decodificador, efectua a síntese LPC usando os parâmetros de produção de voz que recebe do codificador para construir um filtro, o chamado filtro LPC, que tenta imitar o tracto vocal no processo de produção de voz humana.

O filtro LPC não passa de um filtro digital, uma técnica usada em 'Digital Signal Processing' (DSP).

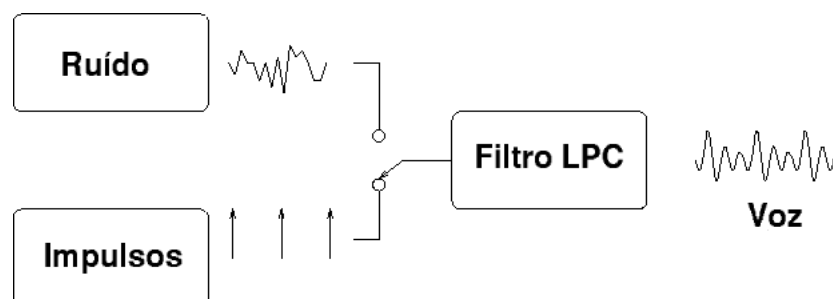
Esta técnica consiste geralmente numa transformação de um número de samples adjacentes a uma sample do sinal de input.

Os filtros designados de Finite Impulse Response (FIR) baseiam-se apenas no sinal de input.

Os filtros designados de Infinite Impulse Response; são um caso particular dos anteriores e baseiam-se no sinal de input e nas samples anteriores do sinal de output.

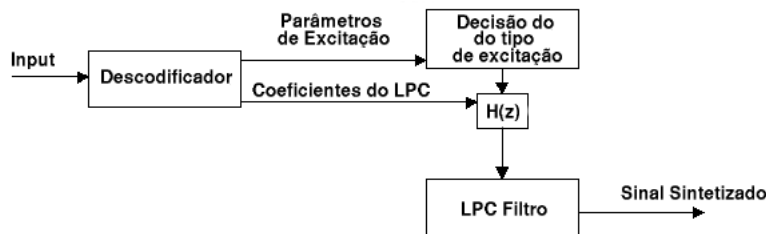
O filtro LPC, quando lhe é passado o sinal de excitação correcto é capaz de reproduzir o sinal de voz original (imitando então o processo de produção de voz humana).

Mais propriamente, a excitação ou turbulência do ar que percorre o tracto vocal é a fonte do som. Esta pode ser periódica, quando se produz sons vozeados, ou aleatória, quando se produz sons não vozeados.



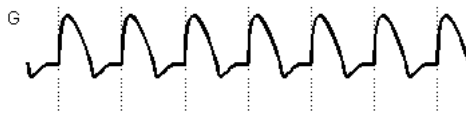
Cada frame é então analisado para determinar se é vozeado ou não vozeado conforme o parâmetro do pitch.

Isto exactamente para distinguir o tipo de excitação que deve ser usada no filtro LPC.



Para frames considerados vozeados é usada uma sequência de impulsos como excitação. O cálculo do pitch no codificador serve agora para determinar a frequência de cada impulso. Caso não seja vozeado, é usado uma sequência de números pseudo-aleatórios.

A sequência de impulsos é uma sequência como: 0, ...,0,1,0, ...,0,1, ...em que o “um” ocorre com a frequência que se quiser.



Por exemplo, caso se considere frames de 160 samples (equivalente a 20 msec) e uma frequência de 50 Hz, um impulso ocorrerá 50 vezes por segundo no início da sequência.

Cada segmento de voz tem um filtro LPC diferente que é produzido usando os coeficientes e o ganho que são recebidos do codificador.

O último passo para decodificar um segmento de voz é passar o sinal de excitação no filtro de maneira a sintetizar a voz.

Para garantir que estamos a fazer a síntese correctamente, pode-se comparar a onda do sinal de voz dado como input e a onda de voz sintetizada. Isto pode ser feito por exemplo usando o GNU Octave e o GNU Plot (ver apêndice 1).

3 Parte Teórica

Interessa agora introduzir os conceitos teóricos que estão na base do Modelo-LPC e dos vocoders nele baseados.

Uma observação muito importante é a de que matematicamente existe uma correlação muito grande entre samples adjacentes de um sinal de voz.

A autocorrelação é, basicamente, definida como o valor esperado da correlação de um sinal com ele próprio com um certo desfaseamento.

A ideia consiste então em estimar a próxima sample do sinal $x[n]$ com base nas N samples anteriores.

Mais precisamente, esta estimativa consiste na operação matemática conhecida como 'Linear Prediction', ou seja, é feita a 'Linear Prediction' da sample N como uma soma

ponderada das samples anteriores.

O processo é também, algumas vezes, referido como 'short-term prediction' devido ao facto de estimar o sinal $x[n]$ com base apenas nas N samples anteriores, onde N é normalmente à volta de 10.

3.1 Codificação, Compressão ou Análise LPC

A mais comum representação para a operação 'Linear Prediction' é dada por:

$$y[n] = \sum_{i=1}^N a_i x[n-i]$$

onde $y[n]$ é a 'linear prediction' de um sinal $x[n]$. O erro gerado por esta estimativa (ou do predictor) é então dado por:

$$e[n] = x[n] - y[n] = x[n] - \sum_{i=1}^N a_i x[n-i]$$

Onde $x[n]$ é o verdadeiro valor do sinal e $y[n]$ o valor estimado.

3.1.1 Coeficientes LPC

O objectivo do LPC é encontrar os melhores coeficientes a_i do predictor, ou seja, os que melhor representam o frame que está a ser analisado. A escolha mais comum para a optimização destes coeficientes é pelo critério da raiz quadrada do erro. Note-se que esta abordagem não é única e que a optimização dos parâmetros é um vasto tópico.

Assim sendo, o que se quer é encontrar os coeficientes de forma a minimizar a função do chamado erro quadrático dada por:

$$E = \sum_{n=0}^{L-1} [e[n]]^2 = \sum_{n=0}^{L-1} [x[n] - \sum_{i=1}^N a_i x[n-i]]^2$$

O processo de encontrar estes coeficientes pode ser feito fazendo todas as derivadas $\frac{dE}{da_i}$ iguais a zero:

$$\frac{dE}{da_i} = \frac{d}{da_i} \sum_{n=0}^{L-1} [x[n] - \sum_{i=1}^N a_i x[n-i]]^2 = 0$$

O que resulta na equação:

$$y[n] = \sum_{i=1}^p a_i R(i-j) = -R(j)$$

para $1 \leq j \leq p$, onde R é a autocorrelação do sinal $x(n)$ e p o número de coeficientes calculados para cada frame.

As equações acima descritas são chamadas de equações normais ou equações de Yule-Walker. Em forma de matriz, as equações podem ser escritas de maneira equivalente como $R * a = r$, onde a matriz de autocorrelação R é uma matriz de Toeplitz¹ com elementos

¹O nome vêm do matemático Alemão Otto Toeplitz (1881-1940)

$r_{i,j} = R(i-j)$ e o vector r é o vector de autocorrelação $r_j = R(j)$ e o vector a é o vector dos parâmetros a_i .

Calcular a solução da matriz $R^{-1}a = r$ é um processo relativamente pesado em termos de computação. O método de eliminação de Gauss é provavelmente a solução mais antiga para resolver matrizes mas este não tira proveito eficazmente da simetria de R e r .

Um algoritmo bem mais rápido é o da recursividade de Levinson, proposto por N. Levinson em 1947 e modificado mais tarde em 1959 por J. Durbin, originando o nome Levinson-Durbin para este algoritmo que recursivamente calcula a solução da matriz.

Os coeficientes a_i do filtro são então calculados usando este algoritmo que parte da auto-correlação $R(m)$.

A auto-correlação requiere que algumas suposições iniciais sejam feitas, nomeadamente, que $x[n]$ seja considerado zero fora do frame actual.

A auto-correlação do sinal $x[n]$ é dada por:

$$R(m) = \sum_{i=0}^{N-i} x[i]x[i-m]$$

Para um filtro de ordem $N = 10$, temos então:

$$R = \begin{pmatrix} R(0) & R(1) & \dots & R(9) \\ R(1) & R(0) & \dots & R(8) \\ \vdots & \vdots & \ddots & \vdots \\ R(9) & R(8) & \dots & R(0) \end{pmatrix}$$

$$r = \begin{pmatrix} R(1) \\ R(2) \\ \vdots \\ R(10) \end{pmatrix}$$

Os coeficientes a_i do filtro são encontrados resolvendo o sistema $Ra = r$. O algoritmo de Levinson-Durbin permite chegar à solução com $O(N^2)$ em vez de $O(N^3)$ usando o conhecimento de que a matriz R é uma matriz de Toeplitz.

Recursividade de Levinson Durbin:

$$E^{(0)} = R(0)$$

$$k_i = [R(i) - \sum_{j=1}^{i-1} \alpha_j^{i-1} R(i-j)]/E^{(i-1)} \quad \text{com } i = 1, 2, \dots, 10$$

$$\alpha_i^{(i)} = k_i$$

$$\alpha_j^{(j)} = \alpha_j^{(i-1)} - k_i \alpha_{(i-j)}^{(i-1)} \quad \text{com } j = 1, 2, \dots, i-1$$

$$E^{(i)} = (1 - k_i^2)E^{(i-1)}$$

Resolve-se para $i = 1, 2, \dots, 10$ e depois atribui-se:

$$a_i = -\alpha_i^{10}$$

3.1.2 Cálculo do Pitch e Classificação Vozeado/Não Vozeado

Determinar se um frame do sinal é vozeado ou não é de extrema importância para posteriormente se poder fazer a síntese correctamente.

Como já foi referido anteriormente, os segmentos não vozeados representam a pronúncia de letras como o 's' e o 'f' e têm amplitudes pequenas comparadas com as amplitudes de um sinal de voz considerado vozeado.

Estas diferenças criam a necessidade de usar dois tipos diferentes de excitação para serem passados para o filtro LPC. O parâmetro vozeado/não vozeado, dá então a informação de que tipo de excitação deve ser passado ao filtro.

Não devíamos dizer aqui como se processa isto?

Uma das ideias para determinar se um frame é vozeado ou não vozeado é tirar partido do facto que a auto-correlação de uma função periódica $r(k)$ qualquer, como um sinal de voz, tem um máximo quando o k é equivalente ao período do pitch.

Cálculo do Pitch:

Computa-se a sequência:

$$r_e(n) = \sum_{k=1}^p r_a(k)r_{ss}(n-k)$$

onde

$$r_a(k) = \sum_{i=1}^p a_p(i)a_p(i+k)$$

é a auto-correlação dos coeficientes LPC.

O pitch é igual ao máximo da sequência $r_e(n)/r_e(0)$ (sequência anterior normalizada) no intervalo de tempo que corresponde a 3 msec - 15 msec em frames de 20 msec.

Se o valor encontrado for maior ou igual que 0.25, o frame é considerado vozeado com um pitch de período igual a n , onde $r_e(n)/r_e(0)$ é um máximo.

Se for menor que 0.25, o frame é considerado não vozeado e o pitch é zero.

3.1.3 Cálculo do Ganho e Detecção de Início/Fim da Fala

Uma técnica para reduzir a quantidade de informação necessária e poupar largura de banda é não transmitir o silêncio visto que cerca de 50% de conversa em conversação normal é silêncio.

Sendo assim, é necessário uma medida para distinguir o silêncio do sinal de voz.

O ganho é o volume e este não é constante num sinal de voz.

Como tal, sobre cada frame calcula-se uma função de "energia" que dá a intensidade de som existente nessa frame, cujo valor é o ganho.

Se estiver acima de um dado valor é falado caso contrário é considerado silêncio, ou seja, energia baixa implica silêncio ou ruído de fundo e uma energia alta som ou fala.

Convém no entanto ajustar um ganho para que quando se comuta entre sons vozeados ou não vozeados haja um salto grande no volume de som.

Como cada frame é uma variável aleatória contínua uniforme que está entre -1 e 1, podemos escrever a função de energia como:

$$Energia = \sum_i^N E[x(k)^2]$$

em que E é o valor esperado e $E[x(k)^2]$ representa a variância.

3.2 Descodificação, Descompressão ou Síntese LPC

Uma sequência de samples ao longo do tempo produz um sinal digital no domínio do tempo, onde a partir da qual se aplicarmos a transformada de Fourier discreta se produz uma representação no domínio da frequência.

A maior parte dos filtros pode, no domínio-Z (domínio da frequência é um subconjunto do domínio-Z), ser descrito pela sua função de Transferência.

Como já foi referido, o 'Linear Predictor' representa cada sample e mais um sinal de erro designado de resíduo.

O filtro é então representado como:

$$x[n] = \sum_{i=1}^N a_i x[n-i] + e[n]$$

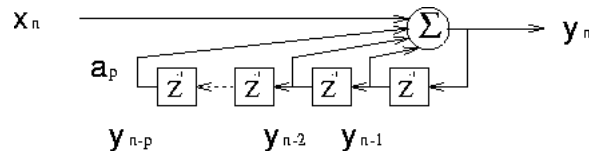
Onde no dominio-Z, equivale a:

$$x(z) = \frac{1}{A(z)} e(z)$$

onde $A(z)$ é definida como:

$$A(z) = 1 - \sum_{i=1}^N a_i z^{-i}$$

Normalmente designa-se $A(z)$ como o filtro da análise LPC e $\frac{1}{A(z)}$ como o filtro da síntese.



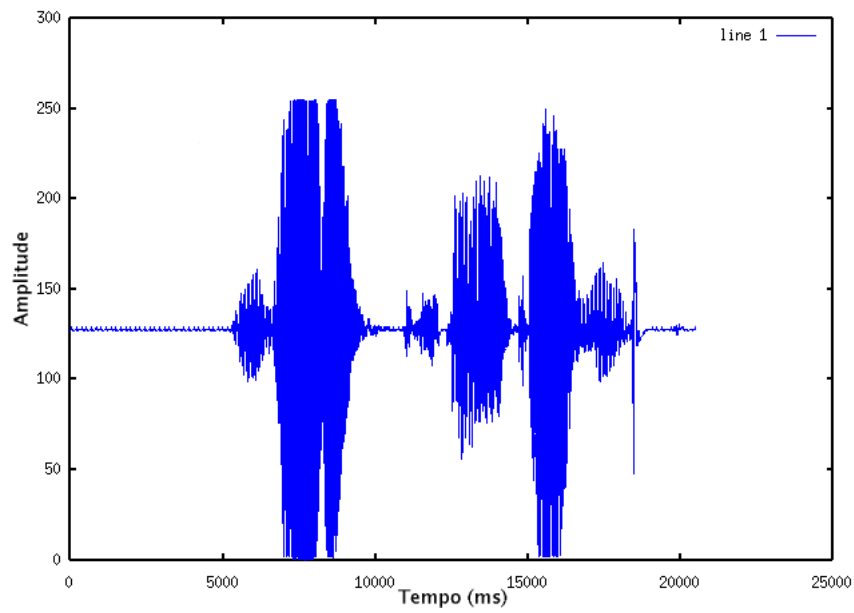
Os vocoders baseados no Modelo-LPC usam um filtro IIR (figura acima) tendo como função de Transferência:

$$H(z) = \frac{G}{1 + \sum_{k=1}^p a_p(k)z^{-k}}$$

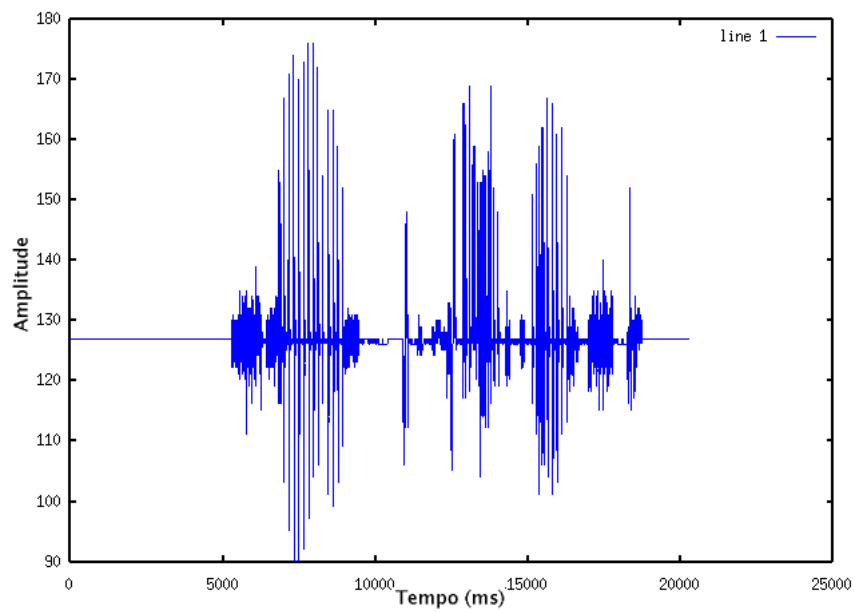
Onde G é o ganho do filtro e p é o número de coeficientes LPC.

4 Construção de Exemplos

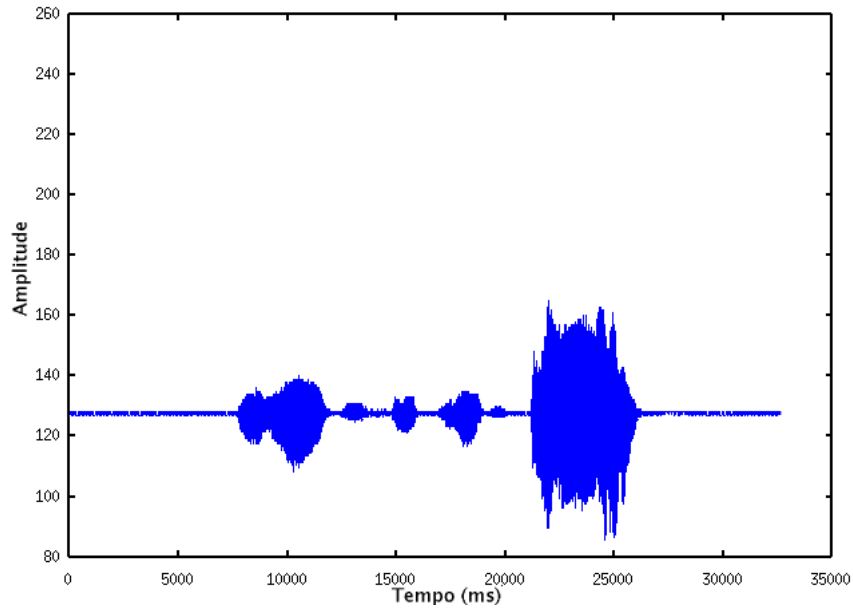
Sinal original da mensagem “Agora estou em casa”



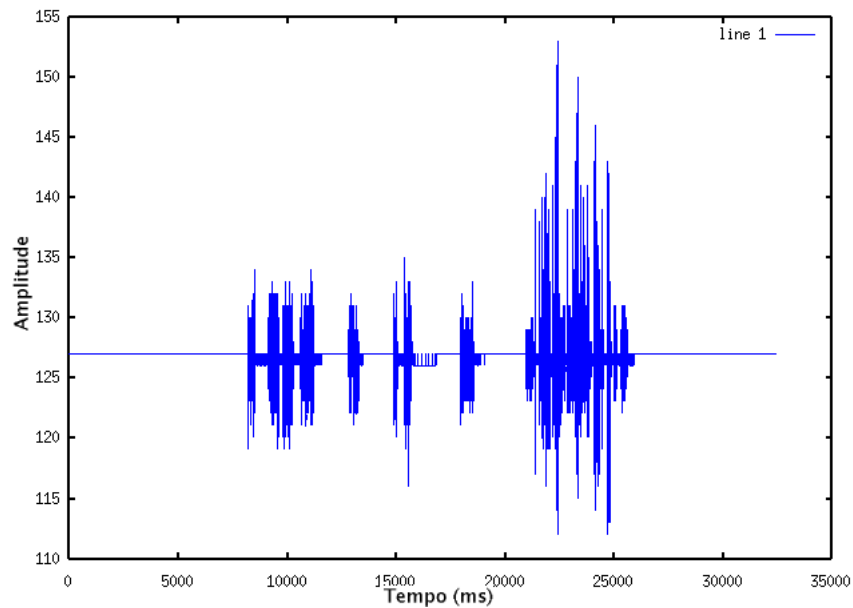
Sinal sintetizado



Sinal original da mensagem “Olá, eu sou o Cláudio AAAhhh”



Sinal sintetizado



4.1 Compressão Obtida

A voz é normalmente transmitida, sem compressão, à taxa de 64 kbps usando samples de 8 bits e uma taxa de amostragem de 8 kHz.

LPC reduz esta taxa, dividindo o sinal de voz em segmentos/frames e transmitindo os parâmetros LPC que dizem respeito a cada frame em vez das próprias samples.

Para cada frame, os parâmetros LPC são:

- 10 coeficientes LPC - double, 8 bytes;
- pitch - short int, 2 bytes;
- ganho - float, 4 bytes;

O que totaliza 688 bits por cada segmento do sinal de voz.

- Taxa de amostragem = 8000 samples por segundo
- Número de Samples por Frame = 160 samples
- Taxa de Frames Transmitidos por segundo = $\frac{\text{Taxa de Amostragem}}{\text{Num de Samples por Frame}}$
 $= \frac{8000}{160}$
 $= 50$
- Número de Bits por Frame = 688 bits
- Bit Rate = Número de Bits por Frame x Taxa de Frames Transmitidos por segundo
 $= 688 \times 50$
 $= 34400 \text{ bits/segundo}$

5 Conclusão e Trabalho Futuro

O processo de codificação de voz é um em que se percorre um caminho sinuoso na busca de algoritmos e métodos cuja adaptabilidade e inteligência sigam os parâmetros da produção de voz humana natural, um processo que é regido pelas leis da física mas cuja imitação matemática exacta é ainda inviável.

No entanto, as técnicas de DSP como a filtragem e o Modelo LPC fornecem um método de codificação de voz, que em conjunto com o algoritmo de Levinson-Durbin e as ferramentas matemáticas como a autocorrelação e a autocovariância, fornecem uma solução em que se obtém uma taxa de compressão elevada e uma qualidade de voz perceptível, embora soe um pouco a voz sintetizada.

A determinação e a optimização dos parâmetros LPC definem largamente a qualidade de um vocoder é são um vasto tópico de investigação em que existem várias abordagens.

A implementação deste vocoder utiliza as técnicas mais básicas de estimação destes parâmetros enquanto que por exemplo ao analisar a complexidade da decisão vozeado/não vozeado do standart DoD LPC-10 torna-se claro que para se obter bons resultados é necessário implementar um bom, inteligente, adaptável e robusto algoritmo que use diferentes abordagens e técnicas em conjunto.

A continuação deste projecto envolveria:

- O estudo da conjunção das diferentes abordagens de modo a obter uma melhor estimação dos parâmetros.
- Manuseamento dos bits. Diminuir os bits de codificação dos parâmetros, ou seja diminuir a precisão, ao número mínimo que torne o sinal de voz ainda perceptível.
- A compressão dos parâmetros obtidos na codificação para cada frame do sinal de voz. Nomeadamente, usando algoritmos de Clustering como o K-Means. A ideia seria definir previamente os clusters com uns valores iniciais aleatórios (depois seriam afinados..) e por cada 10 coeficientes LPC ver o cluster a que pertenciam e transmitir apenas o centroide do cluster. Os clusters seriam os símbolos de um alfabeto. No lado do decodificador, ao receber o centroide, determinar 10 valores à volta desse centroide e utiliza-los como coeficientes LPC na construção do filtro.

Referências

- [1] Celso Aguiar. *Modelling the Excitation Function to Improve Quality in LPC's Resynthesis*. World Wide Web, <http://ccrma-www.stanford.edu/~aguiar/lpc.html>, 2000–2003.
- [2] Several Authors. *Wikipedia Pages*. World Wide Web, <http://www.wikipedia.org>, 2000–2003.
- [3] J. Andrew Fingerhut. *LPC-10 speech coder software*. World Wide Web, <http://www.arl.wustl.edu/~jaf/lpc/>, 1997–2001.
- [4] Wil Howitt. *Linear Predictive Coding (LPC) Tutorial*. World Wide Web, <http://asylum.sf.ca.us/pub/u/howitt/lpc.tutorial.html>, 2000–2003.
- [5] Nam Phamdo. *Speech Compression*. World Wide Web, <http://www.data-compression.com/speech.html>, 2000–2001.
- [6] Tony Robinson. *Speech Analysis*. World Wide Web, <http://svr-www.eng.cam.ac.uk/~ajr/SA95/SpeechAnalysis.html>, 1998–2003.
- [7] Rice University. *Speech Processing: Theory of LPC Analysis and Synthesis*. World Wide Web, <http://cnx.rice.edu/content/m10482/latest/>, 2000–2001.
- [8] Jean-Marc Valin. *Speech Compression*. World Wide Web, <http://www.data-compression.com/speech.html>, 2000–2003.

6 Apêndice 1: Como visualizar a onda de som de um ficheiro de audio no octave

Ao executar o comando:

```
cat /dev/dsp > teste1.au
```

Esta-se a gravar da placa de som para o ficheiro teste1.au raw audio data, ou seja, estamos a gravar um ficheiro audio mas sem o seu header normal, só as samples.

Por defeito quando se abre /dev/dsp o som vai em 8 bits, mono, 8KHz (é possível alterar a configuração da placa com a função `ioctl()`).

Para mais tarde comparar com o resultado da síntese, pode-se analisar o sinal de som gravado através do octave.

As samples ficam representadas no ficheiro de forma sequencial e são do tipo 'unsigned char'.

```
#include <stdio.h>

FILE *in_file;
char x;
char b;
short int y;

int main()
{
    in_file=fopen("record1.au", "rb");
    while (!feof(in_file))
    {
        x=fread((char *)&y, 1, 2, in_file);
        printf("%d\n",y);
    }
    return(0);
}
```

1. Código em C para ler o ficheiro raw audio

```
$ convert > log $\\
$ octave\\
octave:1> load log; plot(log)
```

7 Apêndice 2: Código

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

#include "vocoder.h"

char *outfile;

/*
 * read lpc param:
 * 1) read lpc coef from file into lpc_coef CALC_TYPE array
 * 2) return success or failure
 */
int read_lpc_param(int lpc_param, CALC_TYPE *lpc_coef, double * pitch)
{
    int len;

    /* 1) */
    if ((len = read(lpc_param, lpc_coef, (LPC_ORDER+1)*sizeof(CALC_TYPE))) ==
        -1) {
        perror("lpc_param read");
        exit(EXIT_FAILURE);
    }

    if ((len = read(lpc_param, pitch, 2*sizeof(double))) ==
        -1) {
        perror("lpc_param read");
        exit(EXIT_FAILURE);
    }

    if (len == 0)
        return 0;
    else
        /* read_lpc_param succeeds */
        return 1;
}

/*
 * speech decode :
 * Open the parameters file, capture the frames, and for each frame obtain
 * the sintetized signal, using the IIR filter.
 * 1) open file containing LPC parameters
 * 2) for each frame of LPC parameters:
 *     3) synthetize
 * 4) close file
 */
void speech_decode()
{
    int lpc_param;
    double pitch[2];
    CALC_TYPE lpc_coef[LPC_ORDER+1];
}
```

```

int world[FRAMESIZE];
int i;
int k = 0;
CALC_TYPE last_lpc_order_samples[LPC_ORDER];

for(i = 0; i < FRAMESIZE; i++)
    world[i] = -1;

/* 1) */
if ((lpc_param = open(outfile, O_RDONLY, 0)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

/* initialize stuff */
for (i = 0; i < LPC_ORDER+1; i++)
    lpc_coef[i] = 0;

for (i = 0; i < LPC_ORDER+1 ; i++) {
    last_lpc_order_samples[i] = 0;
}

pitch[0] = 0;
pitch[1] = 0;

/* 2) */
while(read_lpc_param(lpc_param, lpc_coef,pitch)) {
    /* 3) */
    sintese(lpc_coef, last_lpc_order_samples,pitch);
    k++;
}

/* 4) */
if (close(lpc_param) == -1) {
    perror("close lpc_param");
    exit(EXIT_FAILURE);
}
}

/* write_lpc_param: (for now, just the coef)
* 1) open file.
* 2) write coef's.
* 3) close file.
*/
void write_lpc_param(double * pitch, CALC_TYPE *lpc_coef, int teste)
{
    int nbytes;

    if ((nbytes = write(teste,lpc_coef,(LPC_ORDER+1)*sizeof(CALC_TYPE))) ==

```

```

-1)    {
        perror("write");
        exit(EXIT_FAILURE);
    }

    if ((nbytes = write(teste,pitch,2*sizeof(double))) ==
-1)    {
        perror("write");
        exit(EXIT_FAILURE);
    }
}

/* this needs a description */
int read_frame(FILE *audio_fd, CALC_TYPE *frame)
{
    int i = 0;

    while(!feof(audio_fd)) {
        int x = getc(audio_fd);
        unsigned char y = (unsigned char) x;
        /*printf("%d\n", y);*/

        frame[i] = (CALC_TYPE) ((y - 127) / (CALC_TYPE) 127);
        /*printf("%.14f\n", frame[i]);*/

        if(i++ == FRAMESIZE)
            /* go to next frame */
            return 1;
    }

    /* read ends */
    return 0;
}

/* Determination of the type of the frame signal: voiced / unvoiced
*/

CALC_TYPE ra(int k, CALC_TYPE * a){
    int i;
    CALC_TYPE ac;

    ac = 0;
    for(i = 0; i < (LPC_ORDER+1) - k; i++)
        ac += a[i] * a[i+k];

    return(ac);
}

short int find_peak(double * samples){
    short int i,j;
    short int peak = 0;

```

```

    j = 0;

    /* normalize the samples */
    for(i = 0; i < FRAMESIZE; i++) {
        samples[i] = samples[i] / samples[0];
        /*printf("%.14f\n",samples[i]);*/
    }

    /* find the peak between 3 - 15 ms / samples 24-120 */
    for(i = 24; i < 120; i++){
        if(samples[i] > peak){
            peak = samples[i];
            j = i;
        }
    }

    if(peak < 0.30)
        return(0);
    else{
        peak = j;
        return(peak);
    }
}

short int type_signal(double * frame, double * a, double * r){
    int n;
    int k;
    int i;
    short int peak;
    CALC_TYPE ac;
    CALC_TYPE samples[FRAMESIZE];

    for(i = 0; i < FRAMESIZE; i++)
        samples[i] = 0;

    for(n = 0; n < 160; n++){
        ac = 0;
        for(k = 0; k < LPC_ORDER+1; k++){
            ac += ra(k,a) * r[n-k];
        }
        samples[n] = ac;
    }

    peak = find_peak(samples);
    return(peak);
}

double find_minimum(CALC_TYPE * samples){
    int i;
    int j = 10;
    double min;

```

```

        min = 0;
        for(i = 10 ; i < 140; i++){
            if(samples[i] > min){
                min = samples[i];
                j = i;
            }
        }

        return(j);
    }

/* Pitch determination with AMDF function */

double amdf(CALC_TYPE * frame, CALC_TYPE * a){
    int i;
    double ac = 0;
    double pitch;
    CALC_TYPE samples[FRAMESIZE];

    for(i = 10; i < FRAMESIZE; i++){
        ac += frame[i] + frame[i-10];
        samples[i] = ((double)1/(double)FRAMESIZE) * ac;
    }

    pitch = find_minimum(samples);
    return(pitch);
}

float gain(CALC_TYPE * a, CALC_TYPE * r){
    int i;
    float g;
    float ac = 0;

    for(i = 1; i < LPC_ORDER+1; i++){
        ac += a[i] * r[i-1];
    }

    g = sqrt(r[0] + ac);

    return(g);
}

/*
* speech encode:
* 1) open device or wavfile.
* 2) for each frame of samples from file:
*     3) calculate autocorrelation.
*     4) calculate LPC parameters.
*     5) determine type of excitation (voiced/unvoiced)
*     6) calculate pitch
*     7) calculate gain
*     8) write them on a file.
*/

```

```

* 9) close device or wavfile.
*/
void speech_encode(char *file)
{
    CALC_TYPE frame[FRAMESIZE];

    CALC_TYPE autocorr[LPC_ORDER+1];
    CALC_TYPE lpc_coef[LPC_ORDER+1];
    CALC_TYPE ref_coef[LPC_ORDER+1];

    double a;
    double b;
    int i;
    float g;
    int teste;
    double pitch2;
    short int pitch;
    double energy = 0;
    double p[2];
    FILE *audio_fd;

    /* 1) */
    if ((audio_fd = fopen(file,"rb")) == NULL) {
        /* open of device/wavfile failed */
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    /* */
    if ((teste = open(outfile, O_APPEND| O_WRONLY | O_CREAT, S_IRWXU)) == -1)
    {
        /* error opening the device */
        perror("open outfile");
        exit(EXIT_FAILURE);
    }

    /* inicialize stuff */
    for (i = 0; i < FRAMESIZE; i++){
        frame[i] = 0;
    }

    for (i = 0; i < LPC_ORDER+1; i++) {
        autocorr[i] = 0;
        lpc_coef[i] = 0;
        ref_coef[i] = 0;
    }
    /* end of inicializations */

    /* 2) */
    while(read_frame(audio_fd, frame)) {

        /* 3) */
        autocorrelation(FRAMESIZE, frame, autocorr);
    }
}

```

```

        /* 4) */
        levinson_durbin(autocorr, lpc_coef, ref_coef);

        /* 5) & 6) */

        /* Pitch period by autocorrelation */
        pitch = type_signal(frame, lpc_coef, autocorr);

        /* Pitch period by AMDF */
        pitch2 = amdf(frame,lpc_coef);

        /* determination of silence */
        a = frame[0];
        b = frame[FRAMESIZE];
        energy = (double)160*(((b-a)*(b - a)) / (double)12);

        /* energy trashold: 0.01 */
        if(energy <= 0.01)
            p[0] = -1;
        else{
            p[0] = pitch;
            /*p[1] = sqrt(energy);*/
        }
        /* 7) */
        g = gain(autocorr, lpc_coef);
        p[1] = g;

        /* 8) */
        write_lpc_param(p,lpc_coef, teste);
    }

    /* */
    if (close(teste) == EOF) {
        perror("close");
        exit(EXIT_FAILURE);
    }

    /* 9) */
    if (fclose(audio_fd) == EOF) {
        perror("close audio_fd");
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stdout,"Usage: %s <raw .wav file> <output speech encoded>\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }

    outfile = argv[2];

```

```

        /* encode raw .wav file */
        speech_encode(argv[1]);

        /* decoder */
        speech_decode();

        return EXIT_SUCCESS;
    }

/* lpc.c */
#include "vocoder.h"

/*
 * autocorrelation
 * r(k) as described at: http://www.data-compression.com/speech.html
 */

void autocorrelation(int s_length, CALC_TYPE *s, CALC_TYPE *r)
{
    int i;
    int n;
    CALC_TYPE ac;

    for (i = 0; i < LPC_ORDER+1; i++)
    {
        ac = 0;
        for (n = 0; n < (s_length - i); n++)
            ac += s[n] * s[n + i];
        r[i] = ac;
    }
}

CALC_TYPE calc_alfa(int i, int j, CALC_TYPE *k)
{
    CALC_TYPE a1;
    CALC_TYPE a2;

    if (i == j)
        return k[i];
    else
    {
        a1 = calc_alfa(i - 1, j, k);
        a2 = calc_alfa(i - 1, i - j, k);

        return (a1 - (k[i] * a2));
    }
}

void levinson_durbin(CALC_TYPE *r, CALC_TYPE *a, CALC_TYPE *k)
{
    int i, j;

```

```

    CALC_TYPE e[LPC_ORDER];
    CALC_TYPE alfa[LPC_ORDER];
    CALC_TYPE sum = 0;

    if (r[0] == 0) {
        printf("r[0] can't be zero\n");
        exit(0);
    }

    e[0] = r[0];

    for (i = 0; i < LPC_ORDER; i++) alfa[i] = 0;

    k[0] = 1;
    alfa[0] = k[0];

    for (i = 1; i < LPC_ORDER+1; i++) {
        sum = 0;

        for (j = 1; j <= (i - 1); j++) {
            sum += calc_alfa(i-1, j, k) * r[i - j];
        }

        k[i] = (r[i] - sum) / e[i - 1];

        alfa[i] = k[i];

        e[i] = (1 - k[i] * k[i]) * e[i - 1];
    }

    for (i = 1; i < LPC_ORDER+1; i++) {
        a[i] = calc_alfa(LPC_ORDER, i, k) * (-1);
    }
    a[0] = 1;
}

/* sintese.c */
#include "vocoder.h"

/*
 * Pseudo random number generator
 */

#define MIDTAP 1
#define MAXTAP 4

int Rrandom (){
    int the_random;
    static short y[MAXTAP+1]={-21161, -8478, 30892,-10216, 16950};
    static int j=MIDTAP, k=MAXTAP;

    /* The following is a 16 bit 2's complement addition,

```

```

*   with overflow checking disabled
*/

    y[k] += y[j];

    if(y[k] > 32767) /* to make 16 bit rollover to -/+ */
        y[k] = -(32768 - (y[k] & 32767));
    if(y[k] < -32768)
        y[k] = y[k] & 32767;

    the_random = y[k];
    k--;
    if (k < 0) k = MAXTAP;
    j--;
    if (j < 0) j = MAXTAP;

    return(the_random);

}

void sintese(CALC_TYPE *a, CALC_TYPE *aux, CALC_TYPE * p)
{
    int i;
    int n;
    int j;
    double gain;
    CALC_TYPE ac = 0;
    CALC_TYPE acm = 0;
    CALC_TYPE y[FRAMESIZE];
    int pitch;
    double noise[FRAMESIZE];
    pitch = p[0];
    gain = p[1] * 0.1;

    acm = pitch;

    for(j = 0; j < FRAMESIZE; j++){
        noise[j] = Rrandom() % 3;
    }

    for (i = 0; i < FRAMESIZE; y[i] = 0, i++);
        y[i] = 0;

    /* 2) */
    for (n = 1; n < FRAMESIZE ; n++) {
        ac = 0;

        for (i = 1; i < LPC_ORDER + 1; i++) {
            if ((n - i) < 0)
                ac += a[i] * (-1) * aux[(LPC_ORDER - i)];
            else
                ac += a[i] * (-1) * y[n - i];
        }
    }
}

```

```

        /* voiced frame with Pitch = pitch */
        if(pitch != 0){
            if(n == pitch){
                y[n] = ac + gain * 1;
                pitch = pitch + acm;
            }
            else
                y[n] = ac + gain * 0;
        }
        /* silence vs noise */
        else{
            /* silence */
            if(pitch == -1)
                y[n] =ac + gain * 0;
            /* It is necessary to short the gain of the noise*/
            else
                y[n] =ac + gain * (noise[n] * 0.05);
        }
    }

    /* 3 */ for (i = FRAMESIZE - LPC_ORDER; i < FRAMESIZE; i++)
        aux[i - (FRAMESIZE - LPC_ORDER)] = y[i];

    /* 4) */
    for (i = 0; i < FRAMESIZE; i++)
        printf("%c", (unsigned char) ((y[i] * 127) + 127));
    /*printf("%d\n", (unsigned char) ((y[i] * 127) + 127));*/
}

/* vocoder.h */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define LPC_ORDER 10

#define FRAMESIZE 160

#define CALC_TYPE double

void autocorrelation(int s_length, CALC_TYPE *s, CALC_TYPE *r);
void levinson_durbin(CALC_TYPE *r, CALC_TYPE *a, CALC_TYPE *k);
CALC_TYPE calc_alfa(int i, int j, CALC_TYPE *k);

void sintese(CALC_TYPE * a, CALC_TYPE *last_lpc_order_samples, CALC_TYPE * pitch);

```

```

/* Makefile */
CC = gcc
CFLAGS = -Wall -ansi -pedantic

# Define according to your computer
# EXTRACFLAGS= -march=pentium3 -O3 -pipe -fomit-frame-pointer

# Source Files
SRC = vocoder.c sintese.c
BIN = vocoder

# Object Files
OBJ = lpc.o sintese.o

all:    vocoder

vocoder: sintese.o lpc.o vocoder.c
        $(CC) $(CFLAGS) $(EXTRACFLAGS) -lm -g $(OBJ) vocoder.c -o vocoder

lpc.o: vocoder.h lpc.c
        $(CC) $(CFLAGS) $(EXTRACFLAGS) -g -c lpc.c

sintese.o: vocoder.h sintese.c
        $(CC) $(CFLAGS) $(EXTRACFLAGS) -g -c sintese.c

clean:
        rm -f *~ $(BIN) $(OBJ)
        rm -f speech.enc

indent: *.c *.h
        indent -i8 -kr $?
        @rm -rf *~
        @touch indent

```