RECONHECIMENTO DE VOZ PARA O PORTUGUÊS BRASILEIRO

Aluno: Jan Krueger Siqueira
Orientador: Abraham Alcaim

Introdução

Uma das questões atuais de acessibilidade é a criação de sistemas capazes de reconhecer e eventualmente executar alguma ação a partir da voz humana. Entre suas inúmeras aplicações, podemos citar a interação de deficientes visuais com o computador e a operação de equipamentos quando um trabalhador não se encontra com as mãos disponíveis.

Embora já existam inúmeras pesquisas e produtos nessa área, poucos deles são compatíveis com o idioma português do Brasil.

Objetivos

Estudar os princípios e recursos do reconhecimento de voz. Preparar um banco de vozes para treinar o sistema de reconhecimento. Pesquisar em artigos a utilização e os resultados de diversos sistemas de reconhecimento. Configurar os parâmetros do reconhecedor. Testar o sistema.

Metodologia

Existem inúmeros parâmetros que caracterizam a voz humana: espectro de freqüência, número de cruzamentos com zero, análise do sinal e suas derivadas no domínio do tempo, coeficientes cepestrais, etc. Tais parâmetros podem ser usados, por exemplo, na compressão de voz para transmissões digitais ou para arquivos de áudio. Outra de suas aplicações ocorre em diversos sistemas de reconhecimento, entre eles o muito bem sucedido Modelo de Markov Escondido (Hidden Markov Model – HMM). Dessa forma, pensou-se na idéia do reconhecimento remoto a partir de voz obtida de celulares ou de voz sobre IP.

Existem diversos tipos de codificadores utilizados na telefonia móvel e IP atualmente. Para serem usados no sistema de reconhecimento, serão inicialmente testados os seguintes: AMR-NB, AMR-WB e G723.1. Todos eles possuem código fonte aberto, podendo ser compilados em diversos sistemas operacionais.

Após a leitura das especificações dos três codificadores, foi possível utilizá-los na preparação e conversão do banco de vozes. Já estava disponível nesse banco um total de 1000 frases, cada uma pronunciada por 100 locutores (50 homens e 50 mulheres), num total de 100.000 (cem mil) arquivos em formato “.wav” (mono, 16 bits, 16 kHz).

Adequar tais arquivos à entrada dos codificadores exige conhecimento da estrutura do formato “wav” e manipulação em massa de bytes. Já a conversão automática de todo o banco de vozes demanda acesso ao sistema de pastas e arquivos do sistema operacional, bem como a invocação em loop da execução dos codificadores (via prompt). Considerando tudo isso, a ferramenta de apoio que se mostrou mais apta e prática foi o MatLab.

De um modo geral, as modificações nos arquivos “wav” não são muito complexas: os codificadores AMR-NB e G723.1 exigem um arquivo amostrado a 8 kHz, fazendo-se necessária uma filtragem e um downsampling; o AMR-WB processa amostras de 14 bits e o AMR-NB processa com 13 bits, o que requer a anulação e aproximação de bits mais significativos; e o cabeçalho com os metadados é sempre retirado. Esses processamentos geram um arquivo “.inp”.A compilação dos codificadores também não toma muito tempo:

escritos na linguagem C, podem ser transformados em executáveis para Windows ou Unix via GCC. Utilizam-se então, como teste, alguns arquivos com os quais se converte do formato ".wav" para ".inp", e os codificadores e decodificadores para processar os arquivos resultantes. Finalmente, adapta-se o resultado de volta para ".wav". Comparando-se o som original com o final, pode-se verificar a funcionalidade do sistema.

Em seguida, a codificação em massa foi testada. Para organizar os dados, o banco de vozes foi catalogado em pastas separadas de acordo com o locutor. A programação garantiu que os arquivos convertidos fossem organizados do mesmo modo, porém num diretório diferente dos originais.

A primeira etapa do projeto foi então concluída. No momento, pesquisa-se em artigos os resultados e conclusões sobre esta linha de reconhecimento de voz envolvendo codificadores. Outro ponto a ser investigado é o funcionamento mais detalhado dos codificadores/decodificadores, já que a intenção é alterá-los para que forneçam os parâmetros mais adequados ao reconhecedor.

Próximos Passos

Utilizando o HTK (HMM Tool Kit), o reconhecedor de voz será ajustado para os fones e a gramática brasileira. O treinamento será realizado com os parâmetros de voz obtidos por cada um dos codificadores, na busca de saber qual fornece a melhor taxa de reconhecimento.

A expectativa é que o sistema seja capaz de identificar a fala contínua, independentemente da escolha do locutor.

Referências

1 – RABINER, L.; JUANG B. **Fundamentals of Speech Recognition**. New Jersey, USA: Prentice-Hall, c1993. 507p.

2 – RABINER, L. A tutorial on hidden Markov models and selected applications in speech recognition. **Proceedings of the IEEE**, v. 77, n. 2, pp. 257-286, fev. 1989.

3 – http://wiki.multimedia.cx/index.php?title=AMR-NB

4 – http://en.wikipedia.org/wiki/G.722.2

5 – http://www.vocal.com/data_sheets/g723.html

6 – http://htk.eng.cam.ac.uk/

Apêndice – Rotinas e Funções do MatLab

1 – Requisitos dos Codificadores

Os codificadores exigem que os arquivos de entrada estejam nos seguintes formatos:

Codificador	Extensão do Arquivo	Taxa de Amostragem	Tamanho da Amostra*	Quantidade N de amostras por bloco
AMR-NB	.INP	8 kHz	13 bits	160 amostras
AMR-WB	.INP	16 kHz	14 bits	320 amostras
G723.1	.TIN	8 kHz	16 bits	240 amostras

* OBS: complemento a 2 (isto é, o bit mais à direita é o bit do sinal)

2 – Estrutura dos Arquivos

Diretório Raiz:
AMR (pasta)
G723 (pasta)
AmostrarMetadeSinal.m
CalcularEspectroSinal.m
CarregarWav16Bits.m
CompararArquivos.m
CompletarGrupoNAmostras.m
ConverterTodosArquivosWavs.m
DumpArquivoBinario8Bits.m
ZerarBitsMenosSignificativos.m

Pasta AMR:
AMR-NB (pasta)
AMR-WB (pasta)
ConverterAMRParaWav.m
SalvarINP.m

Pasta AMR-NB:
ConverterTudoParaAMR_NB.m
ConverterWavParaAMR_NB.m
decoder.exe
encoder.exe

Pasta AMR-WB:
ConverterTudoParaAMR_WB.m
ConverterWavParaAMR_WB.m
decoder.exe
encoder.exe

Pasta G723:
ConverterG723ParaWav.m
ConverterTudoParaG723.m
ConverterWavParaG723.m
decoder.exe
encoder.exe
SalvarTIN.m

3 – Códigos dos Arquivos

Começamos com as funções básicas do diretório “raiz”.

CarregarWav16Bits: carrega um arquivo .wav com amostras de 16 bits, já retirando os metadados do cabeçalho. Cabe ressaltar que, neste momento, os valores armazenados do sinal estão normalizados – isto é, o valores de amplitude estão contidos no intervalo [–1 , 1].

```
function sinalWav_vet = CarregarWav16Bits ( nomeArquivo_str )

sinalWav_vet = wavread( nomeArquivo_str ) ;
sinalWav_vet = sinalWav_vet' ;
```

AmostrarMetadeSinal: alguns codificadores esperam arquivos amostrados a 8 kHz, enquanto nossa base de voz foi gravada a 16 kHz. Para evitar o efeito de falseamento (*aliasing* em inglês), aplicamos primeiramente um filtro passa-baixa no sinal, para então realizamos o *downsampling*.

```
function sinalAmostrado_vet = AmostrarMetadeSinal ( sinalOriginal_vet )

fatorAmostragem = 2 ; % amostra metade

ordemFiltro = 64 ;
coeficientesXFiltro_vet = fir1( ordemFiltro , 1 / fatorAmostragem ) ;
coeficientesYFiltro_vet = 1 ;
sinalAmostrado_vet = filter( coeficientesXFiltro_vet , ...
                             coeficientesYFiltro_vet , sinalOriginal_vet ) ;

sinalAmostrado_vet = sinalAmostrado_vet ( 1 : fatorAmostragem : ...
                                          length( sinalAmostrado_vet ) ) ;
```

ZerarBitsMenosSignificativos: alguns codificadores trabalham com conjuntos de bits menores que 16, solicitando assim que os últimos bits sejam zerados. Neste momento, desnormalizamos os dados de volta para inteiros de 16 bits, mas de um modo que provoque a zeragem dos bits desejados. Uma precaução extra é tomada no fim para que nenhuma amostra extrapole o intervalo esperado, devido a possíveis aproximações de cálculo do computador.

```
function retorno_vet = ZerarBitsMenosSignificativos ( sinalOriginal_vet ...
                                                     , qtdBitsZerados )

retorno_vet = sinalOriginal_vet ;

potenciaDe2 = 2 ^ ( 15 - qtdBitsZerados ) ;
retorno_vet = floor( retorno_vet * potenciaDe2 ) ;

potenciaDe2 = 2 ^ qtdBitsZerados ;
retorno_vet = retorno_vet * potenciaDe2 ;

% precaução extra
    limite = floor( ( 2 ^ 15  - 1 ) / ( 2 ^ qtdBitsZerados ) ) * ...
             ( 2 ^ qtdBitsZerados ) ;

    indicesElementosRuins_vet = find( retorno_vet >= limite ) ;
    retorno_vet( indicesElementosRuins_vet ) = limite - 1 ;
    indicesElementosRuins_vet = find( retorno_vet < -limite ) ;
    retorno_vet( indicesElementosRuins_vet ) = -limite ;
```

CompletarGrupoNAmostras: os codificadores trabalham com blocos de N amostras. Caso o total de amostras não seja múltiplo de N, as últimas serão descartadas. A fim de evitar tal perda, acrescentamos amostras nulas (silêncio no sinal) até que o total seja divisível por N.

```
function retorno_vet = CompletarGrupoNAmostras ( sinalOriginal_vet , ...
                                                 qtdGrupoNAmostras )

qtdAmostras = length( sinalOriginal_vet ) ;
restoDivisao = mod( qtdAmostras , qtdGrupoNAmostras ) ;

if restoDivisao > 0
    qtdAmostrasQueFaltam = qtdGrupoNAmostras - restoDivisao ;
    elementosParaCompletarN_vet = zeros( 1 , qtdAmostrasQueFaltam ) ;
    retorno_vet = [ sinalOriginal_vet , elementosParaCompletarN_vet ] ;
else
    retorno_vet = sinalOriginal_vet ;
end
```

Passaremos agora para as funções específicas do AMR.

SalvarINP: recebe o sinal já tratado pelas funções anteriores, acrescenta algumas amostras de sincronismo iniciais (especificação do codificador, relativo ao início da transmissão de informação) e salva o conjunto num arquivo de prefixo “.INP”.

```
function SalvarINP ( sinalWav_vet , nomeArquivo_str , qtdAmostrasSincronismo )

prefixo_str = nomeArquivo_str( length( nomeArquivo_str ) - 3 : ...
                               length( nomeArquivo_str ) ) ;
if prefixo_str ~= '.INP'
    nomeArquivo_str = [ nomeArquivo_str , '.INP' ] ;
end

amostrasSincronismo_vet = ones( 1 , qtdAmostrasSincronismo ) * 8 ;
sinalWav_vet = [ amostrasSincronismo_vet , sinalWav_vet ] ;

teste_fp = fopen( nomeArquivo_str , 'w' ) ;
```

```
fwrite( teste_fp , sinalWav_vet , 'int16' ) ;
fclose( teste_fp ) ;
```

ConverterTodosArquivosWavs: dada uma função de conversão de um codificador, aplica-se ela a todos os arquivos .wav das pastas de um diretório.

```
function ConverterTodosArquivosWavs ( diretorioPastasWavs_str , ...
                                      diretorioDestino_str , ...
                                      funcaoConversao_func , ...
                                      sobreescreverArquivo )

pastasWavs_struct_vet = dir( diretorioPastasWavs_str ) ;

for i = 3 : length( pastasWavs_struct_vet )
    if pastasWavs_struct_vet( i ).isdir
        nomePastaWav_str = pastasWavs_struct_vet( i ).name ;
        caminhoPastaWav_str = [ diretorioPastasWavs_str , '\\' , ...
                                nomePastaWav_str ] ;
        arquivosWav_struct_vet = dir( caminhoPastaWav_str ) ;

        for j = 3 : length( arquivosWav_struct_vet )
            nomeArquivoWav_str = arquivosWav_struct_vet( j ).name ;
            caminhoArquivoWav_str = [ caminhoPastaWav_str , '\\' , ...
                                      nomeArquivoWav_str ] ;
            tamanhoNomeArquivo = length( nomeArquivoWav_str ) ;

            ultimos4Indices = ( tamanhoNomeArquivo - 3 ) : ...
                              tamanhoNomeArquivo ;
            ultimos4Caracteres_str = nomeArquivoWav_str( ultimos4Indices ) ;

            if ultimos4Caracteres_str == '.wav'
                warning off ; % evita alertas do mkdir
                mkdir( diretorioDestino_str , nomePastaWav_str ) ;
                warning on ;

                nomeArquivoSemSufixo_str = nomeArquivoWav_str ;
                nomeArquivoSemSufixo_str( ultimos4Indices ) = [ ] ;

                nomeArquivoSaida_str = nomeArquivoSemSufixo_str ;
                caminhoArquivoSaida_str = [ diretorioDestino_str , '\\' , ...
                                            nomePastaWav_str , '\\' , ...
                                            nomeArquivoSaida_str ] ;
                arquivoSaida_fp = fopen( caminhoArquivoSaida_str , 'r' ) ;

                if sobreescreverArquivo | arquivoSaida_fp == -1
                    funcaoConversao_func( caminhoArquivoWav_str , ...
                                          caminhoArquivoSaida_str ) ;
                end

                if arquivoSaida_fp ~= -1
                    fclose( arquivoSaida_fp ) ;
                end

            end
        end
    end
end
```

ConverterWavParaAMR_NB: chama as funções de preparação do sinal e, em seguida, invoca o codificador AMR-NB via prompt do sistema operacional. Em modo DEBUG, verifica a corretude do arquivo codificado chamando o decodificador.

```
function ConverterWavParaAMR_NB ( nomeArquivoWav_str , nomeArquivoINP_str )

% constantes
```

```
    DEBUG = 1 ; % true
    CODIFICADOR_str = 'encoder' ;
    DECODIFICADOR_str = 'decoder' ;
    TAXA_TRANSMISSAO_str = 'MR122' ;
    QTD_AMOSTRAS_POR_QUADRO = 160 ;
    QTD_AMOSTRAS_SINCRONISMO = QTD_AMOSTRAS_POR_QUADRO * 320 ;
    TAXA_AMOSTRAGEM = 8000 ;
    QTD_BITS_ZERADOS = 3 ;

% ajuste do sinal wav para as especificacoes do codificador AMR-NB
    sinalWav_vet = CarregarWav16Bits( nomeArquivoWav_str ) ;

    sinalWav_vet = AmostrarMetadeSinal( sinalWav_vet ) ;
    sinalWav_vet = ZerarBitsMenosSignificativos( sinalWav_vet , ...
                                                 QTD_BITS_ZERADOS ) ;

    sinalWav_vet = CompletarGrupoNAmostras( sinalWav_vet , ...
                                            QTD_AMOSTRAS_POR_QUADRO ) ;

    if DEBUG
        max( sinalWav_vet )
        min( sinalWav_vet )
    end

% salvando formato de entrada para o codificador
    SalvarINP( sinalWav_vet , nomeArquivoINP_str , QTD_AMOSTRAS_SINCRONISMO ) ;

% o arquivo de saida tera o mesmo nome, mas com sufixo .COD
    prefixo_str = nomeArquivoINP_str( length( nomeArquivoINP_str ) - 3 : ...
                                      length( nomeArquivoINP_str ) ) ;
    if prefixo_str ~= '.INP'
        nomeArquivoINP_str = [ nomeArquivoINP_str , '.INP' ] ;
    end

    nomeArquivoCOD_str = [ nomeArquivoINP_str , '.COD' ] ;

% chamando o codificador
    comandoPrompt_str = [ CODIFICADOR_str , ' ' , TAXA_TRANSMISSAO_str , ...
                          ' ' , nomeArquivoINP_str , ' ' , nomeArquivoCOD_str ] ;
    eval( [ '! ' , comandoPrompt_str ] ) ;

if DEBUG
    % chamando o decodificador
        nomeArquivoOUT_str = [ nomeArquivoCOD_str , '.OUT' ] ;
        comandoPrompt_str = [ DECODIFICADOR_str , ' ' , ...
                              nomeArquivoCOD_str , ' ' , ...
                              nomeArquivoOUT_str ] ;
        eval( [ '! ' , comandoPrompt_str ] ) ;

    % convertendo para wav
        nomeArquivoWav_str = [ nomeArquivoOUT_str , '.wav' ] ;
        ConverterAMRParaWav( nomeArquivoOUT_str , nomeArquivoWav_str , ...
                             QTD_AMOSTRAS_SINCRONISMO , TAXA_AMOSTRAGEM ) ;
end
```

ConverterWavParaAMR_WB: similar à função anterior, mas considerando os parâmetros do codificador AMR-WB.

```
function ConverterWavParaAMR_WB ( nomeArquivoWav_str , nomeArquivoINP_str )

% constantes
    DEBUG = 1 ; % true
    CODIFICADOR_str = 'encoder' ;
    DECODIFICADOR_str = 'decoder' ;
    TAXA_TRANSMISSAO_str = '2' ;  % relativo a taxa 12.65 kbit/s
```

```
    QTD_AMOSTRAS_POR_QUADRO = 320 ;
    QTD_AMOSTRAS_SINCRONISMO = 2 * QTD_AMOSTRAS_POR_QUADRO ;
    TAXA_AMOSTRAGEM = 16000 ;
    QTD_BITS_ZERADOS = 2 ;

% ajuste do sinal wav para as especificacoes do codificador AMR-NB
    sinalWav_vet = CarregarWav16Bits( nomeArquivoWav_str ) ;

    sinalWav_vet = ZerarBitsMenosSignificativos( sinalWav_vet , ...
                                                 QTD_BITS_ZERADOS ) ;
    sinalWav_vet = CompletarGrupoNAmostras( sinalWav_vet , ...
                                            QTD_AMOSTRAS_POR_QUADRO ) ;

    if DEBUG
        max( sinalWav_vet )
        min( sinalWav_vet )
    end

% salvando formato de entrada para o codificador
    SalvarINP( sinalWav_vet , nomeArquivoINP_str , QTD_AMOSTRAS_SINCRONISMO ) ;

% o arquivo de saida tera o mesmo nome, mas com sufixo .COD
    prefixo_str = nomeArquivoINP_str( length( nomeArquivoINP_str ) - 3 : ...
                                      length( nomeArquivoINP_str ) ) ;
    if prefixo_str ~= '.INP'
        nomeArquivoINP_str = [ nomeArquivoINP_str , '.INP' ] ;
    end

    nomeArquivoCOD_str = [ nomeArquivoINP_str , '.COD' ] ;

% chamando o codificador
    comandoPrompt_str = [ CODIFICADOR_str , ' ' , TAXA_TRANSMISSAO_str , ...
                          ' ' , nomeArquivoINP_str , ' ' , nomeArquivoCOD_str ] ;
    eval( [ '! ' , comandoPrompt_str ] ) ;

if DEBUG
    % chamando o decodificador
        nomeArquivoOUT_str = [ nomeArquivoCOD_str , '.OUT' ] ;
        comandoPrompt_str = [ DECODIFICADOR_str , ' ' , ...
                              nomeArquivoCOD_str , ' ' , ...
                              nomeArquivoOUT_str ] ;
        eval( [ '! ' , comandoPrompt_str ] ) ;

    % convertendo para wav
        nomeArquivoWav_str = [ nomeArquivoOUT_str , '.wav' ] ;
        ConverterAMRParaWav( nomeArquivoOUT_str , nomeArquivoWav_str , ...
                             QTD_AMOSTRAS_SINCRONISMO , TAXA_AMOSTRAGEM ) ;
end
```

ConverterTudoParaAMR_NB: rotina que solicita conversão para AMR-NB de todos os arquivos .wav das pastas de um diretório.

```
% limpa tela, variáveis e gráficos
    clc ;
    clear ;
    close all ;

% avisa que temos funçõees nas pastas acima
    addpath '../' ;
    addpath '../..' ;

% converte arquivos
    pastaEntrada_str = '../../../TesteArquivosEntrada' ;
    pastaSaida_str = 'TesteArquivosSaida' ;
    funcaoConversao_func = @ConverterWavParaAMR_NB ;
    sobreescreverArquivos = 1 ;
```

```
    ConverterTodosArquivosWavs( pastaEntrada_str , pastaSaida_str , ...
                                funcaoConversao_func , ...
                                sobreescreverArquivos ) ;
```

ConverterTudoParaAMR_NB: similar à rotina anterior, mas relativa ao outro codificador AMR.

```
% limpa tela, variáveis e gráficos
    clc ;
    clear ;
    close all ;

% avisa que temos funções nas pastas acima
    addpath '../' ;
    addpath '../..' ;

% converte arquivos
    pastaEntrada_str = '../../../TesteArquivosEntrada' ;
    pastaSaida_str = 'TesteArquivosSaida' ;
    funcaoConversao_func = @ConverterWavParaAMR_WB ;
    sobreescreverArquivos = 1 ;

    ConverterTodosArquivosWavs( pastaEntrada_str , pastaSaida_str , ...
                                funcaoConversao_func , ...
                                sobreescreverArquivos ) ;
```

Por fim, seguem as funções específicas do G723.1.
SalvarTIN: similar à função SalvarINP.

```
function SalvarTIN ( sinalWav_vet , nomeArquivo_str )

prefixo_str = nomeArquivo_str( length( nomeArquivo_str ) - 3 : ...
                               length( nomeArquivo_str ) ) ;
if prefixo_str ~= '.TIN'
    nomeArquivo_str = [ nomeArquivo_str , '.TIN' ] ;
end

tin_fp = fopen( nomeArquivo_str , 'w' ) ;
fwrite( tin_fp , sinalWav_vet , 'int16' ) ;
fclose( tin_fp ) ;
```

ConverterWavParaG723: similar à função ConverterTudoParaAMR_NB.m.

```
function ConverterWavParaG723 ( nomeArquivoWav_str , nomeArquivoTIN_str )

% constantes
    DEBUG = 1 ; % true
    CODIFICADOR_str = 'encoder' ;
    DECODIFICADOR_str = 'decoder' ;
    PARAMETROS_CODIFICADOR_str = '-c -r63 -v' ;
    PARAMETROS_DECODIFICADOR_str = '-d' ;
    QTD_AMOSTRAS_POR_QUADRO = 240 ;
    TAXA_AMOSTRAGEM = 8000 ;
    QTD_BITS_ZERADOS = 0 ;

% ajuste do sinal wav para as especificacoes do codificador AMR-NB
    sinalWav_vet = CarregarWav16Bits( nomeArquivoWav_str ) ;

    sinalWav_vet = AmostrarMetadeSinal( sinalWav_vet ) ;
    sinalWav_vet = ZerarBitsMenosSignificativos( sinalWav_vet , ...
                                                 QTD_BITS_ZERADOS ) ;
```

```
    sinalWav_vet = CompletarGrupoNAmostras( sinalWav_vet , QTD_AMOSTRAS_POR_QUADRO
) ;

    if DEBUG
        max( sinalWav_vet )
        min( sinalWav_vet )
    end

% salvando formato de entrada para o codificador
    SalvarTIN( sinalWav_vet , nomeArquivoTIN_str ) ;

% o arquivo de saida tera o mesmo nome, mas com sufixo .COD
    prefixo_str = nomeArquivoTIN_str( length( nomeArquivoTIN_str ) - 3 : ...
                                      length( nomeArquivoTIN_str ) ) ;
    if prefixo_str ~= '.TIN'
        nomeArquivoTIN_str = [ nomeArquivoTIN_str , '.TIN' ] ;
    end

    nomeArquivoCOD_str = [ nomeArquivoTIN_str , '.XCO' ] ;

% chamando o codificador
    comandoPrompt_str = [ CODIFICADOR_str , ' ' , ...
                          PARAMETROS_CODIFICADOR_str , ' ' , ...
                          nomeArquivoTIN_str , ' ' , nomeArquivoCOD_str ] ;
    eval( [ '! ' , comandoPrompt_str ] ) ;

if DEBUG
    % chamando o decodificador
        nomeArquivoOUT_str = [ nomeArquivoCOD_str , '.XOU' ] ;
        comandoPrompt_str = [ DECODIFICADOR_str , ' ' , ...
                              PARAMETROS_DECODIFICADOR_str , ' ' , ...
                              nomeArquivoCOD_str , ' ' , ...
                              nomeArquivoOUT_str ] ;
        eval( [ '! ' , comandoPrompt_str ] ) ;

    % convertendo para wav
        nomeArquivoWav_str = [ nomeArquivoOUT_str , '.wav' ] ;
        ConverterG723ParaWav( nomeArquivoOUT_str , nomeArquivoWav_str ) ;
end
```

ConverterTudoParaG723: similar à rotina ConverterTudoParaAMR_NB.m.

```
% limpa tela, variáveis e gráficos
    clc ;
    clear ;
    close all ;

% avisa que temos funções nas pastas acima
    addpath '../' ;
    addpath '../..' ;

% converte arquivos
    pastaEntrada_str = '../../TesteArquivosEntrada' ;
    pastaSaida_str = 'TesteArquivosSaida' ;
    funcaoConversao_func = @ConverterWavParaG723 ;
    sobreescreverArquivos = 1 ;

    ConverterTodosArquivosWavs( pastaEntrada_str , pastaSaida_str , ...
                                funcaoConversao_func , ...
                                sobreescreverArquivos ) ;
```

Outras funções auxiliares foram desenvolvidas, visando verificações e testes das funções anteriores.

CalcularEspectroSinal: retorna os valores de freqüência e módulo do espectro de um sinal, já facilitando sua plotagem.

```
function [ valoresEspectro_vet , frequencias_vet ] = ...
          CalcularEspectroSinal ( sinalY_vet , taxaAmostragemHz )

%qtdPontos = 512 ;
qtdPontos = length( sinalY_vet ) ;
if mod( qtdPontos , 2 ) > 0
    qtdPontos = qtdPontos - 1 ;
end

transformadaFourier_vet = fft( sinalY_vet , qtdPontos ) ;
valoresEspectro_vet = transformadaFourier_vet .* ...
                      conj( transformadaFourier_vet ) / qtdPontos ;
valoresEspectro_vet = valoresEspectro_vet( 1 : ( qtdPontos / 2  + 1 ) ) ;
frequencias_vet = taxaAmostragemHz * ( 0 : ( qtdPontos / 2 ) ) / qtdPontos ;
```

DumpArquivoBinario8Bits: cria um arquivo texto que descreve os bytes de um arquivo binário, de 8 em 8 bits, facilitando sua análise e verificação.

```
function DumpArquivoBinario8Bits ( nomeArquivoBinario_str , nomeArquivoTexto_str )

arquivoBinario_fp = fopen( nomeArquivoBinario_str , 'r' ) ;
if arquivoBinario_fp == -1
    error( [ '*** ERRO *** Não foi possível abrir o arquivo ' ...
             nomeArquivoBinario_str '!' ] ) ;
end
% salva bytes do arquivo num vetor coluna (elementos são do tipo inteiro de
% 0 a 255)
    dadosBinario_vet = fread( arquivoBinario_fp ) ;
fclose( arquivoBinario_fp ) ;

binarios_str_matriz = dec2bin( dadosBinario_vet , 8 ) ;          % strings com os
grupos de 8 bits
hexas_str_matriz = dec2hex( dadosBinario_vet , 2 ) ;             % strings com os
grupos de 2 algarismos em hexa que representam 8 bits
texto_str_matriz = [ binarios_str_matriz , hexas_str_matriz ]' ; % imprimiremos no
arquivo as duas representações

arquivoTexto_fp = fopen( nomeArquivoTexto_str , 'wt' ) ;
fprintf( arquivoTexto_fp , '%c%c%c%c%c%c%c%c  -  %c%c\n%c%c%c%c%c%c%c%c  -
%c%c_\n%c%c%c%c%c%c%c%c  -  %c%c\n%c%c%c%c%c%c%c%c  -  %c%c____\n' ,
texto_str_matriz ) ;
fclose( arquivoTexto_fp ) ;
```

CompararArquivos: verifica se dois arquivos são idênticos em todos os bits. Em modo DEBUG, podemos averiguar o percentual de diferença. Útil para verificar se os codificadores funcionaram conforme o esperado, comparando a saída obtida com aquela fornecida em seu pacote de arquivos.

```
function saoIguais = CompararArquivos ( caminhoArquivo1_str , ...
                                        caminhoArquivo2_str )

DEBUG = 0 ;

% checa parametros
    if nargin ~= 2
        error( [ '*** ERRO *** São necessários 2 argumentos, cada um ' ...
                 'um caminho de arquivo numa String!' ] ) ;
    end

% abre primeiro arquivo
    arquivo1_fp = fopen( caminhoArquivo1_str , 'r' ) ;

    if arquivo1_fp == -1
```

```
        error( [ '*** ERRO *** Não foi possível abrir o arquivo ' , ...
                 caminhoArquivo1_str ] ) ;
    end

    % gera vetor cujos elementos sao os bytes do arquivo 1
        dadosArquivo1_vet = fread( arquivo1_fp ) ;

% abre segundo arquivo
    arquivo2_fp = fopen( caminhoArquivo2_str , 'r' ) ;

    if arquivo2_fp == -1
        error( [ '*** ERRO *** Não foi possível abrir o arquivo ' , ...
                 caminhoArquivo2_str ] ) ;
    end

    % gera vetor cujos elementos sao os bytes do arquivo 1
        dadosArquivo2_vet = fread( arquivo2_fp ) ;

% verifica se os arquivos sao iguais
    if length( dadosArquivo1_vet ) ~= length( dadosArquivo1_vet )
        saoIguais = 0 ;
        if DEBUG
            disp( [ 'Os arquivos ' caminhoArquivo1_str ' e ' ...
                    caminhoArquivo2_str ' são de tamanhos diferentes.' ] ) ;
        end
    elseif dadosArquivo1_vet == dadosArquivo2_vet
        saoIguais = 1 ;
    else
        saoIguais = 0 ;
        if DEBUG
            indicesDiferentes = find( dadosArquivo1_vet ~= dadosArquivo2_vet ) ;
            percetualDiferenca = length( indicesDiferentes ) / ...
                                 length( dadosArquivo1_vet ) * 100 ;
            disp( [ 'Os arquivos ' caminhoArquivo1_str ' e ' ...
                    caminhoArquivo2_str ' são do mesmo tamanho e com ' ...
                    num2str( percetualDiferenca , 4 ) '% de diferenca.' ] ) ;
        end
    end
```

ConverterG723ParaWav: usado após a decodificação de teste (modo DEBUG). Comparamos (auditivamente) o arquivo wav resultante com o original e verificamos então se as perdas no processo foram significativas.

```
function ConverterG723ParaWav ( nomeArquivoTIN_str , nomeArquivoWav_str )

% constantes
    TAXA_AMOSTRAGEM = 8000 ;

% carregando arquivo TIN (binário)
    arquivoBinario_fp = fopen( nomeArquivoTIN_str , 'r' ) ;
    if arquivoBinario_fp == -1
        error( [ '*** ERRO *** Não foi possível abrir o arquivo ' ...
                 nomeArquivoTIN_str '!' ] ) ;
    end
    % salva bytes do arquivo num vetor coluna (elementos são do tipo inteiro de
    %   inteiro de 0 a 2^16 - 1)
        dadosINP_vet = fread( arquivoBinario_fp , inf , 'int16' ) ;
    fclose( arquivoBinario_fp ) ;

% normaliza e salva em .wav
    dadosINP_vet = dadosINP_vet / ( 2 ^15 ) ;
    wavwrite( dadosINP_vet , TAXA_AMOSTRAGEM , 16 , nomeArquivoWav_str ) ;
```

ConverterAMRParaWav: similar à função anterior, porém relativa ao sistema AMR.

```
function ConverterAMRParaWav ( nomeArquivoINP_str , nomeArquivoWav_str , ...
                                qtdAmostrasSincronismo , taxaAmostragem )

% carregando arquivo INP (binário)
    arquivoBinario_fp = fopen( nomeArquivoINP_str , 'r' ) ;
    if arquivoBinario_fp == -1
        error( [ '*** ERRO *** Não foi possível abrir o arquivo ' ...
                 nomeArquivoBinario_str '!' ] ) ;
    end
    % salva bytes do arquivo num vetor coluna (elementos são do tipo inteiro de
    %   inteiro de 0 a 2^16 - 1)
        dadosINP_vet = fread( arquivoBinario_fp , inf , 'int16' ) ;
    fclose( arquivoBinario_fp ) ;

% joga fora as amostras de sincronismo
    dadosINP_vet( 1 : qtdAmostrasSincronismo ) = [ ] ;

% normaliza e salva em .wav
    dadosINP_vet = dadosINP_vet / ( 2 ^15 ) ;
    wavwrite( dadosINP_vet , taxaAmostragem , 16 , nomeArquivoWav_str ) ;
```