Decodificador digital de tons DTMF

Autor: Junio Cesar Ferreira

Data: 14/03/2018; Revisado em 11/11/2018

Resumo:
Este pequeno tutorial tem como objetivo apresentar duas alternativas de algoritmos para decodificar tons do sistema DTMF (*Dual Tone Mult Frequency*), sendo estes, a transformada rápida de Fourier (FFT – *Fast Fourier Transform*) baseada no algoritmo de Cooley-Tukey para vetores com comprimento de base 2, e o algoritmo de Goertzel para cômputo de componentes especificas do espectro de frequências. Apresenta-se o desenvolvimento dos algoritmos e sua implementação em linguagem C. Concluindo com uma breve comparação de desempenho entre as duas alternativas e a apresentação de um software em C# de exemplo didático.

Introdução

O sistema de tons do teclado telefônico é conhecido por sistema DTMF (*Dual Tone Mult Frequency signaling*), este sistema foi introduzido com a chegada dos telefones com teclado, para substituir o antigo sistema baseado no disco de pulsos. No início, utilizava-se filtros analógicos do tipo passa-faixa para detectar os tons DTMF, com o avanço da eletrônica digital, esta filtragem passou a ser realizada em nível computacional. Atualmente, novas aplicações surgem no âmbito da telefonia digital para o sistema DTMF, tais como serviços de URA (Unidade de resposta Audível), onde são transmitidos dados em um canal de voz. Neste contesto, é interessante para engenheiros e técnicos que trabalham com processamento digital de sinais, compreender o funcionamento e as técnicas envolvidos na decodificação dos tons DTMF.

Neste tutorial, pretende-se apresentar de modo claro e sucinto, alguns dos conceitos e ferramentas envolvidos no processamento de sinais, embora seja apresentado como aplicação à decodificação dos tons DTMF, as técnicas abordadas podem ser aplicadas em diversas outras áreas, em que seja necessária a decomposição espectral de um sinal do domínio do tempo ou do espaço para o domínio da frequência.

Nos tópicos a seguir, apresenta-se os princípios de funcionamento do sistema DTMF, a transformada de Fourier como ferramenta matemática para análise em tempo contínuo, a transformada discreta de Fourier para aplicações computacionais, os algoritmos para cômputo eficiente da transformada discreta e suas implementações em linguagem C. Por fim, apresenta-se um software desenvolvido em C# para exemplificar e simular a aplicação dos algoritmos, apresentando ainda uma sugestão para uso em sistemas microcontrolados.

Observação. Em todo o texto, utilizou-se as seguintes notações para os conjunto numéricos: $\mathbb{Z}$ representa o conjunto dos inteiros, $\mathbb{N}$ o conjunto dos naturais iniciando pelo 1, $\mathbb{R}$ o conjunto dos números reais e $\mathbb{C}$ o conjunto dos números complexos.

1. Sistema DTMF

Conforme mencionado na introdução, o sistema DTMF foi desenvolvido para transmitir os números discados em transmissões telefônicas, para compreender melhor, veja a figura 1, onde temos a matriz dos principais dígitos utilizados (pois em algumas versões acrescenta-se mais uma coluna com a frequência de 1633 Hz). Para cada digito duas frequências na faixa de áudio são somadas. A figura 2, apresenta a multiplexação para geração do sinal. Por exemplo, quando a tecla '4' é pressionada, soma-se dois sinais cuja as frequências são de 770 Hz e 1209 Hz. Em geral estes sinais são gerados por osciladores senoidais.

Hz	1209	1336	1477
697	1	2	3
770	4	5	6
852	7	8	9
941	*	0	#

Figura 1 – Teclado DTMF

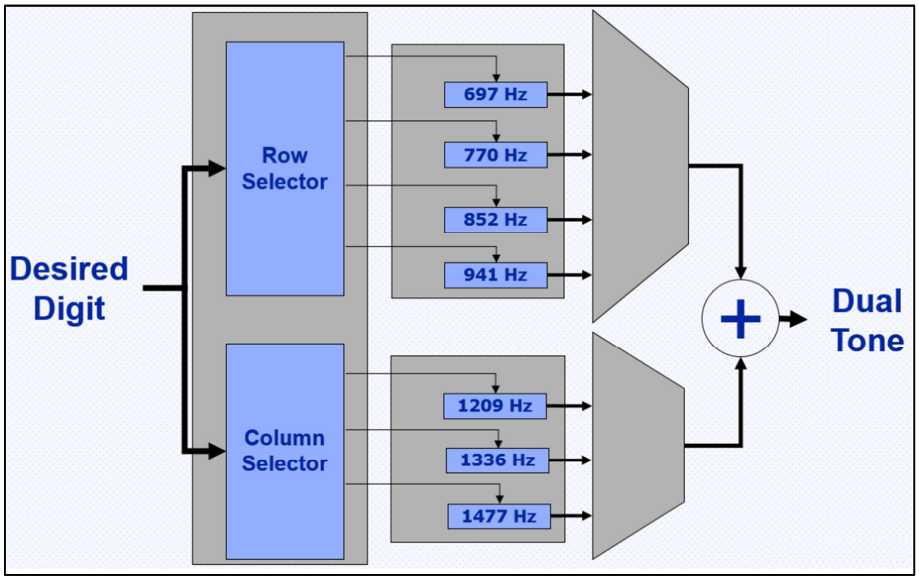


Figura 2 – Geração do Sinal

É natural que exista um padrão para a duração de cada tom, a figura 3 exibe um exemplo ilustrativo de quatro tons sendo transmitidos, onde em média cada tom dura 40 milissegundos, e o mesmo valor é usado para os intervalos entre tons. Para informações técnicas mais detalhadas recomenda-se a leitura de [9].

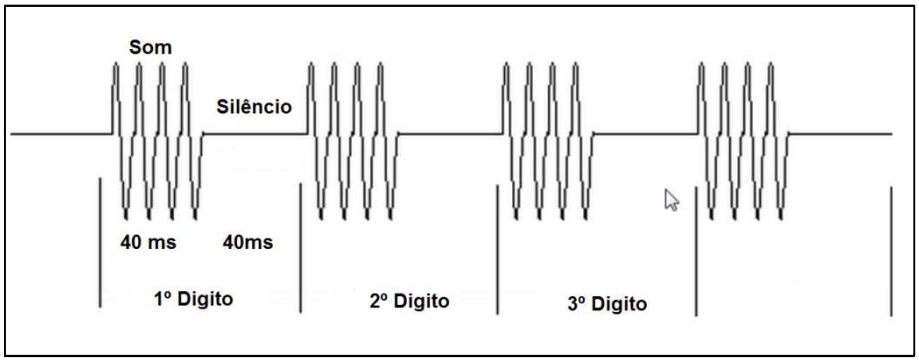


Figura 3 – Padrão DTMF, tempo dos tons.

Nas seções a seguir, apresentam-se todo o ferramental matemático e computacional, necessários à compreensão das técnicas utilizadas para filtrar e decodificar os tons DTMF em um dispositivo digital, tal como um microcontrolador.

2. Transformada de Fourier

A transformada de Fourier é uma ferramenta matemática extremamente importante em processamento digital de sinais, pois, dado um sinal no domínio do tempo, representado pela função $f(t)$, então a transformada de Fourier dada por,

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

onde e é a constante de Napier ou número de Euler, i é a unidade imaginária ($i^2 = -1$) e $F(\omega)$ é uma função complexa que representa o espectro de $f(t)$, ou seja, F é a representação de f no domínio da frequência. Como f depende de t sendo t a variável tempo dada em segundos (SI), logo, F depende de ω que é a frequência angular dada em radianos por segundo. Note que, em geral, $f: \mathbb{R} \to \mathbb{R}$, pois f representa algum modelo físico realizável, enquanto $F: \mathbb{R} \to \mathbb{C}$, daí podemos escrever

$$F(\omega) = A(\omega) e^{i\theta(\omega)}$$

onde $A: \mathbb{R} \to \mathbb{R}_+$ é a função amplitude e $\theta: \mathbb{R} \to [0, 2\pi)$ representa a fase do sinal. Em muitas aplicações estamos interessados apenas em $A(\omega) = |F(\omega)|$, ou seja, o espectro de amplitudes.

Exemplo 1:
Calculemos o espectro da função retangular, definida por:

$$f(t) = \begin{cases} A, & |t| < \dfrac{d}{2} \\ \dfrac{A}{2}, & |t| = \dfrac{d}{2} \\ 0, & |t| > \dfrac{d}{2} \end{cases} \quad \text{com } A, d \in \mathbb{R}$$

Pode-se calcular a transformada de Fourier da seguinte forma

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt = A \int_{-d}^{d} e^{-i\omega t} dt = A \frac{e^{i\omega d} - e^{-i\omega d}}{i\omega}$$

$$\therefore\ F(\omega) = \frac{2A}{\omega} sen\,(\omega d) = 2Ad.sinc(\omega d)$$

O gráfico superior da figura 4 é representação gráfica de $f(t)$, enquanto o gráfico inferior representa o espectro de amplitudes $|F(\omega)|$. Observe que neste caso particular, o espectro é puramente real.

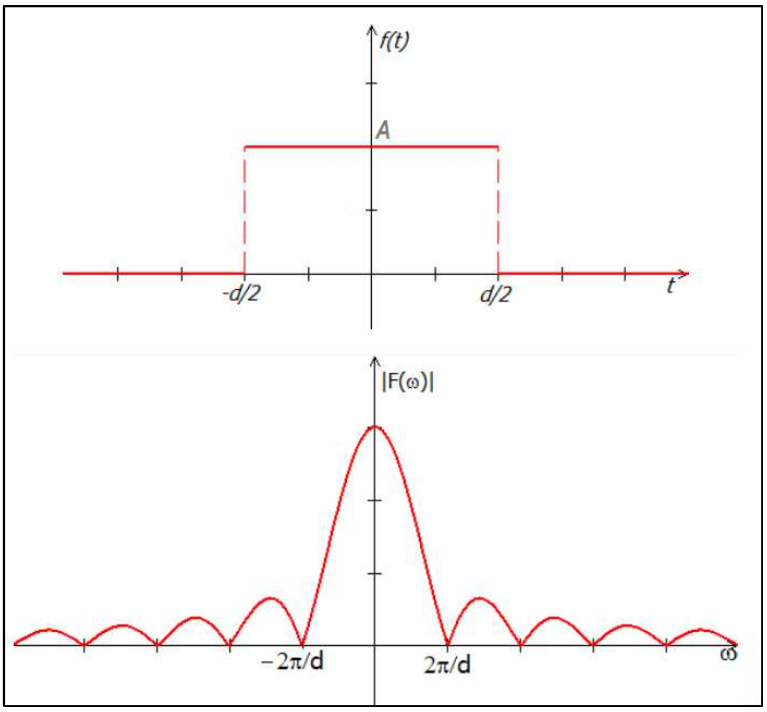


Figura 4 – Representações gráficas da função f e seu respectivo espectro $|F|$.

Exemplo 2:

Calculemos agora, a transformada de Fourier de um sinal periódico, do tipo "onda quadrada", definido pela seguinte função,

$$f(t) = \begin{cases} A, & |t| < \dfrac{d}{2} \\ \dfrac{A}{2}, & |t| = \dfrac{d}{2} \\ 0, & \dfrac{d}{2} < |t| < \dfrac{T}{2} \end{cases}$$

com $f(t) = f(t + kT)$ para $T \in \mathbb{R}$ constante e qualquer $k \in \mathbb{Z}$, logo sua frequência é $\omega_0 = \frac{2\pi}{T}$. A figura 5 apresenta o gráfico de f. Note que, este sinal esta defasado em $\pi/2$ radianos, ou seja, 90º.

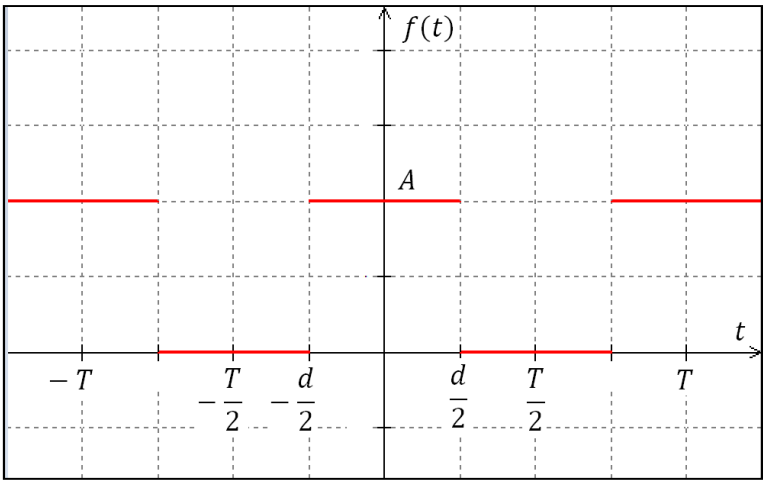


Figura 5 – Gráfico de f um sinal periódico quadrado.

Como f é periódica, então f pode ser representada por uma série de Fourier,

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t}, \qquad \text{onde } c_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{-in\omega_0 t}dt$$

Logo,

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

$$= \int_{-\infty}^{\infty} \left(\sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t} \right) e^{-i\omega t}dt$$

$$= \sum_{n=-\infty}^{\infty} c_n \int_{-\infty}^{\infty} e^{in\omega_0 t} e^{-i\omega t}dt$$

$$= \sum_{n=-\infty}^{\infty} c_n \int_{-\infty}^{\infty} e^{-i(\omega - n\omega_0)t}dt$$

$$\therefore\ F(\omega) = \sum_{n=-\infty}^{\infty} c_n \delta(\omega - n\omega_0)$$

Onde $\delta(\omega - n\omega_0)$ é a “função” delta de Dirac. Para mais detalhes veja [1] e [2]. Restando então calcular os coeficientes c_n, conforme segue,

$$c_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{-in\omega_0 t}dt$$

$$= \frac{A}{T}\int_{-\frac{d}{2}}^{\frac{d}{2}} e^{-in\omega_0 t}dt$$

$$= \frac{A}{T}\frac{1}{-in\omega_0}e^{-in\omega_0 t}\Big|_{-d/2}^{d/2}$$

$$= \frac{A}{T}\frac{1}{in\omega_0}\left(e^{\frac{in\omega_0 d}{2}} - e^{\frac{-in\omega_0 d}{2}}\right)$$

$$= \frac{Ad}{T}\frac{1}{\left(\frac{n\omega_0 d}{2}\right)}\frac{1}{2i}\left(e^{\frac{in\omega_0 d}{2}} - e^{\frac{-in\omega_0 d}{2}}\right)$$

$$= \frac{Ad}{T}\frac{\sin\left(\frac{n\omega_0 d}{2}\right)}{\left(\frac{n\omega_0 d}{2}\right)}$$

$$\therefore\ c_n = \frac{Ad}{T}sinc\left(\frac{n\omega_0 d}{2}\right)$$

Neste caso o espectro de frequências é discreto, analogamente ao espectro contínuo, temos, $c_n = |c_n|e^{i\phi_n} = a_n + ib_n$ com $a_n, b_n \in \mathbb{R}\ \forall n \in \mathbb{Z}$. Consequentemente,

$$|c_n| = \frac{1}{2}\sqrt{a_n^2 + b_n^2}$$

e

$$\phi_n = \tan^{-1}\left(-\frac{b_n}{a_n}\right)$$

A figura 6, apresenta o sinal f quando $A = 1$ e $d = 0{,}1$, seguido dos espectros de amplitude e de fases. Observe que o espectro de amplitudes é um pente de Dirac limitado por uma função sinc, onde os dois deltas de Dirac mais próximos à origem, representam a amplitude do harmônico principal, ω_0.

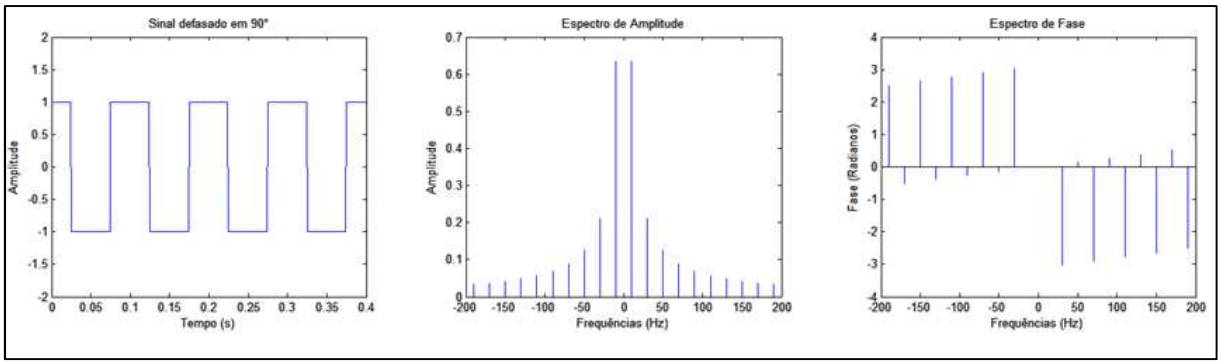


Figura 6 – Onda quadrada com defasagem de 90º e seus espectros de amplitudes e de fases.

A figura 7 apresenta, a título de exemplo, o mesmo sinal, porém, em fase. Observe que, o espectro de amplitudes é idêntico ao apresentado na figura 6, porém, existe uma grande diferença entre os espectros de fases.

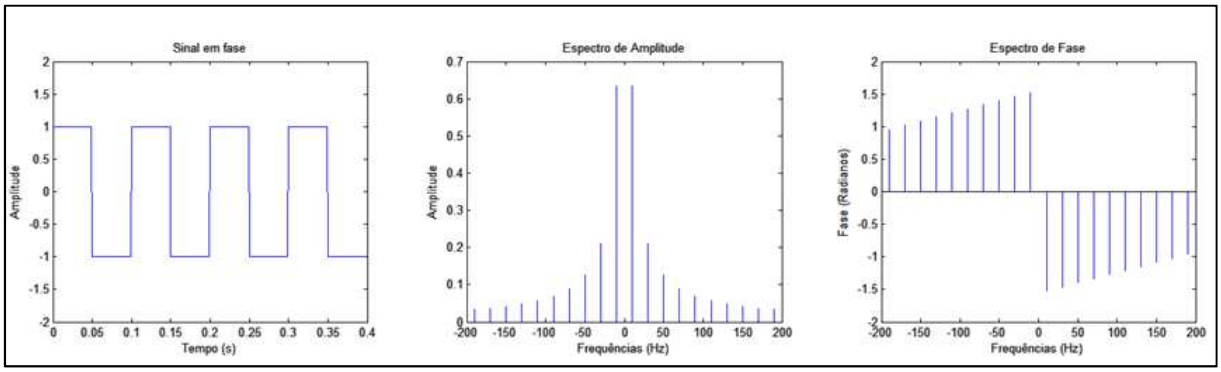


Figura 7 – Onda quadrada em fase relativo à origem e seus espectros de amplitudes e de fases.

É claro que não é objetivo deste tutorial uma revisão detalhada da teoria da transformada de Fourier, para maiores detalhes recomenda-se a leitura de [1], [2] e [3]

3. Transformada discreta de Fourier

Na prática, em aplicações computacionais o tempo é discreto, por isso, é necessário uma versão discreta da transformada de Fourier (DFT – *Discret Fourier Transform*), além disso, os sinais de entrada, que antes eram funções contínuas com domínio $(-\infty, +\infty)$, agora serão discretos e finitos, ou seja, serão vetores de comprimento $N \in \mathbb{N}$.

Um método para definir a transformada discreta é partir da discretização de um sinal contínuo (na prática, isto ocorre por meio da amostragem do sinal), dada por

$$x_n = f(nT) \quad \forall n \in \mathbb{Z}$$

ou seja, x_n é a versão discreta da função contínua $f(t)$ e T é a distância no eixo do tempo entre os pontos x_n, ou em linguagem de processamento de sinais é o intervalo de amostragem. Note que, x_n é uma sequência que se estende para $+\infty$ e para $-\infty$.

De forma generalizada, dado $x_n \in \mathbb{R}, n \in \mathbb{Z}$, a transformada de Fourier de tempo discreto de x_n pode ser definida, partindo da forma integral apresentada no tópico anterior, como:

$$F_k = \sum_{n=-\infty}^{\infty} x_n . e^{-ikn} \quad \text{onde} \quad k \in \mathbb{Z}$$

No entanto, na prática dispomos de dispositivos computacionais cuja memória é finita, daí a necessidade de reduzir esta transformada para o caso finito. Ou seja, x_n deve ser uma sequência finita, um vetor, com efeito, definimos um operador linear que leva o vetor $x = (x_0, \dots, x_{N-1})$ no vetor $F = (F_0, \dots, F_{N-1})$ onde $N \in \mathbb{N}$. Para maiores detalhes recomenda-se [1] e [4]. Aqui apenas definimos que, para um vetor,

$$v = (v_0, \dots, v_{N-1})$$

computamos sua transformada discreta de Fourier $F = (F_0, \dots, F_{N-1})$, por,

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} v_n \omega_N^{kn} \quad \text{para} \quad k = 0, 1, \dots, N-1$$

onde, $\omega_N = e^{-i\frac{2\pi}{N}}$. Ou ainda, em forma matricial,

$$\begin{bmatrix} F_0 \\ \vdots \\ F_{N-1} \end{bmatrix} = \frac{1}{N} \begin{bmatrix} \omega_N^0 & \cdots & \omega_N^{N-1} \\ \vdots & \ddots & \vdots \\ \omega_N^{N-1} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} v_0 \\ \vdots \\ v_{N-1} \end{bmatrix}$$

Note que, ω_N é um número complexo, e que em termos computacionais computar a multiplicação de matrizes exige, N multiplicações para cada F_k, e $N-1$ adições, ou seja, para computar a DFT, o dispositivo deve computar N^2 multiplicações e $N(N-1)$ adições, em notação *big O*, dizemos que a ordem de complexidade da DFT é $O(N^2)$, este tipo de complexidade é dita polinomial, e em geral algoritmos com tal complexidade são viáveis mas demandam alto custo computacional. Para piorar a situação, em geral os dispositivos computacionais não trabalham diretamente com números complexos. Lembrando que, dados $z_1, z_2 \in \mathbb{C}$, escrevemos $z_1 = a_1 + ib_1$ e $z_2 = a_2 + ib_2$ com $a_1, a_2, b_1, b_2 \in \mathbb{R}$, então o produto $z_1 z_2 = (a_1 + ib_1)(a_2 + ib_2) = a_1 a_2 - b_1 b_2 + i(a_1 b_2 + a_2 b_1)$, daí, uma multiplicação complexa equivale a 4 multiplicações reais e 2 adições reais. Para a adição complexa, tem-se que $z_1 + z_2 = a_1 + ib_1 + a_2 + ib_2 = a_1 + a_2 + i(b_1 + b_2)$, ou seja, duas adições reais.

Para saber mais sobre complexidade de algoritmos recomenda-se a leitura de [7] enquanto uma abordagem do caso particular da DFT é realizada em [1] (incluindo comparações e resultados experimentais).

4. FFT Cooley-Tukey

Conforme discutido no tópico anterior, a DFT é inviável computacionalmente, no entanto, nos anos 60 foi desenvolvido o primeiro algoritmo que ficou conhecido como transformada rápida de Fourier (FFT – *Fast Fourier Transform*), proposto por Cooley-Tukey [1], com tal algoritmo é possível reduzir a ordem de complexidade de $O(N^2)$ para $O(N\,Log_2 N)$, o que representou uma grande revolução em processamento digital de sinais.

A ideia proposta por Cooley-Tukey consiste essencialmente em fatorar a DFT, reduzindo o problema em DFTs menores.

Partindo da DFT,

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} v_n \omega_N^{kn} \quad \text{para } k = 0, 1, \dots, N-1$$

Supondo que N é um número composto, isto é, $N = N'N''$, com $N', N" \in \mathbb{N}$. Então podemos reescrever o somatório acima como dois somatórios, com os seguintes índices,

$$n' = 0, 1, \dots, N' - 1$$
$$n'' = 0, 1, \dots, N'' - 1$$

Pois,

$$n = 0, \dots, N-1 \Rightarrow \max\{n\} = N - 1$$

Mas,

$$N - 1 = N'N'' - 1 = N'N'' - 1 + N' - N' = N' - 1 + N'(N'' - 1)$$

Daí conclui-se que,

$$n = n' + N'n''$$

Analogamente, tem-se,

$$k = k'' + N''k' \quad \text{e} \quad \begin{matrix} k' = 0, \dots, N' - 1 \\ k'' = 0, \dots, N'' - 1 \end{matrix}$$

Consequentemente, podemos reescrever o somatório da seguinte forma,

$$F_{(k''+N''k')} = \sum_{n'=0}^{N'-1} \sum_{n''=0}^{N''-1} W^{(n'+n''N')(k''+N''k')} v_{(n'+N'n'')}$$

Neste ponto, utilizou-se $\omega = \omega_N$, para não sobrecarregar desnecessariamente a notação.

Aplicando a distributiva no expoente de ω apresentado acima, obtém-se,

$$(n' + n''N')(k'' + N''k') = n'k'' + n'N''k' + n''N'k'' + n''N'N''k'$$

No lado esquerdo da equação anterior, aparece uma parcela a multiplicar $N'N''$, nesta, a exponencial será igual um, pois, $e^{-K\frac{2\pi i}{N}N'N''} = 1$, onde K é um número inteiro qualquer, portanto essa parcela desaparece do expoente. Para simplificar a escrita, tome $\gamma = \omega^{N'}$ e $\beta = \omega^{N''}$, substituindo tudo na DFT, tem-se,

$$F_{(k''+N''k')} = \sum_{n'=0}^{N'-1} \left[\beta^{n'k'} \omega^{n'k''} \sum_{n''=0}^{N''-1} \gamma^{n''k''} v_{(n'+N'n'')} \right]$$

Ao trocar F_N por $F_{N'N''}$, finalmente encontra-se a expressão chave para o algoritmo de Cooley-Tukey,

$$F_{k'k''} = \sum_{n'=0}^{N'-1} \left[\beta^{n'k'} \omega^{n'k''} \sum_{n''=0}^{N''-1} \gamma^{n''k''} v_{n'n''} \right]$$

Este resultado representa a decomposição de uma DFT de comprimento N em N' DFTs menores de comprimento N''. O algoritmo de Cooley-Tukey consiste em aplicar esta decomposição iterativamente, reduzindo o máximo possível o tamanho das transformadas.

Maiores detalhes sobre a dedução acima são apresentados em [1] e uma abordagem mais geral é encontrada em [5] e [6].

Um caso particular do algoritmo, ocorre quando $N = 2^m$ com $m \in \mathbb{N}$, neste caso temos uma FFT *radix-2* Cooley-Tukey. Este caso é interessante para fins práticos por sua facilidade de implementação.

Como $N = 2^m$, tome na dedução anterior, n' igual a 2 e n'' igual a 2^{m-1}, isto é, $n' = 2$ e $n'' = n/2$, daí,

$$F_k = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \omega^{2nk} v_{2n} + \omega^k \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \omega^{2nk} v_{2n+1} \qquad (*)$$

$$F_{k+\frac{N}{2}} = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \omega^{2nk} v_{2n} - \omega^{k} \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \omega^{2nk} v_{2n+1} \qquad (**)$$

Nas duas equações acima, $k = 0, \ldots, \frac{N}{2} - 1$, onde foi utilizado o fato que $\beta = \omega^{N''} = \omega^{N/2} = e^{-\pi i} = -1$. Em termos computacionais, isto equivale a dividir em duas transformadas menores, dadas por:

$$F'_k = \sum_{n=0}^{\frac{N}{2}-1} \mu^{nk} v_{2n} \quad e \quad F''_k = \sum_{n=0}^{\frac{N}{2}-1} \mu^{nk} v_{2n+1}$$

Onde $\mu = \omega^2$. Cada uma dessas expressões equivale a uma transformada de Fourier de comprimento N/2.

O valor de cada F_k com $k = 0, \ldots, \frac{N}{2} - 1$, é obtido pelas expressões

$$F_k = F'_k + \omega^k F''_k \text{ e } F_{k+\frac{N}{2}} = F'_k - \omega^k F''_k$$

Agora, pode-se bisseccionar as transformadas de N/2 pontos em transformadas menores de N/4 pontos, e continuar com esse processo até que as transformadas se reduzam a apenas um ponto, onde a transformada de um ponto é o próprio ponto.

A operação básica para computar o Cooley-Tukey conforme mostrado nas equações (*) e (**), é conhecida como "*Butterfly*" ou "Borboleta", esta operação é ilustrada pelo diagrama da figura 8.

O processo de fatoração descrito acima, pode ser realizado em forma de matrizes, para o leitor interessado recomenda-se o vídeo [8].

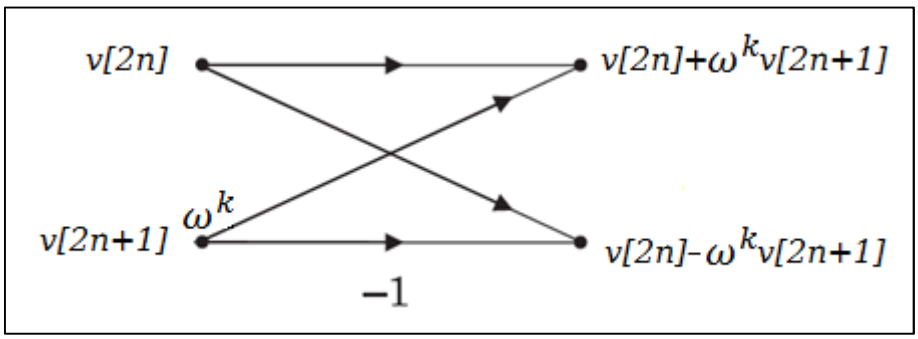


Figura 8 – Borboleta para computar Cooley-Tukey base 2 FFT *decimation-in-time*.

A figura 9 apresenta o fluxograma das borboletas para um exemplo onde $N = 8$. Observe que, a indexação dos pontos do vetor de entrada está com uma permutação conhecida por espelhamento binário, como o nome propõe, esta permutação consiste em espelhar os bits do índice em binário, por exemplo, $4 = 0b100$, espelhando os bits tem-se $0b001 = 1$. Existem duas formas de implementar o *radix-2* Cooley-Tukey, estas

duas formas dependem de como iniciamos a fatoração, isto é, a escolha dos n', n'', k' e k'', a que foi apresentada acima, é conhecida como *decimation-in-time* (DIT), nessa espelhamento ocorre na entrada, pois a frequência é conservada, no caso *decimation-in-frequency* (DIF), o espelhamento ocorre no vetor de saída. Portanto, durante a implementação do algoritmo *radix-2* Cooley-Tukey, seja DIT ou DIF, deve-se considerar o espelhamento binário. Em [1] apresenta-se um exemplo ilustrativo para melhor compreensão do *radix-2* Cooley-Tukey e uma breve análise de como as bisseções do algoritmo, no caso DIT, implicam no espelhamento binário. Em [5] e [6] são apresentados outros algoritmos para computar a FFT, além de variantes do Cooley-Tukey e outros algoritmos para cômputo da FFT.

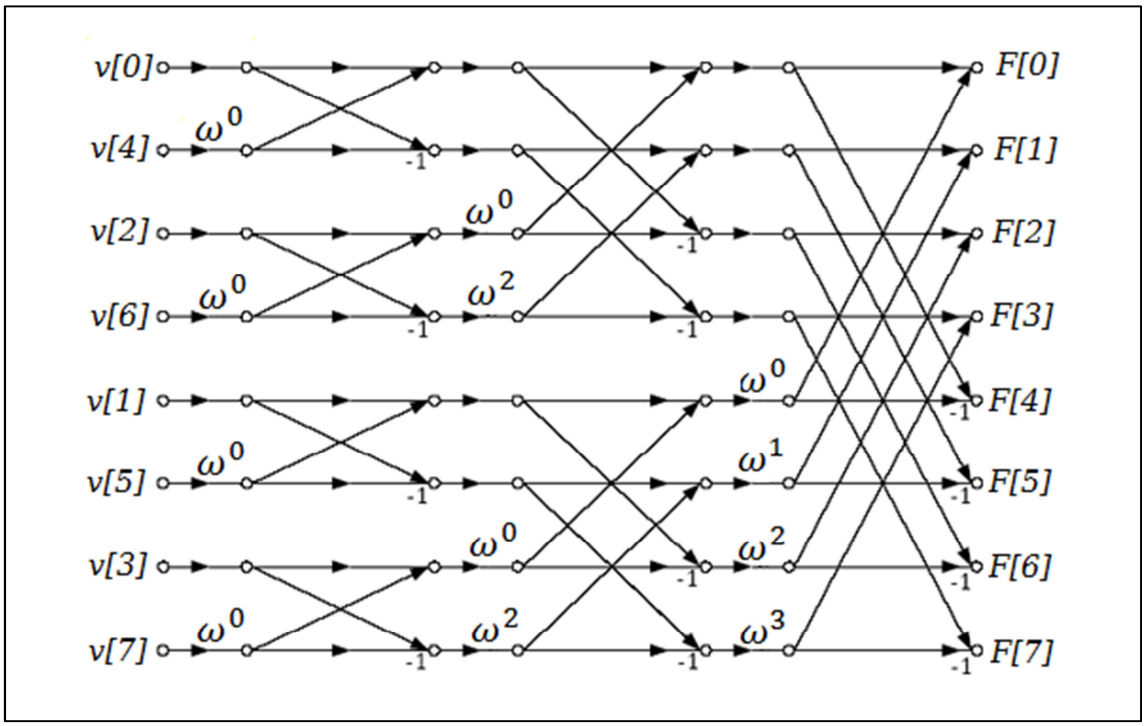


Figura 9 – Diagrama das Butterflys para FFT-DIT Cooley-Tukey, com N=8.

5. Algoritmo de Goertzel

O algoritmo de Goertzel (AG) não é um algoritmo do tipo FFT, mas sim, um algoritmo que permite computar coordenadas especificas de $F = (F_0, \dots, F_{N-1})$, isto é muito útil em aplicações onde não necessitamos do espectro inteiro, tais como filtragem de sinais, onde estamos interessados apenas em certas frequências, como por exemplo, na decodificação dos tons DTMF.

Para compreender este algoritmo, apresenta-se primeiramente, a regra de Horner, esta regra é útil para reduzir o custo computacional no computo de polinômios, seja,

$$v(x) = \sum_{n=0}^{N-1} v_n x^n = v_{N-1}x^{N-1} + \cdots + v_0$$

Consideremos que $v(x)$ é um polinômio com coeficientes reais. Para avaliarmos o valor em um dado ponto $\alpha \in \mathbb{R}$, temos então que computar $N-1$ multiplicações dos coeficientes, mais as $\sum_{n=1}^{N-2} n$ multiplicações das potências. Por exemplo, dado $v(x) = v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0$, então para computar $v(\alpha)$, computa-se 3+2+1=6 multiplicações das potências, mais 4 multiplicações dos coeficientes, totalizando 10 multiplicações.

A regra de Horner consiste em rearranjamos o polinômio da seguinte forma

$$v(\alpha) = \big(\cdots\big((v_{N-1}\alpha + v_{N-2})\alpha + v_{N-3}\big) + \cdots + v_1\big)\alpha + v_0$$

No exemplo em que $v(x) = v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0$, temos,

$$v(\alpha) = \Big(\big((v_4\alpha + v_3)\alpha + v_2\big)\alpha + v_1\Big)\,\alpha + v_0$$

É fácil ver que, desta forma, temos que computar apenas 4 multiplicações, observe também que a quantidade de adições permanece a mesma. É claro que, se os valores das potências são previamente memorizados no sistema, a regra de Horner não oferece nenhuma vantagem computacional.

Se $\alpha = \omega^k$, então a regra de Horner computa uma coordenada de F com N-1 multiplicações complexas e N-1 somas complexas. Como observado no tópico 3, uma multiplicação complexa equivale a 4 multiplicações reais e 2 adições reais, enquanto a adição de complexos equivale a 2 adições reais, logo, para computar k-ésima componente da transformada de Fourier, precisamos de 4(N-1) multiplicações e 4(N-1) somas reais.

O Algoritmo de Goertzel, permite reduzir este custo computacional para cômputo de uma componente da transformada de Fourier. Desejamos calcular,

$$F_k = \sum_{n=0}^{N-1} \omega^{nk} v_n$$

para um k fixo.

Definamos o seguinte polinômio,

$$p(x) = (x - \omega^k)(x - \omega^{-k})$$

Lembrando que $\omega^k = e^{-i\frac{2\pi}{N}k}$ e $e^{-i\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}k\right) - i\sin(\frac{2\pi}{N}k)$, então obtemos,

$$p(x) = x^2 - 2\cos\left(\frac{2\pi}{N}k\right)x + 1$$

Seja,

$$v(x) = \sum_{n=0}^{N-1} v_n x^n$$

Pelo algoritmo da divisão de polinômios, podemos escrever

$$v(x) = p(x)Q(x) + r(x)$$

onde $Q(x)$ é o polinômio quociente e $r(x)$ é o resto da divisão de polinômios. Então F_k é computado pelo resto da divisão, isto é

$$\begin{aligned} F_k &= v(\omega^k) \\ &= p(\omega^k)Q(\omega^k) + r(\omega^k) \\ &= (\omega^k - \omega^k)(\omega^k - \omega^{-k})Q(\omega^k) + r(\omega^k) \\ &= r(\omega^k) \end{aligned}$$

Se $v(x)$ tem coeficientes complexos, então o algoritmo da divisão por $p(x)$ demanda 2(N-2) multiplicações reais e 4(N-2) adições reais. Se $v(x)$ tem coeficientes reais então o algoritmo da divisão demanda N-2 multiplicações e 2(N-2) adições reais. Como $r(x)$ tem grau 1, computar $r(\omega^k)$ após a divisão, requer apenas mais uma multiplicação complexa e mais uma adição complexa. Consequentemente, dada uma entrada complexa o custo computacional do algoritmo de Goertzel é de 2N-1 multiplicações reais e 4N-1 adições reais, para cada componente.

A figura 10 apresenta o diagrama em blocos do algoritmo de Goertzel, onde a entrada dos coeficientes é por deslocamento (*shift*), sejam as entradas reais ou complexas,

$$A = 2\cos\left(\frac{2\pi}{N}k\right)$$

após deslocar os coeficientes de $v(x)$ no circuito da figura 10, obtendo o resto da divisão $r(x)$. Calcula-se a componente,

$$F_k = \omega^k r_1 + r_0$$

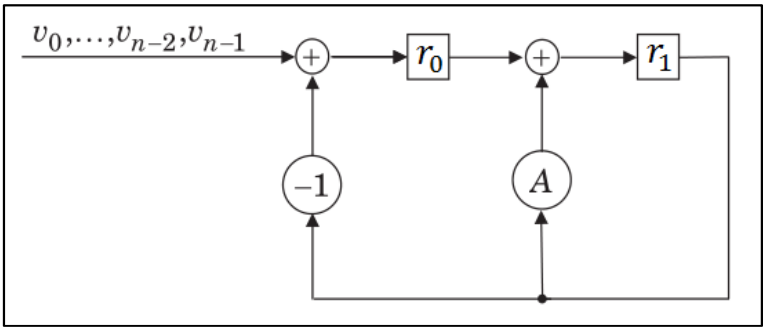


Figura 10 – Diagrama em blocos do AG [5].

Observe que o polinômio $Q(x)$ não é calculado, pois não tem relevância para o AG, além disso, note que este circuito tem a forma de um filtro auto regressivo, isso é consequência do fato do algoritmo da divisão ser um filtro auto regressivo [5].

6. Codificando a FFT

A seguir são explicadas cada parte de um programa que implementa o radix-2 Cooley-Tukey FFT-DIT, o programa foi codificado em linguagem C, no entanto, uma vez compreendida sua implementação, é fácil "traduzi-lo" para outras linguagens.

A figura 11, apresenta o cabeçalho, as diretrizes de pré-processamento de chamada das bibliotecas e a definição de constantes do programa.

```
1  /*
2      Programa de testes Transformada Discreta de Fourier
3      Algoritmo FFT Cooley-Tukey radix-2 Decimation-in-time
4      Desenvolvido por Junio Cesar Ferreira
5      Data: 19/06/2016
6  */
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <math.h>
10
11 #define N    1024  // Comprimento dos vetores de entrada e saída
12 #define pi   3.14159265358979323846264338327950288419716939937751
13
```

Figura 11 – Cabeçalho e diretrizes de processamento.

O trecho de código apresentado na figura 12, corresponde à função que faz o espelhamento binário, conforme visto no tópico 4, será necessária uma rotina para fazer o espelhamento, existem varias formas de implementar o espelhamento, neste código, optou-se por utilizar o comando *switch*, nativo da linguagem C, para retornar o $\log_2 N$, pois, o computo do logaritmo demanda custo computacional superior ao processamento do seleção do tipo *switch*, é importante ficar atento ao alterar o programa, pois se o valor de N não for nenhuma das opções (*case*), haverá problemas na execução. Observe que, a operação de espelhamento binário é realizada no laço *for* da linha 40.

```
14 unsigned int Table_Reverse(unsigned int index, unsigned int Lenght){
15     unsigned int mirror, exp, Lim;
16     switch (Lenght){
17         case 32:
18             Lim=5;
19             break;
20         case 64:
21             Lim=6;
22             break;
23         case 128:
24             Lim=7;
25             break;
26         case 256:
27             Lim=8;
28             break;
29         case 512:
30             Lim=9;
31             break;
32         case 1024:
33             Lim=10;
34             break;
35         case 2048:
36             Lim=11;
37             break;
38     }
39     mirror = 0;
40     for (exp=0;exp<Lim;exp++){
41         mirror=mirror<<1;
42         mirror+=(0x01&index);
43         index=index>>1;
44     }
45     return mirror;
46 }
```

Figura 12 – Rotina de espelhamento binário do índice.

O código examinado, é um programa com objetivo didático, por isso, a FFT foi codificada dentro da rotina *main()*, no entanto, na prática, é recomentado fazer uma função que compute a FFT fora da rotina principal, para tanto, no caso da linguagem C temos duas opções, ou declarar os vetores de processamento como variáveis globais ou utilizar ponteiros para que a função acesse os vetores de entrada e de saída, sendo a primeira solução mais simples para iniciantes. Na figura 13, apresenta-se o código da primeira parte da rotina principal, onde declaramos e inicializamos o vetor x, este simula um sinal de entrada, lembrando que em uma aplicação prática, este vetor é carregado com amostras do sinal mensurado, por isso, no caso geral, a entrada sempre é composta apenas por valores reais, advindos de medições de grandezas físicas. Os valores de entrada em que x é inicializado estão ocultos por serem neste momento irrelevantes. Seguindo, nas linhas 91 e 92, onde são declarados os vetores Xreal e Ximag, pois o sistema não possuí variáveis do tipo número complexo, sendo assim, precisamos representar as partes real e imaginária da transformada separadamente. Na linha seguinte, declaramos mais um vetor V, este receberá o resultado do módulo da

FFT. Os vetores Wr e Wi são os coeficientes da transformada, como vistos nos tópicos 3 e 4, sendo Wr a parte real e Wi a parte imaginária. A variável pi_N pode ser definida como constante. As variáveis ar, ai, br e bi são as entradas da *butterfly*, conforme mostrado na figura 8, é evidente o significado dos r's e i's nestas variáveis. Na linha 97, temos a declaração de algumas variáveis do tipo inteiro, o 'i' e o 'k' são indexadores de uso geral para laços *for*, a variável L é utilizada para limitar os deslocamentos das operações *butterflys* em um nível, enquanto a variável desl representa o fator de deslocamento das *butterflys*, a variável P indica em qual "coluna" do diagrama da figura 9 encontra-se o processamento, enquanto a variável NP indica o nível, este nível nada mais é, que em qual grupo de *butterflys* da coluna encontra-se o processamento, por exemplo, na primeira coluna da figura 9, temos quatro níveis com apenas uma *butterfly* por nível, já na segunda coluna, encontramos 2 níveis com duas *butterflys* em cada nível.

Ainda no trecho de código da figura 13, encontramos dois laços *for* que realizam a inicialização dos vetores, note que o processamento será realizado sobre o vetor de saída, que na verdade é complexo, sendo representado no programa pelos vetores Xreal e Ximag.

```
47
48  main(){
49      double x[N]={
91      double Xreal[N];          // Vetor Saída Real
92      double Ximag[N];          // Vetor Saída Imaginária
93      double V[N];
94      double Wr[N/2], Wi[N/2];  // Vetores de coeficientes
95      double pi_N=2*pi/N;;      // Auxiliar Pi*n
96      double ar, ai, br, bi;
97      unsigned int i, k, L, desl, P, NP;
98      printf("\n * Transformada Rapida de Fourier * ");
99      // Calculo da FFT
100     // Carrega vetores
101     for (i=0;i<N;i++){
102         Xreal[Table_Reverse(i,N)]=x[i];
103         Ximag[i]=0;
104     }
105     // Carrega vetor de coeficientes
106     for (i=0;i<N/2;i++){
107         Wr[i] = cos(pi_N*i);
108         Wi[i] = -sin(pi_N*i);
109         printf("\nWr[%d]=%.3f \tWi[%d]=%.3f",i,Wr[i],i,Wi[i]);
110     }
```

Figura 13 – Inicio da rotina principal, declarações de variáveis e inicialização dos vetores.

A figura 14 exibe o trecho do código onde é computada a FFT (parte mais importante deste código), o primeiro laço *for* da linha 113 é responsável pela varredura das colunas, conforme explicado acima, por exemplo, o diagrama da figura 9, onde N=8, tem 3 colunas. O segundo laço *for* que é interno ao primeiro, é responsável pela varredura dos níveis, novamente observando o diagrama da figura 9, como exemplo,

neste temos na primeira coluna 4 níveis, enquanto na segunda coluna tem 2 níveis, e na terceira apenas 1. O laço *for* interno (linha 121) é responsável pelos deslocamentos das operações *butterflys* do nível. Enquanto o laço *for* da linha 137, tem como finalidade normalizar os resultados e computar o módulo, carregando o vetor V com o resultado final, o espectro de amplitudes da transformada de Fourier.

```
111
112     // Computa FFT Cooley-Tukey radix-2
113     for (P=N;P>=2;P=P/2){
114         NP = (N/P);  // Comprimento da DFT decomposta
115         printf("\nP=%d", P);
116         L=P/2;      // Limite para deslocamentos da Butterfly do nível atual
117         desl = 2*NP; // Fator de deslocamento interno do nível
118         printf("\n L=%d; N/P=%d; desl=%d", L, NP, desl);
119         // Deslocamento da Butterfly principal
120         for (k=0;k<NP;k++){
121             for (i=0;i<L;i++){
122                 printf("\n i=%d, k=%d",i,k);
123                 // Prepara entradas da Butterfly
124                 ar=Xreal[k+i*desl];
125                 ai=Ximag[k+i*desl];
126                 br=Xreal[k+NP+i*desl];
127                 bi=Ximag[k+NP+i*desl];
128                 // Computa Butterfly DIT ...
129                 Xreal[k+i*desl]=ar+br*Wr[k*P/2]-bi*Wi[k*P/2];
130                 Ximag[k+i*desl]=ai+br*Wi[k*P/2]+bi*Wr[k*P/2];
131                 Xreal[k+NP+i*desl]=ar-br*Wr[k*P/2]+bi*Wi[k*P/2];
132                 Ximag[k+NP+i*desl]=ai-br*Wi[k*P/2]-bi*Wr[k*P/2];
133             }
134         }
135     }
136     // Normaliza resultado
137     for (i=0;i<N;i++){
138         Xreal[i]=2*Xreal[i]/N;
139         Ximag[i]=2*Ximag[i]/N;
140         V[i]=sqrt(pow(Xreal[i],2)+pow(Ximag[i],2));
141     }
```

Figura 14 – Trecho do código que computa a FFT.

Como este programa foi idealizado para ser executado em um prompt de comando em um computador, apenas a título de curiosidade é apresentado na figura 15 a parte final do código que consiste apenas na etapa de exibição da informação, não tendo relevância para o entendimento da implementação do algoritmo.

```
142     // Exibe vetor de entrada
143     printf("\n\nEntrada:\n");
144     for (i=0;i<N/2;i++) printf(" %.4f ", x[i]);
145     printf("\n");
146     for (;i<N;i++) printf(" %.4f ", x[i]);
147     // Exibe resultado em complexos
148     printf("\n\nResultado FFT:\n");
149     for (i=0;i<N/4;i++){
150         if (Ximag[i]>=0) printf(" %.2f+%.2fi ", Xreal[i],Ximag[i]);
151         else printf(" %.2f%.2fi ", Xreal[i],Ximag[i]);
152     }
153     printf("\n");
154     for (;i<N/2;i++){
155         if (Ximag[i]>=0) printf(" %.2f+%.2fi ", Xreal[i],Ximag[i]);
156         else printf(" %.2f%.2fi ", Xreal[i],Ximag[i]);
157     }
158     printf("\n");
159     for (;i<3*N/4;i++){
160         if (Ximag[i]>=0) printf(" %.2f+%.2fi ", Xreal[i],Ximag[i]);
161         else printf(" %.2f%.2fi ", Xreal[i],Ximag[i]);
162     }
163     printf("\n");
164     for (;i<N;i++){
165         if (Ximag[i]>=0) printf(" %.2f+%.2fi ", Xreal[i],Ximag[i]);
166         else printf(" %.2f%.2fi ", Xreal[i],Ximag[i]);
167     }
168     // Exibe módulo do resultado
169     printf("\n\nModulo da saida:\n");
170     for (i=0;i<N/2;i++) printf(" %.4f ", V[i]);
171     printf("\n");
172     for (;i<N;i++) printf(" %.4f ", V[i]);
173 }
```

Figura 15 – Parte final da rotina principal, apenas exibe resultados numéricos no prompt.

7. Codificando o AG

Codificar o AG é bem mais simples do que a FFT, pois conforme visto no tópico 5, o AG tem como entrada o vetor correspondente ao sinal no domínio de tempo, e como saída apenas uma coordenada do vetor de espectro, ou seja, apenas um ponto ou uma componente da DFT resultante. Por isso, optou-se por apresentar apenas o código de implementação da função ou sub-rotina, *Goertzel_Algorithm( )*, visto na figura 16.

A linhas 1 do trecho de código apresentado na figura 16, mostra a inclusão do arquivo math.h, pois precisamos das funções matemáticas: seno, cosseno e raiz quadrada, respectivamente sin, cos e sqrt.

As linhas 3,4 e 5 apresentam as constantes do programa, sendo a constante universal π, o comprimento do vetor a ser utilizado e a frequência de amostragem em Hertz.

A linha 7, mostra a declaração de um vetor do tipo *double* como variável global, este vetor será utilizado para armazenar o resultado da transformada de Fourier. O tipo *double* foi adotado apenas por preferencia do autor, poderia ser do tipo *float*.

Na linha 11, temos a função *Goertzel_Algorithm()*, cujo o parâmetro de entrada Fr é a frequência em que se deseja obter o resultado. Na linha 12, já dentro da função *Goertzel_Algorithm()* , temos a declaração da variável k, esta inicialização de k equivale a

$$k = Fr\frac{N}{Fs}$$

onde Fr é a frequência desejada, N é o comprimento do vetor de entrada e Fs é a frequência de amostragem. Logo k é a melhor aproximação, do índice do vetor de saída da transformada, cuja componente representa a frequência Fr. Aproximação, pois observe que as contas são realizadas com ponto flutuante, mas o resultado deve ser convertido para inteiro.

A variável w recebe o resultado da conversão da frequência em rad/s referente ao índice k, este resultado é utilizado nas linhas seguintes para carregar as variáveis cosw e sinw que recebem os resultados das funções cosseno e seno. Ainda na sequência (linha 16) temos a declaração e inicialização da variável A, conforme visto no tópico 5.

Na linha 17, são declaradas as variáveis r0, r1 e r2, que representam os blocos apresentados no diagrama da figura 10, logo estes serão os registros utilizados para computar o algoritmo recursivamente fazendo os deslocamentos. Finalizando a declaração de variáveis, na linha 18, temos a variável n, que é utilizada para controle do laço *for*.

O laço *for* da linha 20, essencialmente executa o algoritmo mostrado no diagrama da figura 10. Na linha 25 r0 recebe o resultado correspondente à componente real e na linha 26 r1 recebe o resultado correspondente à componente imaginária,

finalizando a rotina na linha 27 com o cômputo da amplitude que é carregada no vetor de saída.

Para alterar essa sub-rotina para retornar a amplitude no lugar de carrega-la em uma variável global, conforme é apresentado, basta modificar a declaração da função, substituindo o tipo *void* por *double*, e no final substitua F[k] por r2, adicionando em seguida mais uma linha com o comando “return r2”;.

```
1  #include <math.h>
2
3  #define pi 3.14159265358979323846264433
4  #define N  512
5  #define Fs 8000
6
7  double F[N]; // Variavel global, vetor que resultado da DFT
8
9  // *** Função Algoritmo de Goertzel (AG) para uma frequência
10 void Goertzel_Algorithm(double Fr){
11     // Variaveis locais
12     int k = (int)(Fr * N / Fs);
13     double w=2*pi*k/N;
14     double cosw = cos(w);
15     double sinw = sin(w);
16     double A = 2*cosw;
17     double r0=0, r1=0, r2; // Auxiliares para contas do resto
18     unsigned short n;  // indexador
19     // Computa Algoritmo de Goertzel
20     for (n=0;n<N;n++){
21         r0 = A*r0-r1+v[N-n-1];
22         r2=r1;
23         r1=r0;
24     }
25     r0=(r1-r2*cosw);        // Componente Real
26     r1=(r0*sinw);           // Componente imaginária
27     F[k]=sqrt(r0*r0+r1*r1); // Magnitude
28 }
```

Figura 16 – Codificação do algoritmo de Goertzel em uma sub-rotina.

8. Software didático

O software apresentado aqui, pode ser encontrado em

< https://github.com/JunioCesarFerreira/Teclado_DTMF_Simula-o>.

8.1 Interface que exibe sinal no domínio do tempo e da frequência

Esta é uma interface desenvolvida apenas para mostrar o sinal no domínio do tempo e seu respectivo espectro de frequências, sendo possível. As entradas são realizadas em um teclado apresentado na interface. Enquanto as saídas são apresentadas em dois gráficos também apresentados na interface.

A figura 17, apresenta a interface em execução. Onde podemos observar o *GroupBox* DTMF_key, onde encontra-se o teclado DTMF, além disso, logo abaixo do título encontra-se uma *CheckBox* chamada “Quad”, quando esta está selecionada, o sinal gerado passa a ser uma onda quadrada, no caso contrario, o sinal gerado é uma senoide normalizada. Esta opção foi inserida para simular situações em que o sinal é gerado por um microcontrolador simples, ou seja, o sinal é digital advindo de uma saída digital tipicamente TTL.

Logo abaixo do *GroupBox* “DTMF_key”, temos um campo de saída de texto, neste aparecem os número digitados no teclado DTMF.

Além dos botões de entrada do teclado DTMF, a interface possui mais dois botões, sendo um para limpeza dos dados e o outro para alternar os algoritmos, sendo que por default a interface inicia com a FFT Cooley-Tukey. O botão de alteração dos algoritmos encontra-se em uma *GroupBox* com título “Decode”, logo acima do botão encontra-se uma saída de texto que exibe o algoritmo que esta ativo no momento. Logo abaixo do botão “Change Algorithm” aparecem dois pequenos textos, estes exibem as duas frequencias encontradas no espectro com amplitude máxima, ainda abaixo destes, apresenta-se mais uma saída de texto que exibe o número decodificado com base no espectro de frequências.

A maior parte da interface é ocupada pelos dois gráficos de saída, sendo o de cima (preto) o tom DTMF no domínio do tempo, enquanto o de baixo (azul) apresenta o espectro de amplitudes computado pelo algoritmo que estiver ativo. Veja figura 17.

Uma vez compreendido o funcionamento da interface, o leitor está apto a compreender o código, conforme mencionado no início desta seção, o código encontra-se disponível no Github.

Observação: Note que a decodificação para de funcionar quando o sinal é quadrado e o algoritmo escolhido é a FFT, isto decorre da componente contínua que aparece. Este problema pode ser contornado, porém, na prática o algoritmo recomendado para esta aplicação é o AG, por demandar menos custo computacional.

Adicional de revisão: Após a primeira postagem no repositório, adicionei uma versão 3 do código deste aplicativo, nesta está separado em classes os métodos que executam os algoritmos e os métodos de geração dos sinais e som. Isto facilita a compreensão do código, ainda assim deixo a versão 2 disponível, pois esta deve ser mais compreensível para programadores menos acostumados a programação orientado a objetos.

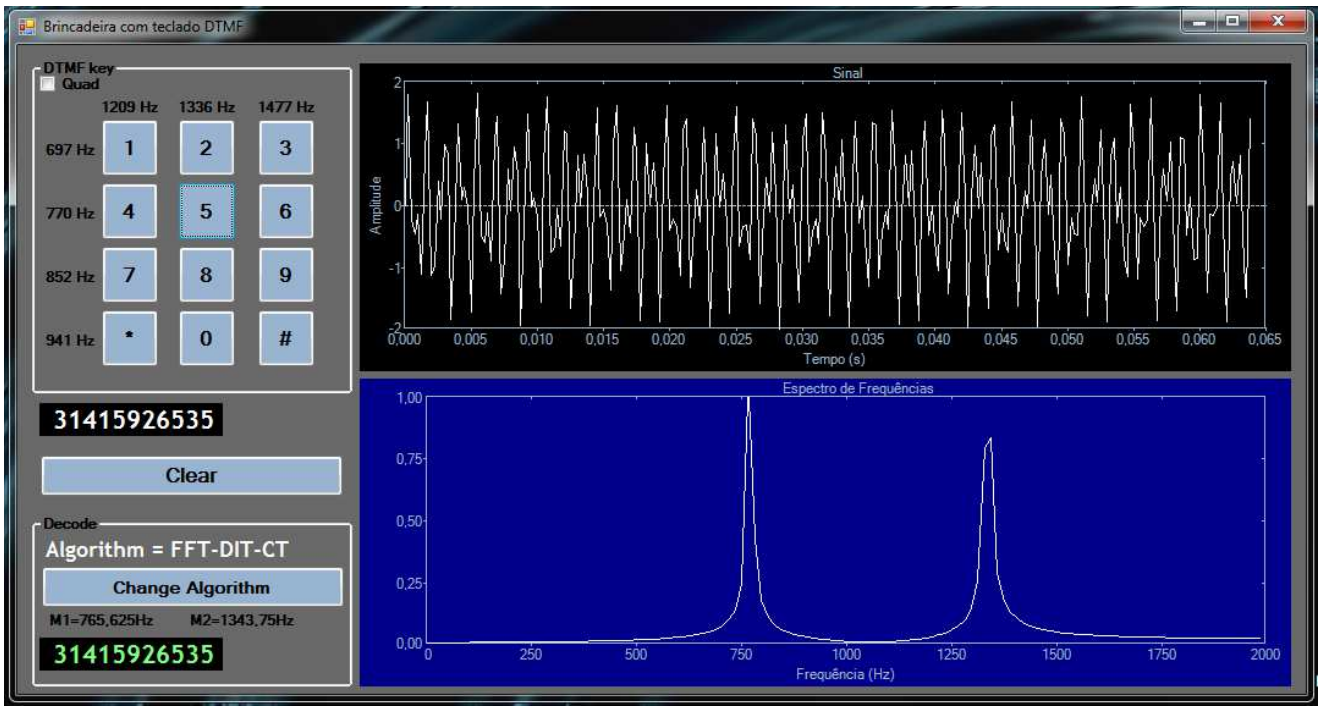


Figura 17 – Visual da interface.

8.2 Sugestão para uso em um microcontrolador

No repositório está incluso um exemplo, codificado em C usando o MikroC Pro V3.2. Para a validação do funcionamento, utilizou-se uma simulação do circuito no software Proteus 7.8.

O circuito utilizado para simular tal situação é visto na figura 18. Neste adotou-se por usar o MCU PIC18F2550 da Microchip, tanto para gerar um sinal DTMF, quanto apara decodificar o sinal, porém em microcontroladores separados, isto é, um microcontrolador gera o sinal e o outro decodifica o sinal.

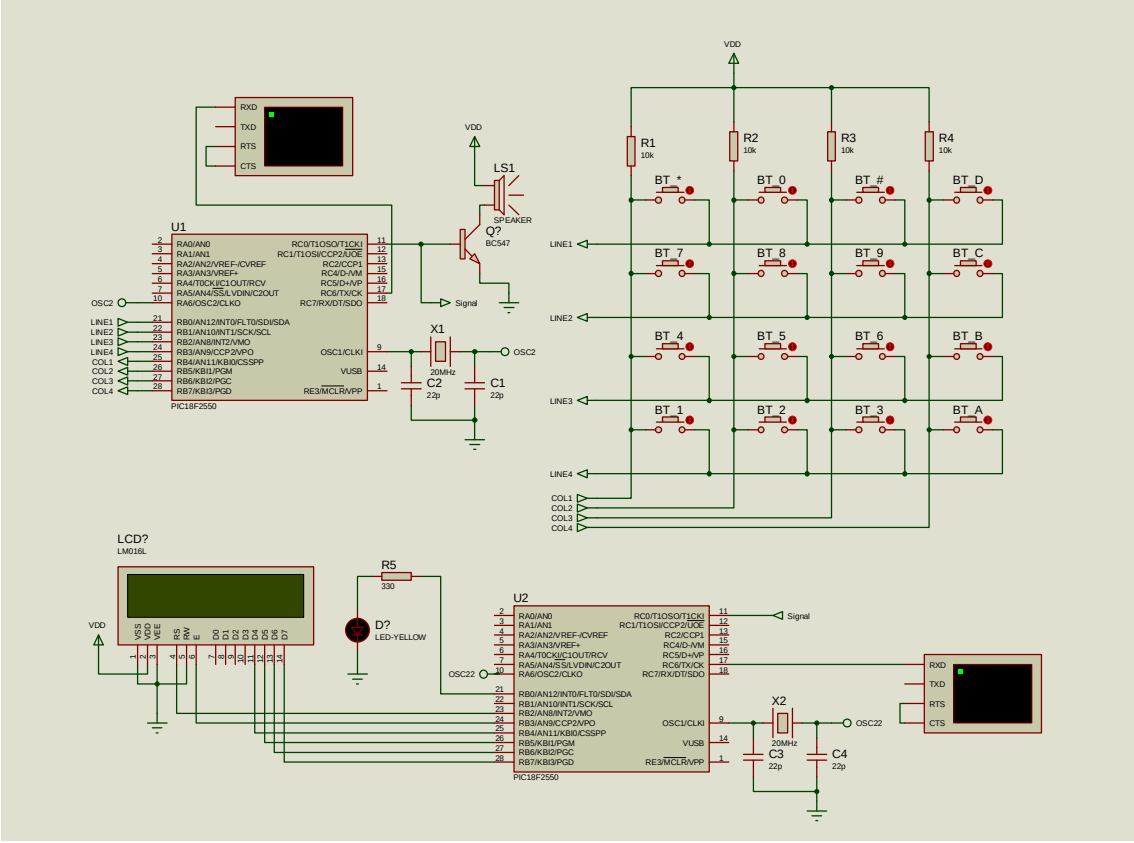
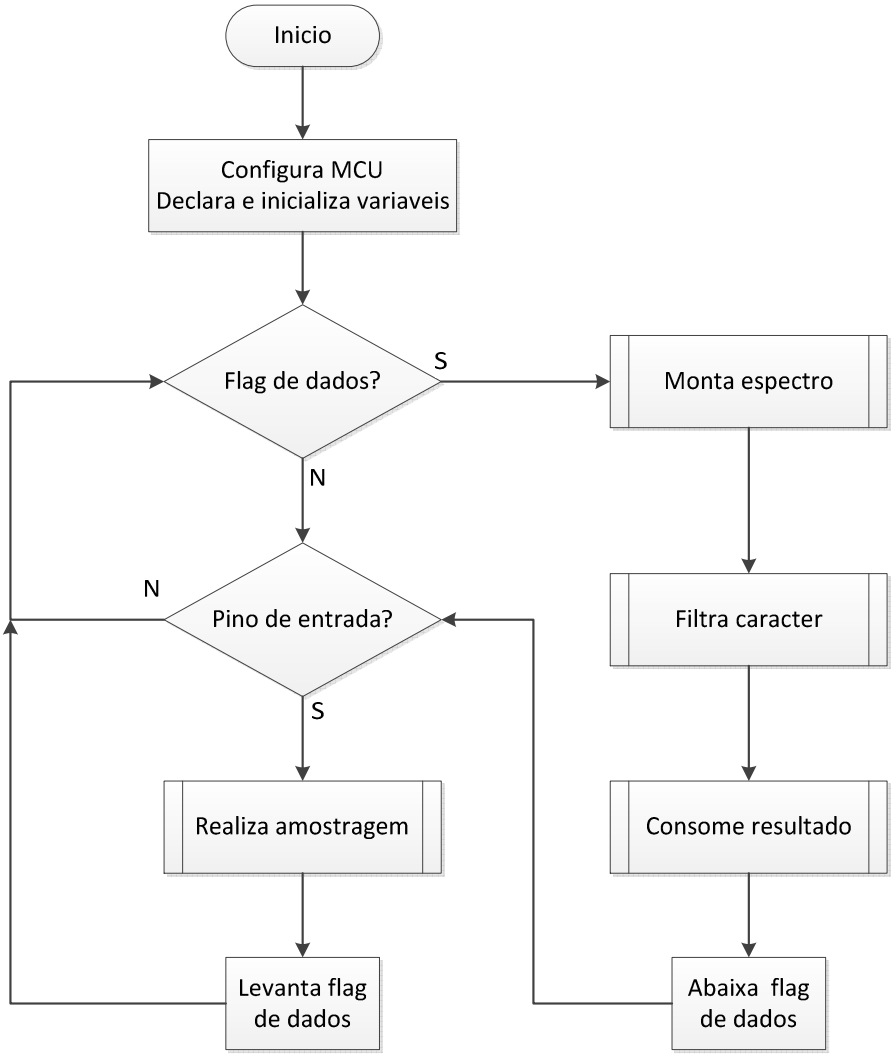


Figura 18 – Visual da interface.

O funcionamento do circuito é simples, como é visto na figura 18, o microcontrolador U1 é conectado ao teclado matricial, o um speaker e a um terminal serial, este ultimo é apenas para fins de debug e acompanhamento no processo de validação do funcionamento, ao fechar qualquer tecla, o microcontrolador detecta na varredura e gera os sinais quadrados com as frequências correspondentes DTMF, soma os sinais e transmite através do pino 11 (RC0), o sinal vai para um transistor que aciona o dispositivo de som, mas vai também para o mesmo pino no microcontrolador U2. O microcontrolador U2 está executando uma rotina, cuja a visão geral é dada pelo fluxograma a seguir, após fazer a amostragem do sinal, o MCU U2 computa as frequências de interesse do espectro, por meio do algoritmo de Goertzel, depois realiza uma filtragem, basicamente copiada do programa em C#, e exibe no display LCD o

valor correspondente ao sinal. Assim como no caso do U1, há um terminal serial para fins de debug e acompanhamento do funcionamento no U2.



Os códigos que estão a serem executados em ambos os microcontroladores estão disponíveis no Github, juntamente com todo o resto dos arquivos utilizados. Acredita-se que após ler este tutorial é fácil compreender os códigos.

8.3 Consideração final

Eventualmente estarei atualizando este arquivo, toda sugestão é bem vinda. No caso de encontrar algum erro, qualquer dúvida ou sugestão me contate em

jcf_ssp@hotmail.com

Referências:

[1] FERREIRA, J. C.; **Desempenho comparativo das Transformadas de Fourier e Hartley e aplicação embarcada para análise de vibrações mecânicas.** 2016. 113 f. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) – Universidade de Franca, Franca.

[2] HSU, H. P. **Análise de Fourier.** Rio de Janeiro: L.T.C., 1972.

[3] FIGUEIREDO, D. G. **Análise de Fourier e equações diferenciais parciais.** 4. ed. Rio de Janeiro: IMPA, 2007.

[4] HSU, H. P. **Sistemas e Sinais**. 2ª Edição. São Paulo: Bookman Companhia editora Ltda, 2012.

[5] BLAHUT, R. E.; **Fast Algorithms for Signal Processing.** Nova York: Cambridge University Press 2010.

[6] RAO, K.R.; KIM, D.N.; HWANG, J.J.; **Fast Fourier Transform: Algorithms and Applications.** Londres/Nova York: Springer Dordrecht Heidelberg, 2010.

[7] TOSCANI, L. V.; VELOSO, P. A. S. **Complexidade de algoritmos.** 3. ed. Porto Alegre: Bookman, 2012.

[8] STRANG, G. S. **Lec 26 | MIT 18.06 Linear Algebra, Spring 2005**. Vídeo de aula ministrada no Instituto tecnológico de Massachusetts (MIT). <https://www.youtube.com/watch?v=M0Sa8fLOajA> , Acesso em 14/03/2018.

[9] ARSLAN, G. **Dual-Tone Multiple Frequency (DTMF) Detector Implementation**. For EE382C, Embedded Software Systems, 1998. Disponível em: <http://users.ece.utexas.edu/~bevans/courses/ee382c/projects/spring98/arslan/finalreport.pdf >, Acesso em 14/03/2018.