UNIVERSIDADE FEDERAL DE SANTA MARIA – UFSM

CENTRO DE TECNOLOGIA – CT

PROCESSAMENTO DIGITAL DE SINAIS

ANÁLISE DE SINAIS DIGITAIS: A APLICAÇÃO DE UM FILTRO DIGITAL POR MEIO DO MÉTODO OVERLAP SAVE

Luis Felipe de Deus

Nathanael Jorge Luchetta

Tiago Knorst

SANTA MARIA, DEZEMBRO DE 2018

1) INTRODUÇÃO

Na área de engenharia muitas vezes o profissional se depara com o problema de análise de sinais, onde o sinal ao qual se está interessado está "mascarado" a outros sinais, impossibilitando a devida análise ou utilização do sinal proposto. A solução para este problema é a utilização de filtros, que irão, como o nome já sugere, separar as componentes indesejadas no sinal, deixando apenas o sinal que se está interessado.

Há uma enorme gama de modelos de filtros, analógicos ou digitais, cada qual com suas características particulares, neste trabalho será abordado o estudo de um filtro digital do tipo FIR (*Finite Impulse Response*).

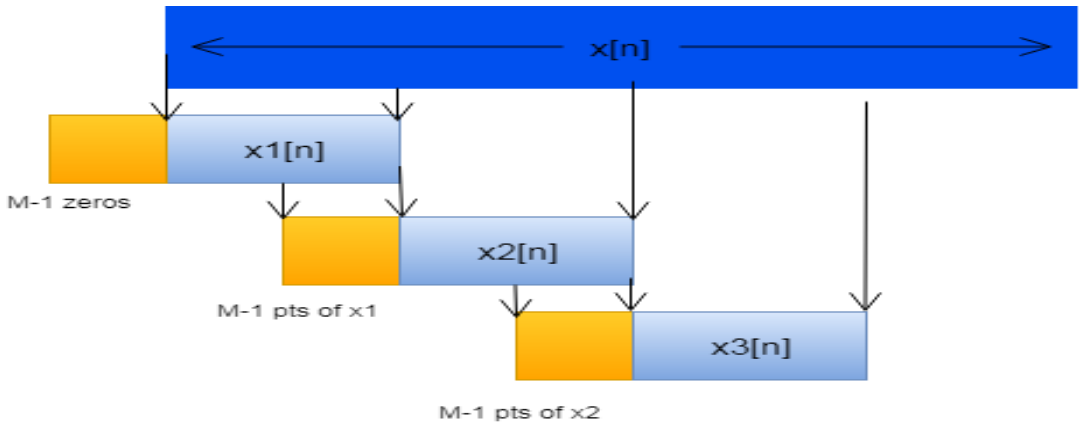
Entretanto, a maior parte dos sinais a serem analisados tem uma longa sequencia de dados, o que acarreta em um alto custo de processamento, ou seja, tempo de CPU, e também um alto uso de memoria para guardar estas informações. Foi então necessário o estudo de estratégias para tentar diminuir o custo de processamento dos sinais que contavam com uma grande sequencia de dados, para isso foram desenvolvidos dois métodos para amenizar o curso de computação, o método *Overlap Add* e o *Overlap Save*, neste trabalho será abordado e utilizado o método *Overlap Save* com o desenvolvimento de uma aplicação foi desenvolvida em Python, pois é de fácil compreensão, até mesmo para leigos.

2) MÉTODO OVERLAP SAVE

O método *Overlap Save* é utilizado para diminuir o processamento de um sinal, visto que a maioria dos sinais são extremamente grandes. O método utiliza a tática de quebrar o processamento em blocos, ao invés de processar o sinal todo de uma só vez.

Funciona da seguinte maneira, primeiramente a entrada "x[n]" é separada em sequencias de comprimento N = L+(M-1), sendo L o tamanho do bloco, como demonstra a Figura 1. Para o primeiro bloco os primeiros M-1 valores são preenchidos com zeros, para os seguintes são os M-1 valores do bloco anterior, é utilizado comumente M do mesmo tamanho do filtro "h[n]". Após é calculado a DFT (*Discrete Fourier Transform*) de todas as componentes xi[n] e do filtro h[n], gerando sinais "xi[k]" e "h[k]".

Figura 1 - Separação do Sinal em Blocos

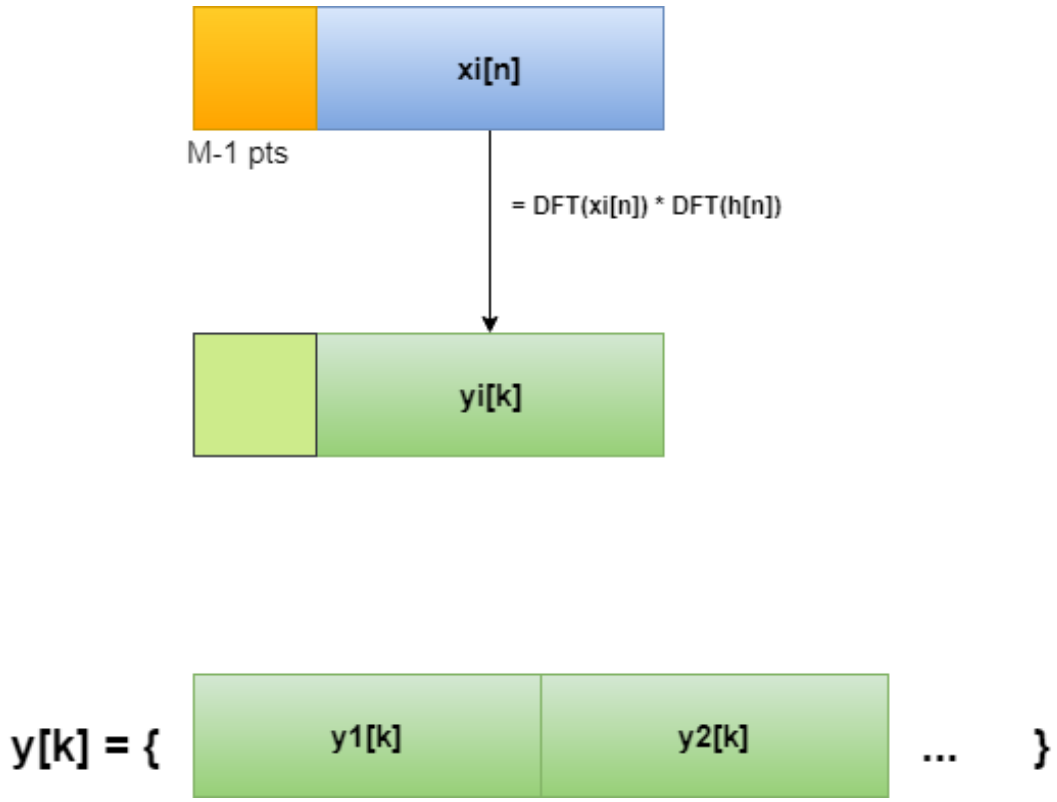


Fonte: Autores.

Por fim é efetuado o produto entre os sinais xi[k] e h[k], gerando as componentes

yi[k], e então é efetuado a concatenação das componentes yi[k] descartando os M-1 valores de todos as componentes, formando o resultado final y[k], como demonstra a Figura 2.

Figura 2 - Cálculo por blocos e concatenação dos blocos para o sinal de saída



Fonte: Autores.

O método *Overlap Save* implementado em Python está descrito abaixo, na Figura 3. Nota-se que as operações mais importantes para o correto funcionamento do método são:

1. Calcular a FFT do filtro;
2. Quebrar o sinal de entrada em N componentes;
3. Multiplicar a FFT de um bloco pela FFT do filtro;
4. Efetuar a FFT inversa da multiplicação do passo 3, para obter o resultado de um bloco do sinal de saída;
5. Grava em ordem os blocos calculados no passo 4;
6. Verifica se precisa ou não cortar alguns blocos para que o sinal de saída fique do mesmo tamanho que o sinal de entrada.;
7. Retorna o sinal de saída.

Figura 3 - Implementação em Python do Overlap Save

```python
#Overlap Save, Matriz
def overlap_save (signal_x, filter_h, range_L):
    global first_time
    global last_X
    #M = filter_h.size.
    M = filter_h.size
    #Define o número de iterações.
    N      = range_L + M - 1
    max_it = (signal_x.size // range_L) + 1
    #Preenche com zeros o filter_h para fazer com que ele tenha o mesmo tamanho que X_order.
    filter_h = np.append(filter_h, np.zeros(N-filter_h.size, dtype=filter_h.dtype))
    #Extende o sig com zeros ou com last_X para fácil indexação.
    if first_time == True:
        sigX_append = np.append(np.zeros(M - 1, dtype=signal_x.dtype), signal_x)
        first_time = False;
    else:
        sigX_append = np.append(last_X, signal_x)
    #O próximo last_X é o signal_x corrente.
    last_X = np.copy(signal_x[-(M - 1):])
    sigX_append = np.append(sigX_append, np.zeros(N - signal_x.size % range_L, dtype=signal_x.dtype))
    #Cria Y_store para armazenar cada Y_store[0], Y_store[1], ...
    Y_store = np.ndarray(shape=(max_it, N), dtype=signal_x.dtype)
    #Faz a FFT do filter_h.
    h_fft = np.fft.fft(filter_h)
    #Iteração para obter Y_order.
    for order in range(0, max_it):
        #Pega X_order.
        X_order = signal_n_generator(sigX_append, order, range_L, N)
        #Multiplica as frequências no domínio do tempo.
        Mult_fft = np.multiply(np.fft.fft(X_order), h_fft)
        #Gera o Y_order para a frequência do tempo.
        Y_order = np.fft.ifft(Mult_fft)
        #Salva Y_order para juntar futuramente.
        Y_store[order] = np.around(Y_order.real).astype(signal_x.dtype)
    #Junta cada Y_store em um y_result.
    y_result = merge_n_results(Y_store, M, cut=True, size_X=signal_x.size)
    return y_result
```

Fonte: Autores.

3) DESENVOLVIMENTO PRÁTICO

A proposta do trabalho consistia em projetar um filtro digital do tipo FIR, e aplica-lo a uma entrada x[n] qualquer utilizando o método *Overlap Save*. Para gerar o filtro, fez-se o uso da ferramenta FDATOOL, nativa do software Matlab, onde é possível exportar os coeficientes do filtro como um *array*, facilitando a etapa de codificação em Python da aplicação.

Para projetar os filtros utilizados, antes pensou-se um cenário onde os mesmos pudessem ser aplicados e os resultados de saída fossem nítidos. Tomou-se então a decisão de aplicar o filtro em uma faixa de áudio *.wav*.

Com isso, foram gerados três filtros:

- Passa baixa com frequência de corte em 300Hz
    - Com o intuito de reter somente as baixas frequências;
- Passa alta com frequência de corte em 400Hz
    - Com o intuito de retirar as frequências graves da música;
- Passa banda com frequência de corte em 500Hz e 2000Hz
    - Com o intuito de atuar como um Vocal Remover.

Após projetados os filtros, partiu-se para codificação da aplicação em Python, onde recebe-se um arquivo de áudio *.mp3*, converte-se em *.wav* para a obtenção de um *array* da faixa de áudio.

Com o sinal de entrada em formato de *array* o filtro pôde ser aplicado, por meio

do método *Overlap Save "**y_result = overlap_save(signal_x, filter_h, L)**"*, onde o sinal filtrado é atribuído ao ***y_result*** para que possa ser reconstruído futuramente. O método do *Overlap Save* foi implementado conforme explicado anteriormente, sendo importante notar que *L* possui o comprimento do vetor de filter_h.

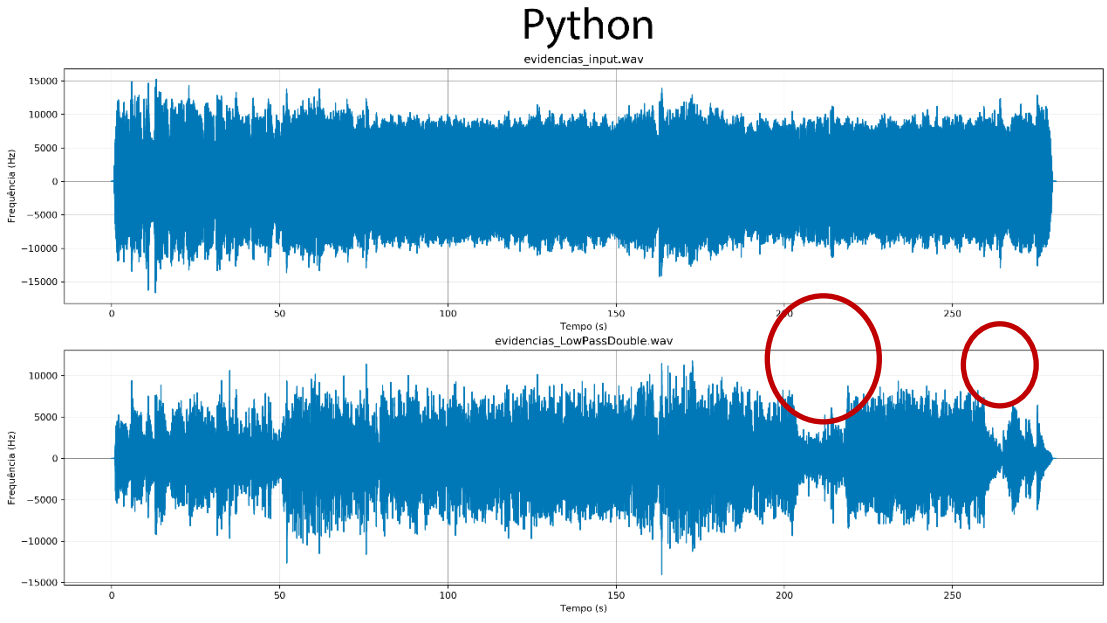
Após a conversão, o sinal filtrado é exportado para um arquivo *.wav*, com taxa de amostragem igual ao dobro da taxa do sinal original, isso devido ao **Teorema da amostragem de Nyquist–Shannon** que basicamente diz, "Se um sinal arbitrário é transmitido através de um canal de largura de banda B *Hz*, o sinal resultante da filtragem poderá ser completamente reconstruído pelo receptor através da amostragem do sinal transmitido, a uma freqüência igual a, no mínimo 2B vezes por segundo.".

No caso dos vetores de teste utilizados, a música Back In Black – ACDC e também Evidências – Chitãozinho e Xororó, possuíam uma taxa de amostragem de 44100 Hz, ou seja, o sinal filtrado foi exportado para o *.wav* tendo sua taxa de amostragem igual a 88200 Hz.

O interessante da linguagem Python é a infinidade de recursos que ele traz, como por exemplo a possibilidade de plotar gráficos facilmente, utilizando a biblioteca *matplotlib*, pois com ela é possível plotar-se gráficos em tela. Portanto, foi possível plotar o espectro do sinal original e final, por meio da FFT, um seguido do outro para facilitar a visualização da atuação do filtro.

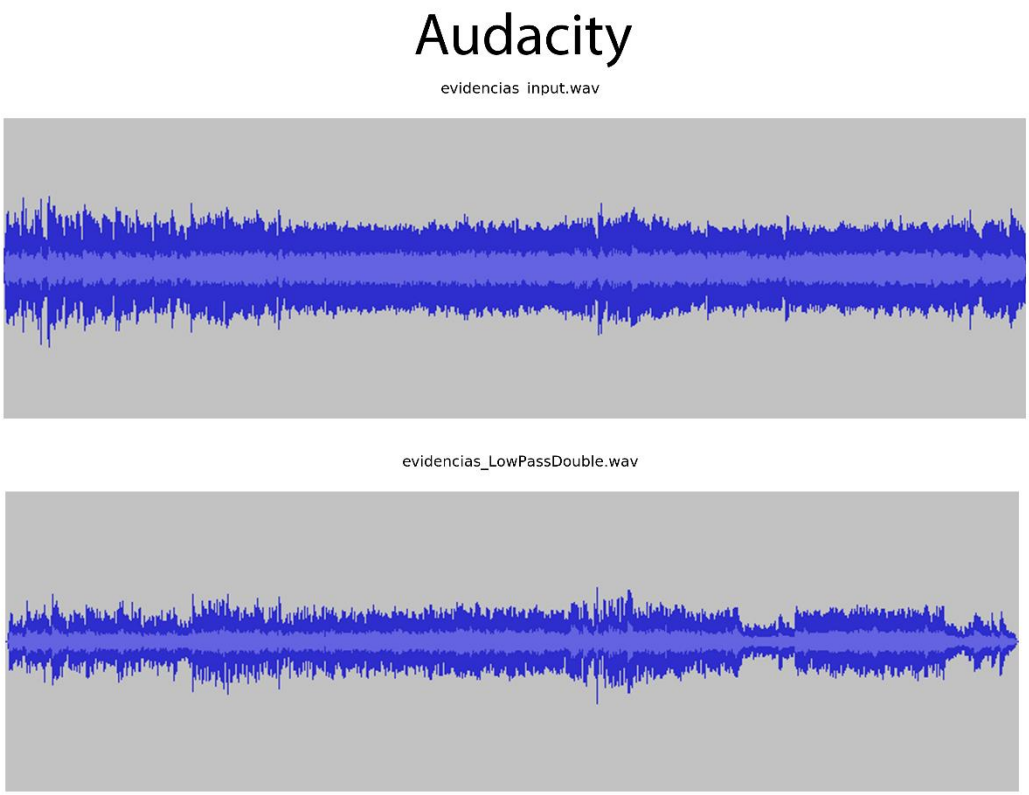
Além disso, utilizou-se o software de áudio *Audacity*, que possuí diversas ferramentas, como a aplicação de filtros, para se comparar a eficácia do filtro por meio do método *Overlap Save*, como pode ser visualizado abaixo, na Figura 4 e na Figura 5.

Figura 4 - Espectro dos sinais de entrada e saída obtidos a partir do Python



Fonte: Autores.

Figura 5 - Espectro dos sinais de entrada e saída obtidos a partir do Audacity



Fonte: Autores.

4) CONCLUSÃO

O trabalho proposto teve como finalidade explorar o funcionamento de um filtro FIR e do método Overlap Save, neste último caso, nota-se que o método tem um grande potencial a ser explorado no que se diz respeito ao quesito desempenho, pois o cálculo é feito de forma parcial, ou seja, quebrado em blocos. Portanto, pode-se realizar uma exploração a nível de threads, ou seja, executar o cálculo de forma paralela, com um adendo muito importante, cuidando o retorno, para que não grave no sinal os blocos fora de ordenação.

Ainda quanto ao Overlap Save, o mesmo apresenta funcionamento diferente do Overlap Add, pois não faz o mesmo preenchimento com zeros, mas, em vez disso, reutiliza valores do intervalo de entrada anterior.

Além disso, após análise com diferentes tipos de janela de filtro, a que se mostrou melhor foi a janela Kaiser, pois oferece uma maior atenuação.

Ante o exposto, conclui-se que o trabalho proposto foi de extrema e fundamental importância tanto para o desenvolvimento pessoal, quanto para o desenvolvimento profissional, além de mostrar o quão fascinante é a manipulação de sinais e áudio.

REFERÊNCIAS

[1] Elder R. **Overlap Add, Overlap Save Visual Explanation**. Disponível em: <http://blog.robertelder.org/overlap-add-overlap-save/> Acessado em: 29 nov. 2018

[2] **Matplotlib**. Disponível em: <https://matplotlib.org/> Acessado em: 29 nov. 2018

[3] **SciPy**. Disponível em: <https://www.scipy.org/> Acessado em: 29 nov. 2018