Universidade Federal de Pernambuco
Centro de Informática - CIn

Relatório de Projeto

Rebeca Paula Alves de Oliveira

Processamento Digital de Sinais
Profº Carlos Alexandre Barros de Mello

Recife, 10 de junho de 2019

Sumário

1 - Sinais

Questão 1.1 - Filtro FIR

Para projetar o Filtro FIR Passa Alta com Ws = 0.6π, Wp = 0.75π e A=50 dB foram utilizadas como base as funções *ideal_lp* e *freqz*, retiradas das Notas de Aula [1] implementando suas versões em Python, com auxílio das bibliotecas *Math*, *Matplotlib*, *Numpy* e *Scipy*, e fazendo as alterações necessárias para atender as especificações do projeto.

A função *ideal_lp* implementa a resposta ao impulso ideal para um Filtro FIR Passa Baixa assim, para conseguir obter a resposta ao impulso ideal de um Filtro FIR Passa Alta foi necessário alterar a resposta ao impulso através da técnica de Inversão Espectral que consiste em inverter o valor de todas as amostras e adicionar 1 ao valor central. Na Figura 1.1.1 estão os códigos da função *ideal_lp* e da função alterada *ideal_hp*. Uma vez alterada a resposta ao impulso, a função *freqz* foi apenas transcrita para Python, como pode ser visto na Figura 1.1.2.

```
function hd = ideal_lp(wc, M)
% Ideal low pass filter
% wc=cutoff frequency
% M=length of the ideal filter

alpha = (M - 1)/2

n = [0:(M-1)];

m = n - alpha + eps;

hd = sin(wc*m)./(pi*m);
```

(a)

```
def ideal_hp(wc, M):
    ## Filtro Highpass Ideal:
    ## wc = frequência de corte
    ## M = comprimento do filtro ideal
    alpha = (M - 1)/2
    n = arange(0,M)
    m = [item - alpha + spacing(1) for item in n]

## inversão espectral: inverte todas as amostras e
## soma 1 a amostra central
    hd = [-sin(wc*item)/(pi*item) for item in m]
    hd[int(len(hd)/2)) += 1
    return hd
```

(b)

Figura 1.1.1 - Funções (a) ideal_lp em Matlab e (b) ideal_hp em Python.

```
function [db, mag, pha, w] =
freqz_m(b, a)
% Versao modificada da funcao
%freqz
[H,w]=freqz(b,a,1000,'whole');
H = (H(1:501))';
w = (w(1:501))';
mag = abs(H);

db = 20*log10((mag +
eps)/(max(mag)));

pha = angle(H);
```

(a)

```
def freqz_m(b, a):
  ## Versao modificada da funcao freqz

  [w, H] = freqz(b,a,1000,whole=True)
  H = H[:501]
  w = w[:501]
  mag = abs(H)

  maxMag = max(mag)
  db =[20*log10((i +
      spacing(1))/maxMag)for i in mag]

  pha = angle(H)
```

(b)

Figura 1.1.2 - Funções (a) freqz_m em Matlab e (b) freqz_m em Python.

O janelamento escolhido foi Hamming, porque a janela de Hamming fornece uma atenuação de mais de 50 dB e possui uma ordem menor porque tem uma banda de transição

menor. Como essa janela possui uma largura de lóbulo principal larga, a convolução no domínio da frequência é mais suave fazendo com que a região de transição na resposta do filtro FIR seja mais larga.

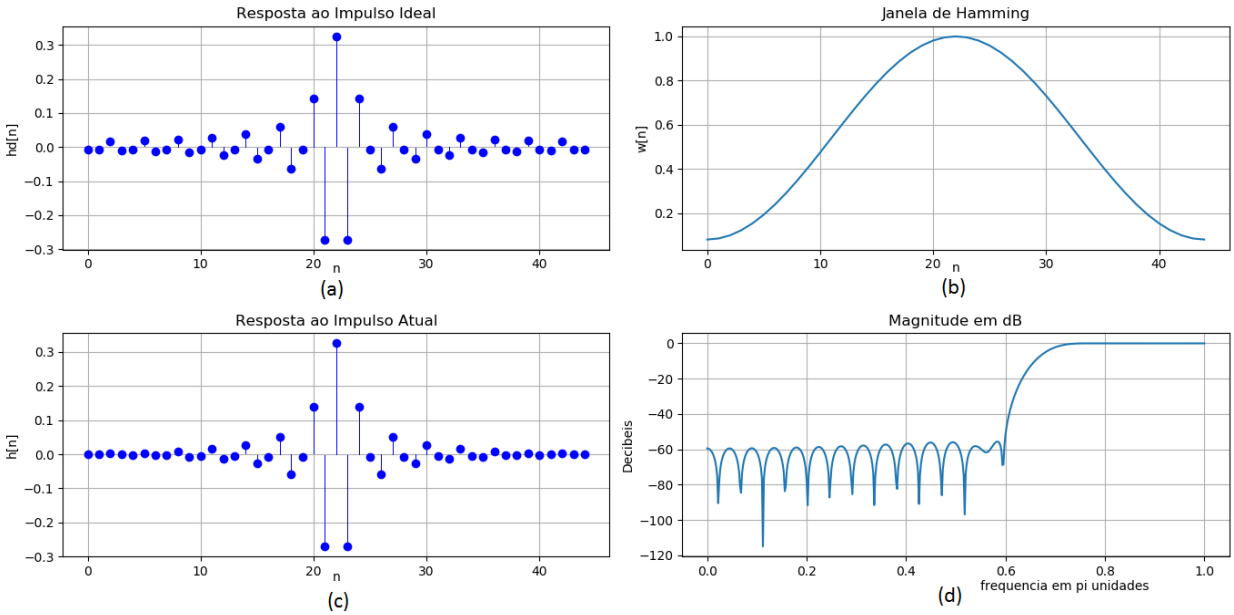


Figura 1.1.2 - (a) Resposta ao impulso ideal, (b) janela de Hamming, (c) Resposta ao impulso atual e (d) magnitude em decibéis do filtro.

As respostas ao impulso ideal e atual podem ser vistas, respectivamente, nas Figuras 1.1.3.a e 1.1.3.c. A janela de Hamming e a magnitude podem ser visualizadas nas Figuras 1.1.3.b e 1.1.3.d. Os valores resultantes foram M = 45, alpha = 22, Rp = 0.03813 dB e As = 52 dB.

Questão 1.2 - Adição de eco e aplicação de filtro reverberador

A primeira etapa consiste em utilizar o sinal de áudio ‘sp04.wav’ como entrada de um sistema que adiciona eco com um delay D=500 ms e com um fator de ampliação de 0.5, ou seja, utiliza o sinal de entrada, x[n] para criar o sinal x2[n], que é representado pela equação 1.2.1 abaixo. A implementação foi feita em Python 3.7 e foram utilizadas as bibliotecas: *AudioOp, Matplotlib, Numpy, Playsound* e *Wave*.

$$[1.2.1] \quad x2[n] = x[n] + 0.5x[n - D]$$

Para conseguir gerar o sinal x2[n], o sinal x[n] é adicionado a uma versão multiplicada por um fator de ampliação de 0.5 atrasada em D=500 ms de si mesmo. O resultado desta etapa pode ser visto da Figura 1.2.2, que mostra a forma de onda do sinal resultante.

O efeito de reverberação é conseguido através da convolução do sinal de entrada com a resposta ao impulso do mesmo sinal filtrado. Gerando as formas de onda que estão exibidas a seguir.

As imagens a seguir mostram os resultados com a = {0.5, 0.9, 0.25} e D = 500, para o filtro cuja função transferência é dada por

$$[1.2.2] \quad H(z) = \frac{1}{1 - az^{-D}}$$

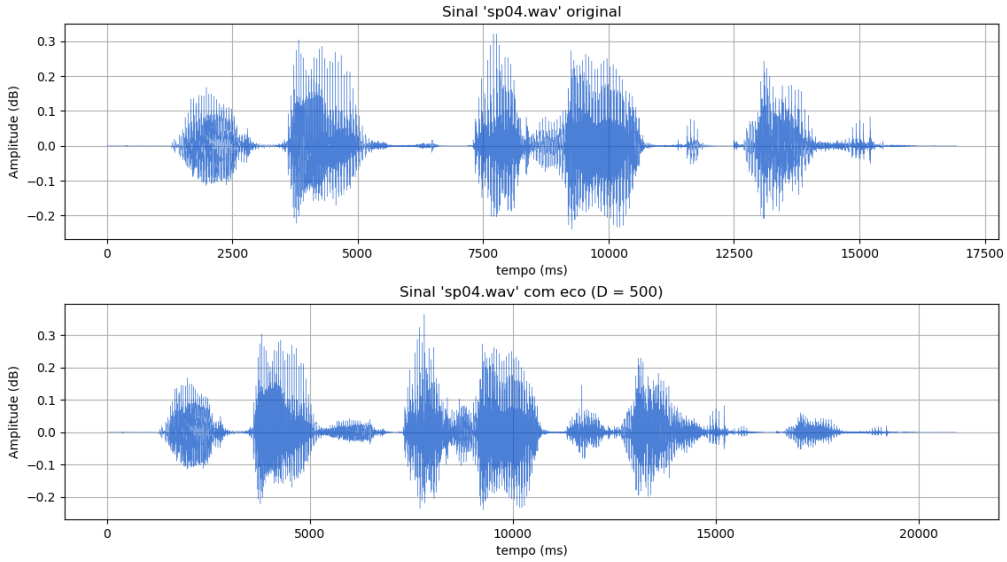


Figura 1.2.1 - Formas de onda dos sinais (a) x[n] e (b) x2[n] com a = 0.5.

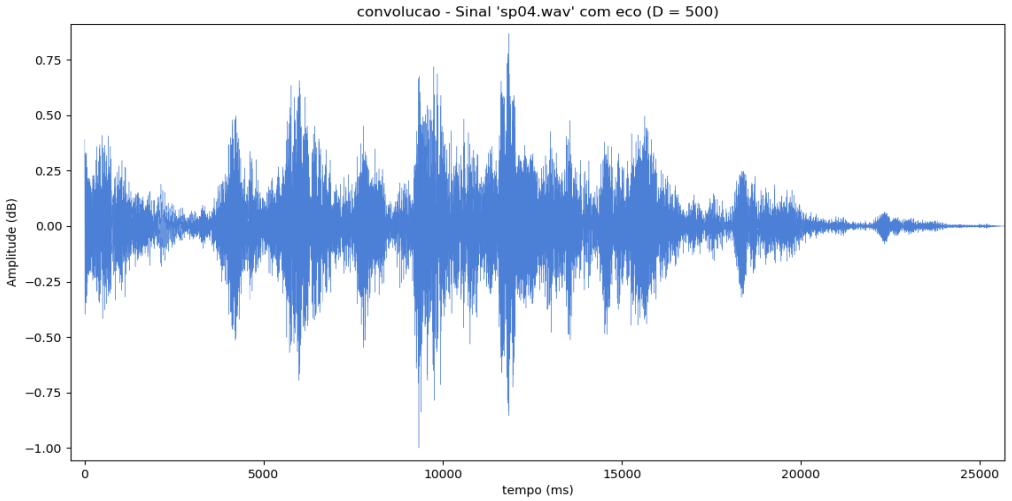


Figura 1.2.2 - Sinal resultante aplicação do filtro reverberador com a = 0.5.

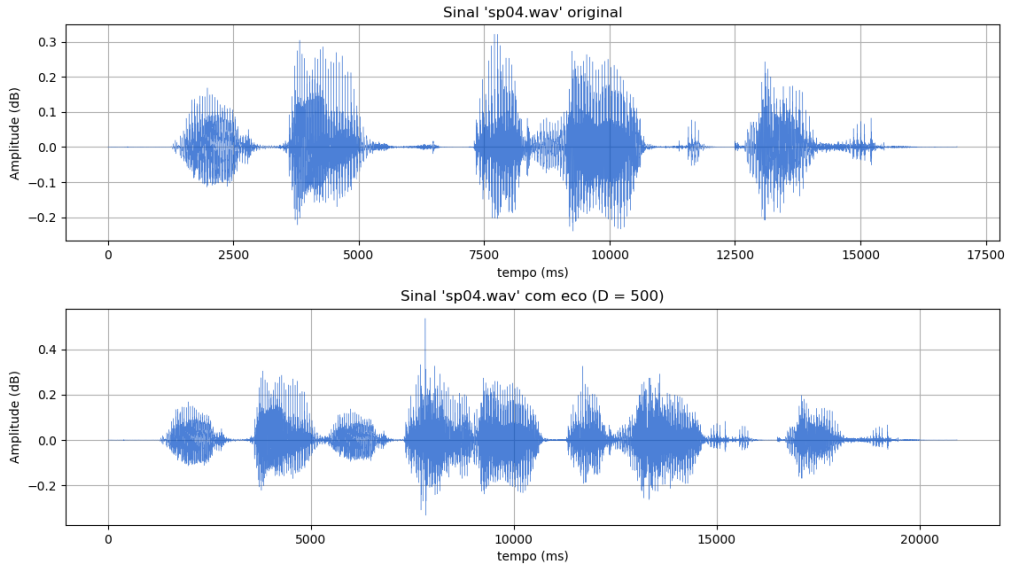


Figura 1.2.3 - Formas de onda dos sinais (a) x[n] e (b) x2[n] com a = 0.9.

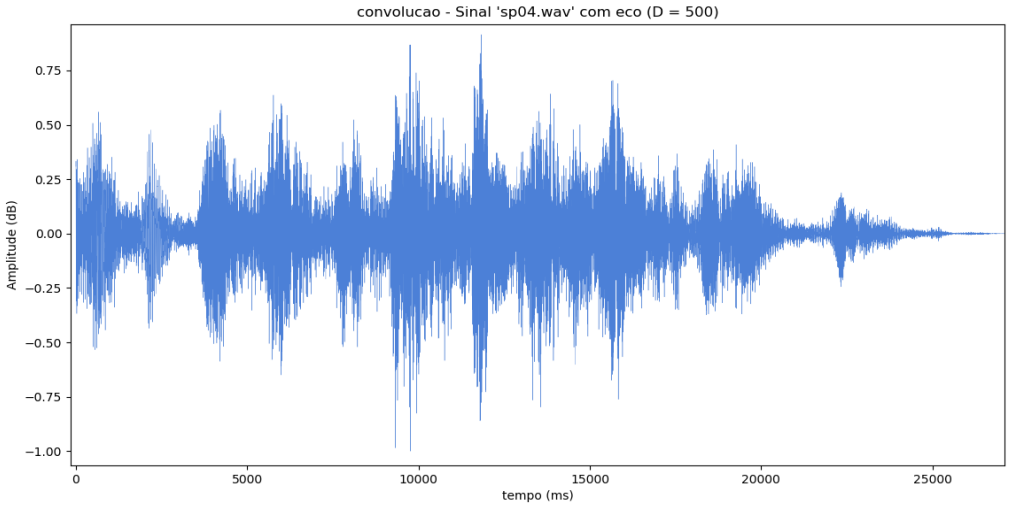


Figura 1.2.4 - Sinal resultante aplicação do filtro reverberador com a = 0.9.

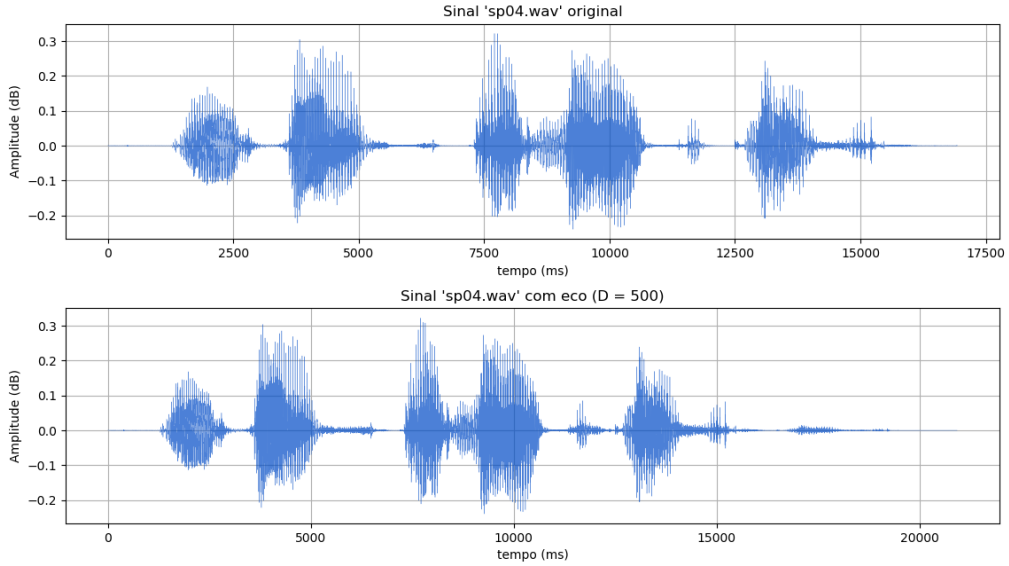


Figura 1.2.5 - Formas de onda dos sinais (a) x[n] e (b) x2[n] com a = 0.25.

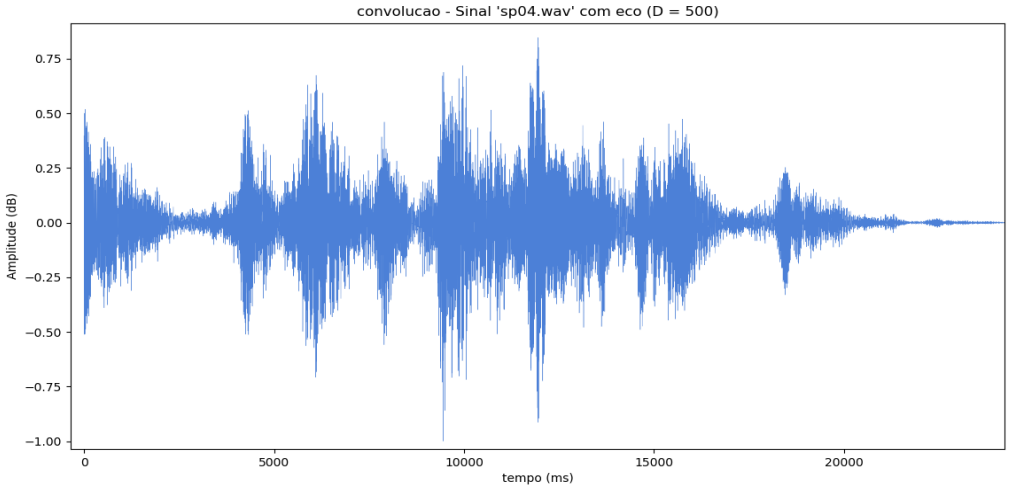


Figura 1.2.6 - Sinal resultante aplicação do filtro reverberador com a = 0.25.

2 - Imagens

Questão 2.1 - Redução do efeito *ringings*

Para diminuir o efeito de *ringings* em uma imagem é necessário aplicar um Filtro Passa Baixa (FPB). O FPB permite a passagem de de baixas frequências que causa um efeito de embaçamento nas imagens, porque ele atenua áreas com mudanças bruscas de pixels e mesmo que não consiga eliminar o efeito de *ringings*, um FPB será capaz de atenuar o ruído.

Foram testados três filtros com implementação em Python: os filtros *BoxBlur* e *GaussianBlur* da biblioteca *Pillow* (PIL) e o filtro Gaussian_filter da biblioteca *Scipy*.

BoxBlur o FPB Box que utiliza o valor médio dos pixels em uma janela quadrada e recebe como parâmetro o raio de extensão dos pixels. Os filtros *GaussianBlur* e *Gaussian_filter* e utilizam o filtro Gaussiano, que recebe o paramêtro sigma que é o desvio padrão mínimo. Para os três filtros, quanto maior o parâmetro de entrada, maior a suavização e mais borrada ficará a imagem.

As Figuras 2.1.1, 2.2.2 e 2.2.3 mostram, respectivamente, os resultados para os três filtros com Raio = Sigma = {0.3, 0.7, 1.1, 1.5, 1.9}.

(a) (b) (c) .

Figura 2.1.3 - Resultado da aplicação do filtro para sigma=radius= 0.3 com os filtros (a) Boxblur, (b) Gaussianblur e (c)Gaussian_filter.

(a) (b) (c) .

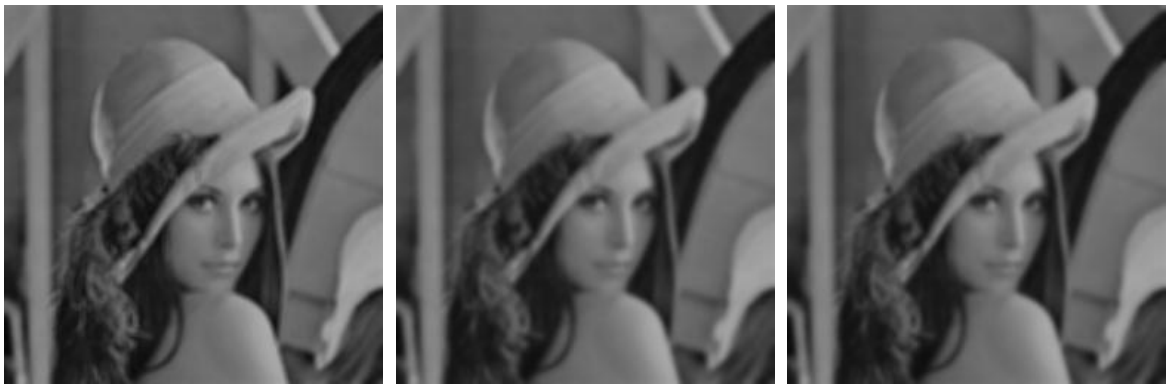
Figura 2.1.3 - Resultado da aplicação do filtro para sigma=radius= 0.7 com os filtros (a) Boxblur, (b) Gaussianblur e (c)Gaussian_filter.

(a) (b) (c)

Figura 2.1.3 - Resultado da aplicação do filtro para sigma=radius= 0.7 com os filtros (a) Boxblur, (b) Gaussianblur e (c)Gaussian_filter.

(a) (b) (c)

Figura 2.1.3 - Resultado da aplicação do filtro para sigma=radius= 1.1 com os filtros (a) Boxblur, (b) Gaussianblur e (c)Gaussian_filter.



(a) (b) (c)

Figura 2.1.3 - Resultado da aplicação do filtro para sigma=radius= 1.9 com os filtros (a) Boxblur, (b) Gaussianblur e (c)Gaussian_filter.

Comparando o resultado de cada filtro entre si, não há diferença visível considerável. Para os três, percebemos que o resultado final de suavização do efeito de *ringings* se torna mais acentuado com o aumento dos respectivos parâmetros de entrada.

O parâmetro Raio=Sigma=1.1 foi o que apresentou o melhor resultado pois conseguiu suavizar o efeito de *ringings* sem perda acentuada de nitidez como aconteceu nos casos de Raio = Sigma = {1.5, 1.9}. Já para Raio = Sigma = {0.3, 0.7} a suavização foi mais sutil, sendo que o efeito de *ringings* continuou sendo visível.

Questão 2.2 - Contagem automática regiões com diferentes texturas

A imagem alumgrns.bmp (ver Figura 2.2.1) possui regiões com diferentes texturas e para contar quantas texturas existem é necessário que o algoritmo consiga detectar mudanças mais bruscas na coloração dos pixels, que correspondem às bordas que separam as regiões, assim, a primeira etapa do algoritmo consiste aplicar o filtro Sobel de detecção de bordas na imagem original e gerar uma nova imagem. A implementação foi feita em Python 3.7 e foram utilizadas as bibliotecas: *ImageIO, Math, Numpy, OpenCV(cv2), Pillow(PIL)* e *Scipy* .

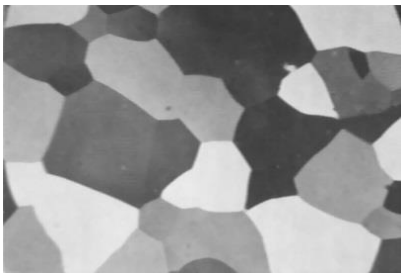


Figura 2.2.1 - imagem original alumgrns.bmp.

Realizar a detecção de bordas permite que a contagem de texturas ignore variações internas de cada região, o que poderia interferir no resultado final. Para realizar a detecção de bordas na imagem original foram testadas duas formas, a primeira utilizando a imagem original e a segunda com redução de ruídos aplicando o filtro gaussiano, implementado pela biblioteca Scipy (*scipy.ndimage.filters.gaussian_filter*), na imagem original. Para o caso com aplicação do filtro gaussiano foram realizados testes com $\boldsymbol{\sigma}$ = {0.1, 0.5, 1.0, 1.5, 2.0}. Os resultados da detecção de bordas para os dois casos podem ser vistos na Figura 2.2.1.

Para a contagem das regiões, foi aplicada busca em largura. Para cada novo nó visitado é verificado se sua cor é diferente da cor da borda, caso seja, o número de texturas é incrementado e os vizinhos mais próximos (os nós imediatamente acima, abaixo, à esquerda e à direita) são visitados em seguida, fazendo a mesma análise de cor, passando então para o próximo nível de vizinhos. Os resultados obtidos podem ser vistos na Tabela 2.2.2.

Durante os testes, ficou claro que o algoritmo de contagem é muito suscetível a ruídos, por isso, para considerar que uma determinada quantidade de pixels seja realmente uma área de textura e incrementar a contagem foi inserido um parâmetro, chamado *rangeArea*, que define um tamanho mínimo para que uma região de pixels com cor diferente da cor da borda seja considerado uma nova textura, caso essa região seja menor que esse limiar, a região é considerada como um ruído e ignorada na contagem. O valor utilizado para o range foi de 50.

Como pode ser visto na Tabela 2.2.1, quando não há aplicação do filtro gaussiano e quando $\boldsymbol{\sigma}$ = 0.5, $\boldsymbol{\sigma}$ = 1.0 ou $\boldsymbol{\sigma}$ = 1.5, o número de texturas contabilizados apresentam resultados próximos enquanto que $\boldsymbol{\sigma}$ = 2.0 obteve um resultado bem menor que os outros. Com isso, pode-se concluir que quanto maior a redução de ruídos, ou seja, quanto maior for o $\boldsymbol{\sigma}$, menor a quantidade de texturas contabilizadas.

Analisando as imagens na Figura 2.2.2, conclui-se que essa redução ocorre pela fusão de regiões com texturas mais semelhantes, ou seja, onde as bordas não apresentam um gradiente de variação alto, aumentando o erro na contagem de texturas.

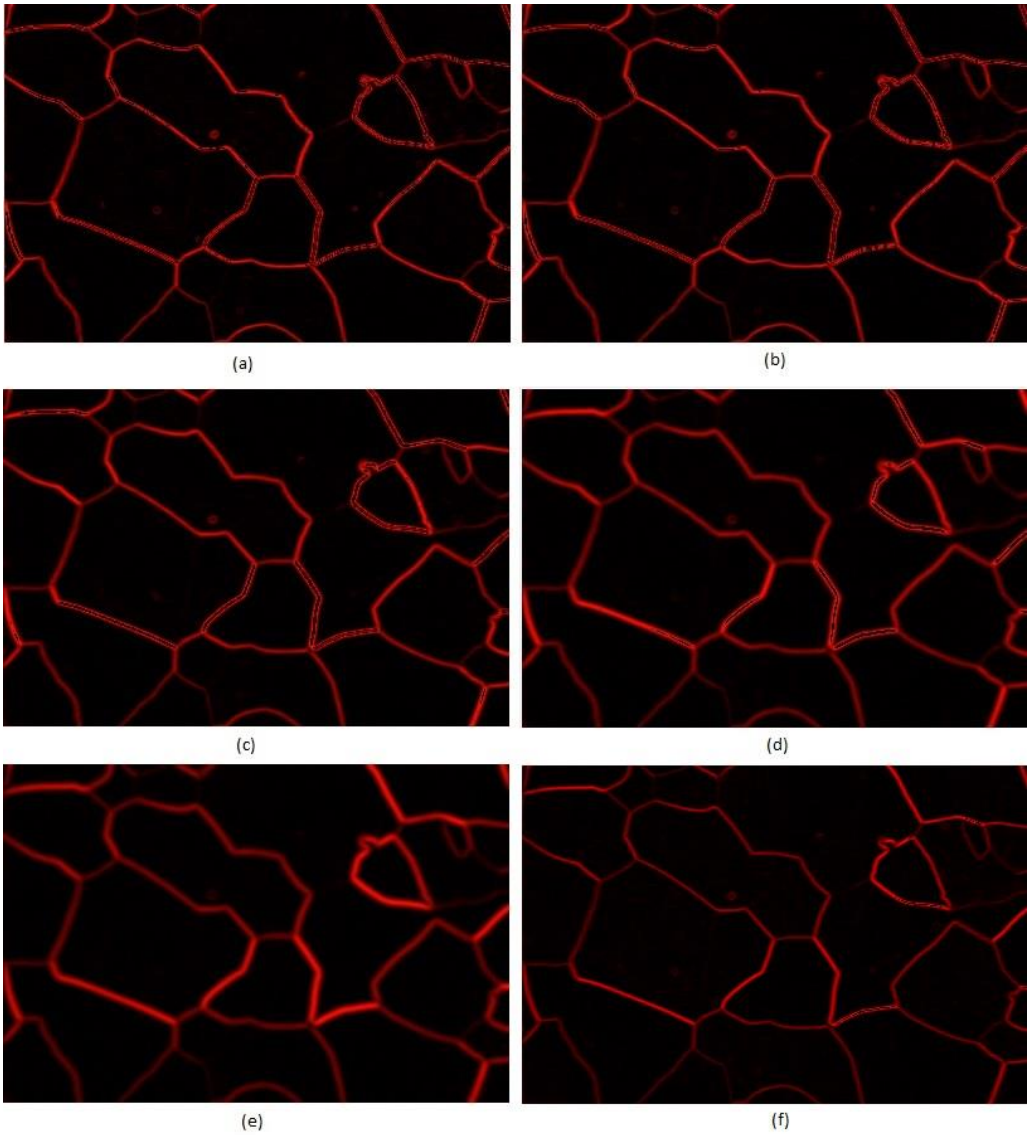


Figura 2.2.2 - Resultado da suavização de ruídos para (a) σ = 0.1, (b) σ = 0.5 (c) σ = 1.0, (d) σ = 1.5, (e) σ = 2.0 e (f) sem suavização de ruído.

Tabela 2.2.1 - Resultados obtidos após a execução do algoritmo

	Sem redução de ruído	Com redução de ruído			
		σ = 0.5	σ = 1.0	σ = 1.5	σ = 2.5
Quantidade de Texturas	22	22	21	20	16

Questão 2.3 - Melhorar a distinção dos números para daltônicos

No tipo de daltonismo mais comum, a pessoa tem uma redução ou completa ausência do pigmento vermelho, que pode ser confundido com o verde. Outro tipo de daltonismo consiste na redução ou incapacidade total de enxergar o verde e o terceiro tipo, mais raro, a

incapacidade está em enxergar corretamente os tons de azul, que é confundido com o verde. A implementação foi feita em Python 3.7 utilizando a biblioteca *Pillow(PIL)*.

Para que um daltônico consiga identificar o número contido na imagem 'dalton.bmp' (ver Figura 2.3.1.a) é necessário realçar o contraste da imagem para separar as cores. Para conseguir esse efeito foi utilizada a operação de *Stretch*, uma operação linear que altera a intensidade dos pixels utilizando a equação de normalização, abaixo

$$[2.3.1]\ Iout = (Iin - MINin) \times (((MAXout - MINout)/(MAXin - MINin)) + MINout)$$

onde *Iout* é a intensidade do pixel resultante da normalização, *Iin* é a intensidade original do pixel, *MINin* é o menor valor da intensidade do pixel original, *MINout* é o menor valor da intensidade do pixel resultante, *MAXin* é o maior valor da intensidade do pixel original e *MAXout* é o maior valor da intensidade do pixel resultante da normalização.

Para realizar a operação de *stretch*, primeiro é necessário separar as bandas RGB da imagem. Após a separação, cada banda é normalizada individualmente aplicando a equação 2.3.1. Depois é feita a combinação das resultantes de normalização de cada banda RGB.

Os valores de *MINout* e *MAXout* para todas as bandas foram respectivamente, 0 e 255 e os valores de *MINin* e *MAXin* foram definidos individualmente para cada banda RGB utilizando o método de experimentação e podem ser vistos na Tabela 2.3.1. E as imagens obtidas podem ser vistas na Figura 2.3.1.

Os parâmetros utilizados na operação de *stretch* foram eficazes em intensificar o contraste causando uma separação das cores, como mostram as Figuras 2.3.1.b e 2.3.1.c, permitindo que os algarismos escondidos na imagem original possam ser identificados por pessoas com daltonismo.

Tabela 2.3.1 - Parâmetros da operação de *stretch* para cada banda RGB

	Teste 1		Teste 2	
Banda	MINin	MAXin	MINin	MAXin
Vermelho (R)	86	230	86	230
Verde (G)	0	255	100	230
Azul (B)	200	210	200	210

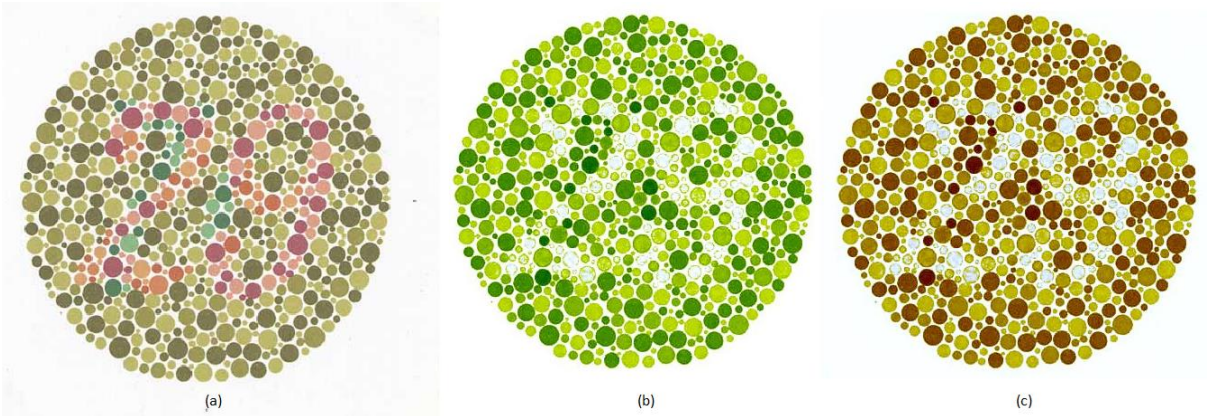


Figura 2.3.1 - (a)Imagem original e imagem após aplicação de *stretching para o (b) teste 1 e para o (c) teste 2*.

3 - Vídeos

Questão 3.1 - Geração de vídeo apenas com os carros que passam

O arquivo de vídeo ‘video1.avi’ consiste de uma cena de tráfego onde carros circulam num cruzamento e o desafio é remover todo o *background* e produzir um vídeo que mostre apenas os carros.

A técnica aplicada consiste em realizar uma subtração de *background* frame a frame, para que permaneçam apenas os elementos que são alterados de um frame para o outro. Como o vídeo não possui elementos em movimento além dos carros e não há mudanças relevantes no cenário, como alteração de iluminação, realizar a subtração de background é uma técnica satisfatória, caso contrário, uma técnica mais sofisticada poderia seria necessária.

Utilizando Python 3.7 e com auxílio das bibliotecas *Numpy* e *OpenCV* (cv2), o algoritmo percorre o vídeo frame a frame e calcula a diferença absoluta entre o frame atual e o frame imediatamente anterior, essa diferença gera um frame que corresponde visualmente apenas às mudanças que ocorreram de uma cena para outra. Foram realizados dois testes, um com o vídeo original e um aplicando o filtro *Gaussian Blur*, que é uma técnica de redução de ruído fornecida pela cv2.

Figura 3.1.1 - Frame (a)original, (b) com subtração de background sem redução de ruído e (c) com subtração de background e com redução de ruído no instante igual a 2 segundos

Figura 3.1.1 - Frame (a)original, (b) com subtração de background sem redução de ruído e (c) com subtração de background e com redução de ruído no instante igual a 10 segundos.

Como a cena não possui grande complexidade quando se trata de elementos em movimento, a subtração de *background* é eficaz e consegue gerar um vídeo contendo apenas a silhueta dos carros. A diferença entre utilizar uma redução de ruído está na suavização do background, já que o ruído faz com que pixels do *background* continuem aparecendo no vídeo

final, porém a perda de nitidez também diminui a capacidade de detectar alguns carros mais afastados da câmera. Portanto, se o foco é identificar os carros na cena, não aplicar suavização de ruído é mais interessante.

4 - Voz e Som

Questão 4.1 - Remoção do som do carro

Para remover o ruído causado pelo carro, um filtro FIR foi implementado utilizando as bibliotecas *Scipy*, *Numpy*, *Soundfile* e *Matplotlib*.

Foram utilizadas duas operações de janelamento: Kaiser e Hamming. Para a janela de Kaiser, os parâmetros foram calculados pela função *scipy.signal.kaiserord*, que retorna o valor de **β** e o número de coeficientes.

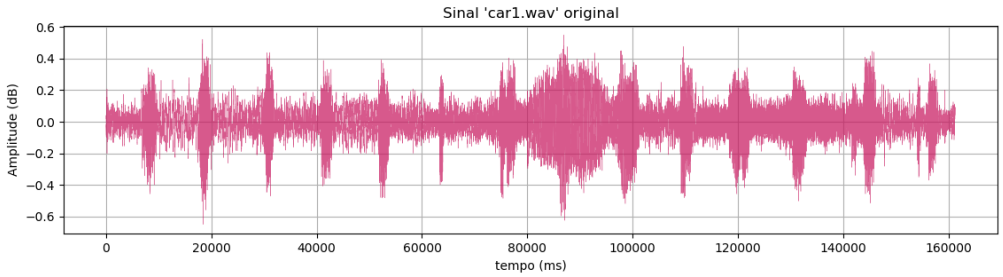


Figura 4.1.1 - Sinal de entrada ‘car1.wav’

Os coeficientes do filtro FIR foram calculados pela função *scipy.signal.firwin*, recebendo como parâmetros o número de coeficientes, a frequência de corte e a janela, com seus respectivos parâmetros.

Tabela 4.1.1 - Parâmetros de filtragem

Variável	Valor	Significado
número de amostras (nsamples)	161227	comprimento do sinal de entrada
taxa de amostragem (sample_rate)	11025	taxa de amostragem do sinal de entra
taxa de Nyquist (nyq_rate)	5512.5	taxa de amostragem / 2.0
tamanho da janela (width)	9,07.10^-4	0.5 / taxa de Nyquist
atenuação de kaiser (ripple_db)	60 dB	60.0 db
frequência de corte	3000/nyq_rate = 0.75, 1000/nyq_rate = 0.25, 500/nyq_rate = 0.125	

A função *scipy.signal..lfilter* foi utilizada para aplicar o filtro no sinal de entrada ‘car1.wav’. A seguir estão os resultados obtidos pela aplicação do filtro FIR gerado.

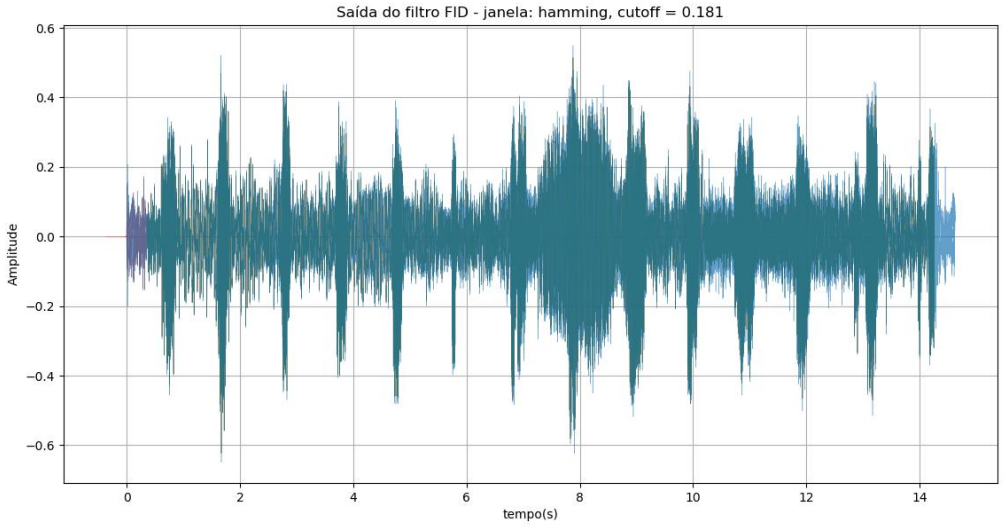


Figura 4.1.2 - Resposta do filtro FIR com janela Hamming e frequência de corte 0.181 Hz

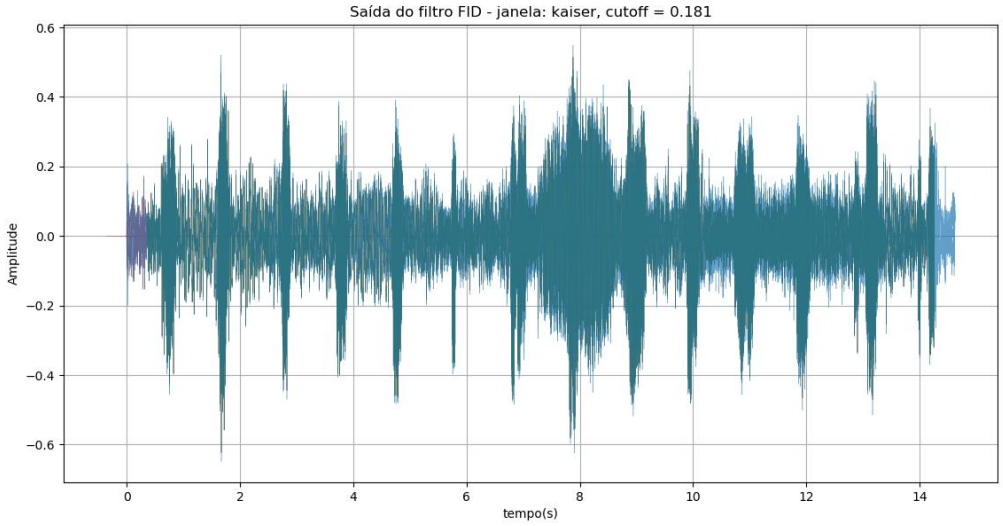


Figura 4.1.3 - Resposta do filtro FIR com janela Kaiser e frequência de corte 0.181 Hz

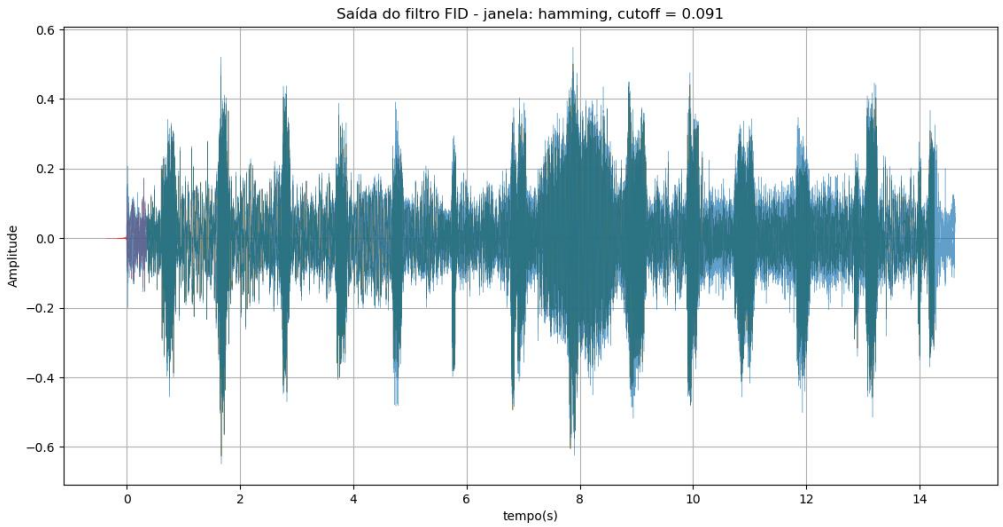


Figura 4.1.4 - Resposta do filtro FIR com janela Hamming e frequência de corte 0.091 Hz

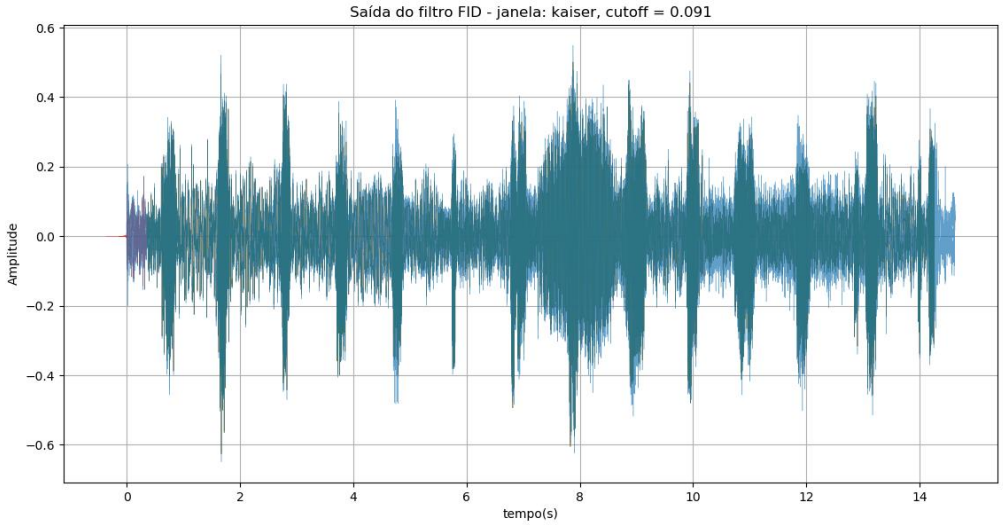


Figura 4.1.5 - Resposta do filtro FIR com janela Kaiser e frequência de corte 0.091 Hz

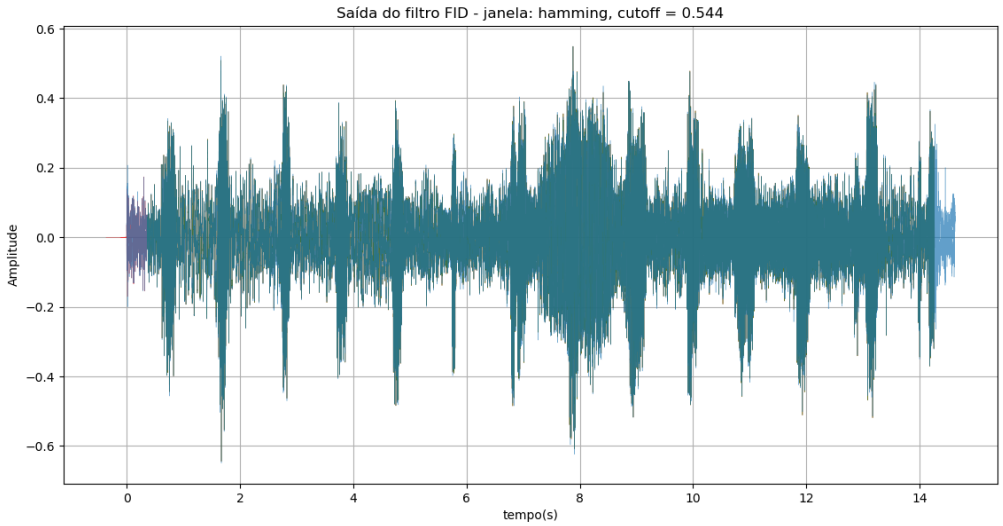


Figura 4.1.6 - Resposta do filtro FIR com janela Hamming e frequência de corte 0.544 Hz

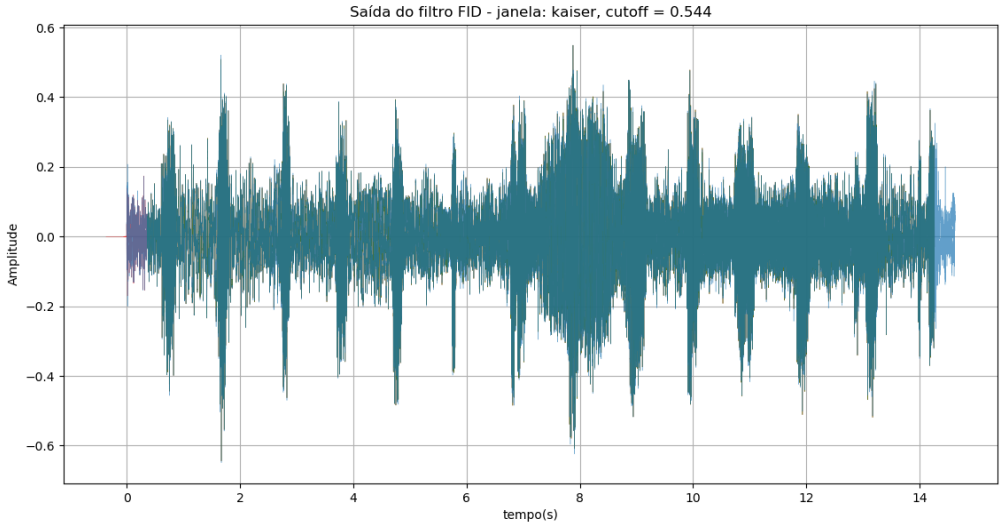


Figura 4.1.7 - Resposta do filtro FIR com janela Kaiser e frequência de corte 0.544 Hz.

A redução do ruído não foi tão significativa em nenhum dos resultados obtidos. Ao janelamento de Kaiser com o janelamento de Hamming percebemos que não houve uma grande disparidade entre os resultado.

A melhor redução de ruído, ou seja, o que mais removeu o som do carro foi o caso em que a janela de corte foi igual a *1000/nyq_rate*, porém houve maior perda na voz, ficando mais difícil de compreender as palavras ditas no áudio. Poderiam ser feitos mais testes próximos a essa frequência de corte para encontrar uma redução mais satisfatória.

Questão 4.2 - Filtros FIR e IIR

O sinal de entrada foi primeiro multiplicado por um fator de 30 e em seguida foi utilizada a função *sin* da biblioteca *Numpy* para gerar um sinal senoidal com frequência 466.16 Hz. Esses dois sinais foram adicionados resultando no sinal mostrado na Figura 4.2.1 (em azul).

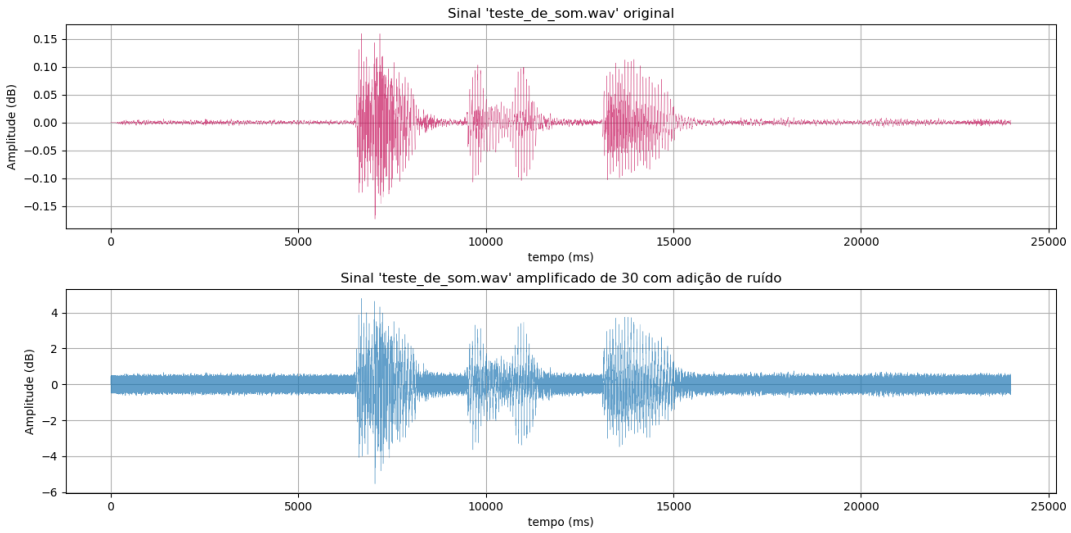


Figura 4.2.1 - Sinal original (acima) e sinal amplificado e com ruído (abaixo)

O Filtro FIR empregado nessa questão foi exatamente o mesmo da questão anterior, explicado na seção 4.1.

Tabela 4.2.1 - Parâmetros de filtragem

Variável	Valor	Significado
número de amostras (nsamples)	24000	comprimento do sinal de entrada
taxa de amostragem (sample_rate)	8000	taxa de amostragem do sinal de entra
taxa de Nyquist (nyq_rate)	4000	taxa de amostragem / 2.0
tamanho da janela (width)	1,25.10^-3	0.5 / taxa de Nyquist
atenuação de kaiser (ripple_db)	60 dB	60.0 db
frequência de corte	3000/nyq_rate = 0.544, 1000/nyq_rate = 0.181, 500/nyq_rate = 0.091	

As imagens abaixam mostram as saídas obtidas após a utilização do filtro FIR. Ao contrário da questão anterior, a redução do ruído foi um pouco mais significativa, mas a mudança do janelamento não teve, assim como anteriormente, disparidade considerável nos resultado.

Para a frequência de corte igual a *1000/nyq_rate* e *500/nyq_rate* , a resposta do filtro foi um sinal mais grave e mais difícil de compreender as palavras, apesar de ter removido a componente senoidal. Para a frequência de corte igual a 3000/nyq_rate, o sinal senoidal não foi atenuado, mas também houve uma piora na qualidade da voz. Poderiam ser feitos mais testes próximos a essa frequência de corte para encontrar uma redução mais satisfatória.

Não foi implementado um filtro IIR para completar a resolução da questão.

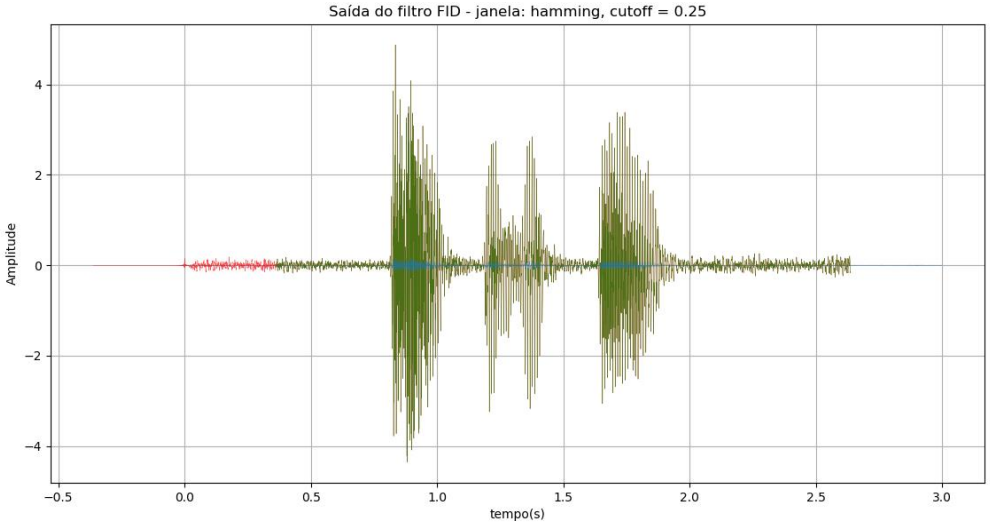


Figura 4.2.2 - Resposta do filtro FIR com janelamento de hamming e frequência de corte 0.025Hz

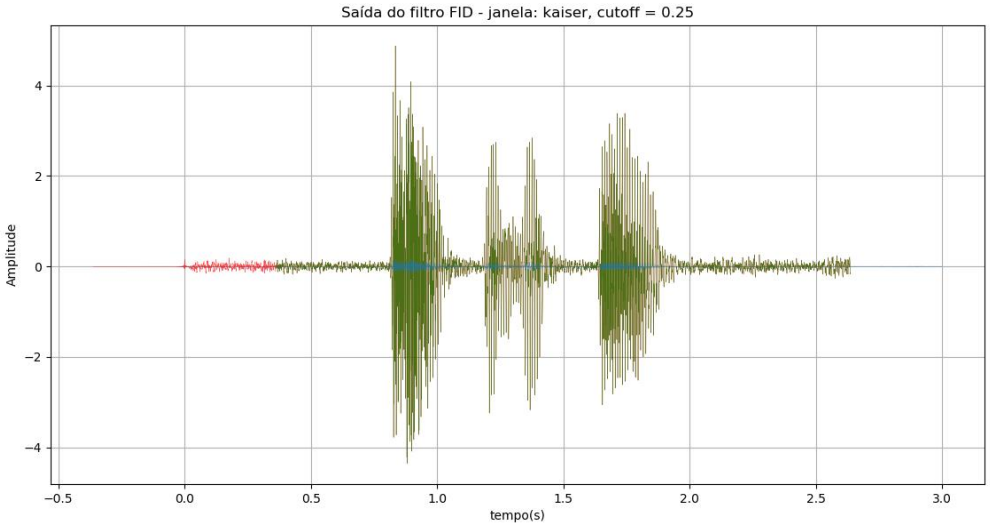


Figura 4.2.3 - Resposta do filtro FIR com janelamento kaiser e frequência de corte 0.025Hz

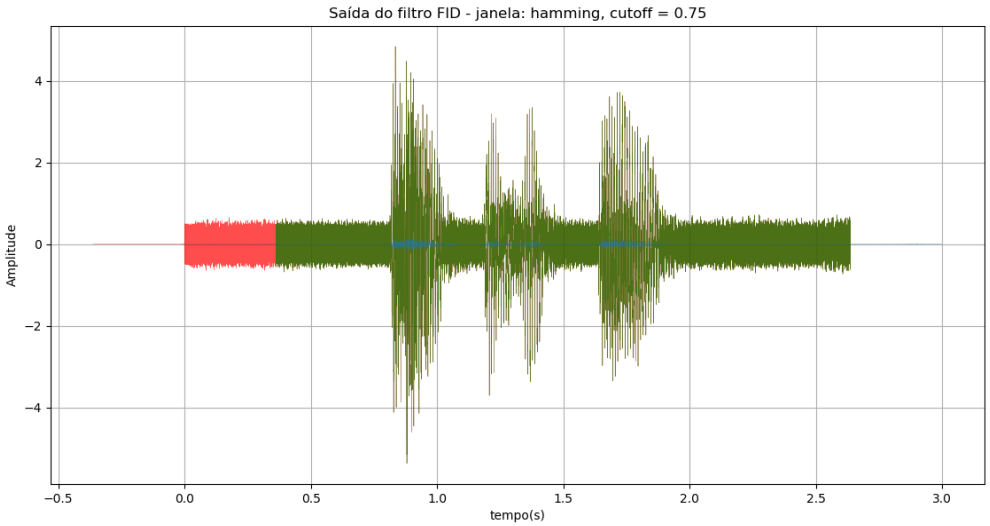


Figura 4.2.4 - Resposta do filtro FIR com janelamento hamming e frequência de corte 0.075Hz

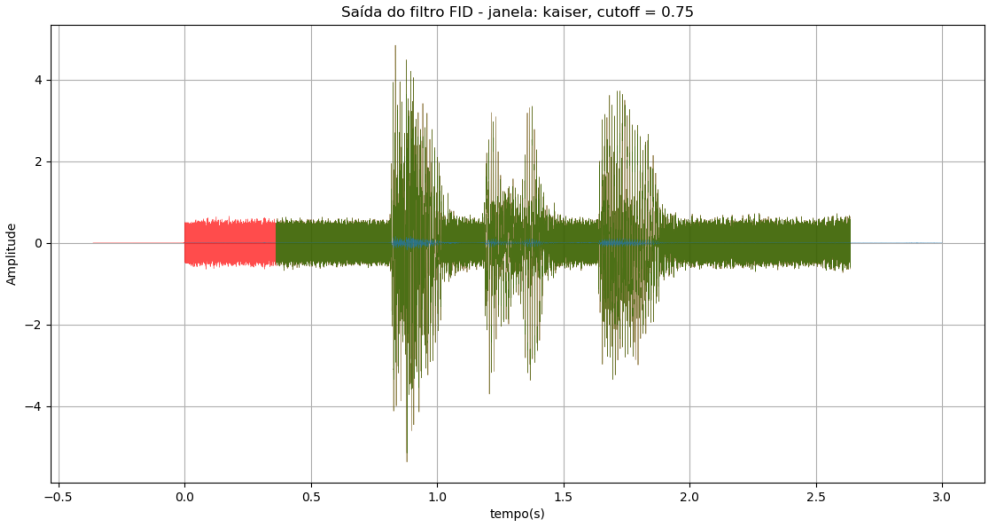


Figura 4.2.5 - Resposta do filtro FIR com janelamento kaiser e frequência de corte 0.075Hz

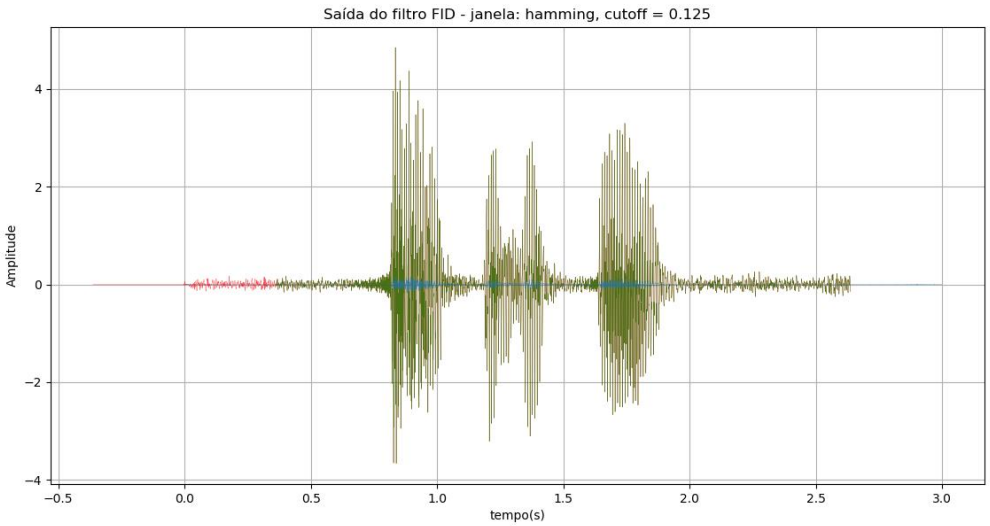


Figura 4.2.6 - Resposta do filtro FIR com janelamento hamming e frequência de corte 0.125Hz

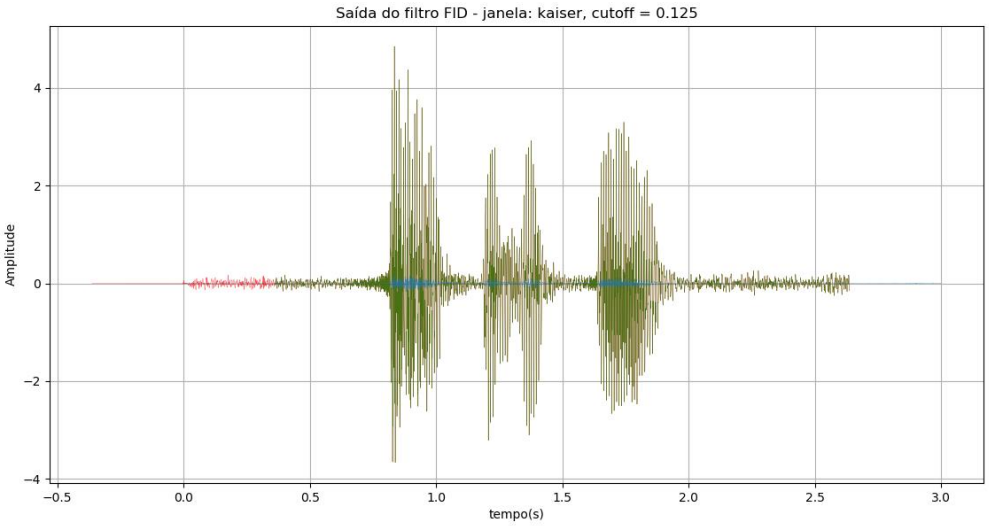


Figura 4.2.7 - Resposta do filtro FIR com janelamento kaiser e frequência de corte 0.125Hz

4 - Bibliografia

1. Especificação de Projeto, https://www.cin.ufpe.br/~cabm/pds/projeto2019_1.pdf <último acesso em 07 de junho de 2019>

2. Notas de Aula, https://www.cin.ufpe.br/~cabm/pds/PDS.pdf <último acesso em 07 de junho de 2019>

3. Reverberation time sound analysis python, https://start-up.house/blog/articles/reverberation-time-sound-analysis-python <último acesso em 07 de junho de 2019>

4. Unidade de Eco e Reverberação (ART2761), https://www.newtoncbraga.com.br/index.php/eletronica/52-artigos-diversos/11858-unidade-de-eco-e-reverberacao-art2761 <último acesso em 07 de junho de 2019>

5. Processamento Digital de Imagens - Filtragens Espaciais, http://www.dpi.inpe.br/~carlos/Academicos/Cursos/Pdi/pdi_filtros.htm <último acesso em 07 de junho de 2019>

6. Contagem da área de objetos em uma imagem binária, https://medium.com/@lucashelal/detec%C3%A7%C3%A3o-e-contagem-da-%C3%A1rea-de-objetos-em-uma-imagem-bin%C3%A1ria-440759a7e034 <último acesso em 07 de junho de 2019>

7. How to build amazing images filters with median filter sobel, https://medium.com/@enzoftware/how-to-build-amazing-images-filters-with-python-median-filter-sobel-filter-%EF%B8%8F-%EF%B8%8F-22aeb8e2f540 <último acesso em 07 de junho de 2019>

8. Canny Edge Detection Step by Step in Python - Computer Vision, https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123 <último acesso em 07 de junho de 2019>

9. Como os daltônicos enxergam as cores?, https://segredosdomundo.r7.com/como-os-daltonicos-enxergam-as-cores/ <último acesso em 07 de junho de 2019>

10. Contrast stretching, http://www.ece.ubc.ca/~irenek/techpaps/introip/manual03.html <último acesso em 07 de junho de 2019>

11. Discover Background Subtraction And BMC, https://web.archive.org/web/20140418093037/http://bmc.univ-bpclermont.fr/ <último acesso em 07 de junho de 2019>

12. Processamento Digital de Sinais - DSP - parte 2, https://www.embarcados.com.br/processamento-digital-de-sinais-dsp-parte-2/ <último acesso em 07 de junho de 2019>

13. Generate Audio with Python, https://zach.se/generate-audio-with-python/ <último acesso em 07 de junho de 2019>

14. Geradores de ruído em python, https://cadernodelaboratorio.com.br/2018/06/25/geradores-de-ruido-em-python/ <último acesso em 07 de junho de 2019>

15. Geração de funções através de um script python, https://cadernodelaboratorio.com.br/2017/10/06/gerador-de-funcoes-atraves-de-um-script-python/ <último acesso em 07 de junho de 2019>

16. Applying a FIR filter, https://scipy-cookbook.readthedocs.io/items/ApplyFIRFilter.html <último acesso em 07 de junho de 2019>