Relatório 1
Projeto em Eletrônica I - EEL7801

Universidade Federal de Santa Catarina - UFSC
Professora: Daniela Ota Hisayasu Suzuki

Luiz Augusto Frazatto Fernandes: *17202752*
Leonardo José Held: *17203984*

02 de Maio de 2019

Nota: O projeto todo, incluindo este documento e os demais códigos de simulação e de projeto podem ser encontrados em `https://github.com/leonheld/EEL7801`

Chapter 1

Metodologia

1.1 Modulação e demodulação de sinais

1.1.1 Motivação da escolha do algoritmo

Escolheu-se o processo de modulação por Chaveamento de Deslocamento de Frequência (FSK, em inglês). São usadas duas frequências ótimas para se representar 0 e 1, e ambas são obtidas experimentalmente: a partir de testes realizados com o transdutor (microfone), é gerada uma curva normal, em que a resposta desse ao sinal recebido é ótima para uma frequência específica (frequência da onda portadora). São, então, obtidas duas outras frequências equidistantes do centro da curva gaussiana, e a cada uma é associado um valor binário.

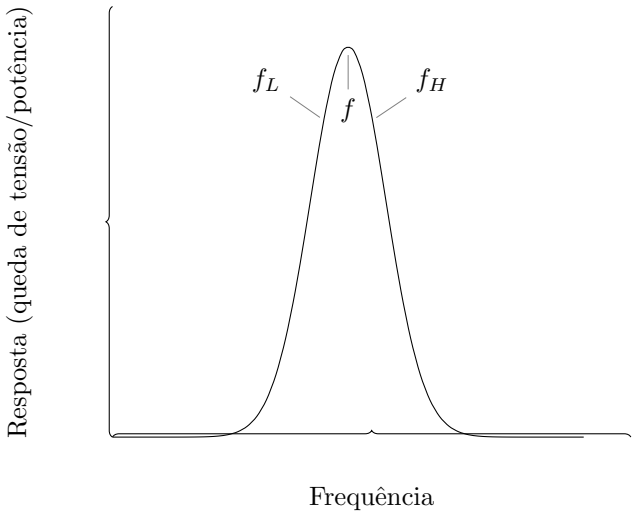
O processo de modulação, em si, consiste na transformação de um sinal PWM em um analógico (que é controlado por um STM32) que, por sua vez, é emitido por um tweeter. O sinal (sonoro) é recebido por um microfone controlado por outro STM32, que realizará o processo de demodulação do sinal.

1.1.2 Algoritmo da modulação (FSK)

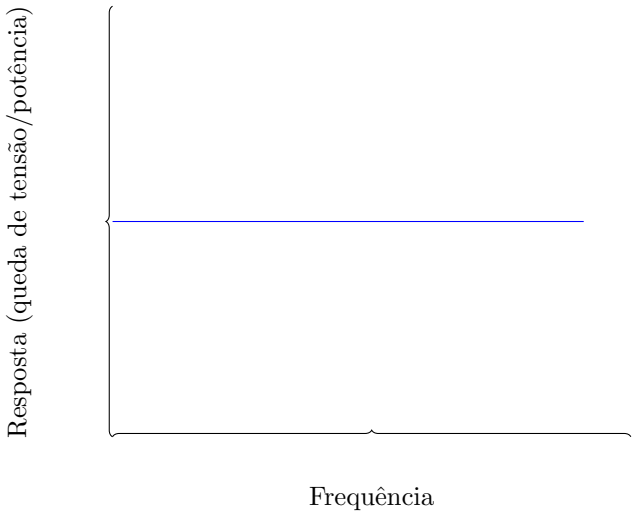
Empiricamente são testadas (com o microfone) diferentes frequências emitidas pelo tweeter e recebidas pelo microfone. A de melhor resposta (dada, no gráfico abaixo, por f) é associada à onda portadora (carrier). É, então, estabelecido um desvio Δf, e, a partir desse, são determinadas:

$$f_L = f - \Delta f$$
$$f_H = f + \Delta f$$

de tal forma que à f_H se associa o bit 1, e à f_L, 0.



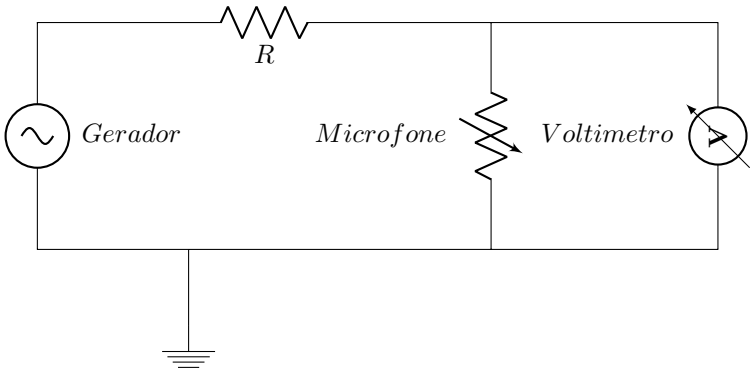
Há, também, uma segunda possibilidade: de o microfone não reagir significativamente a diferentes frequências, mas de ainda assim conseguir diferenciá-las. Isto é: o microfone identifica quando duas frequências recebidas são diferentes, mas sua curva de queda de tensão se aproxima de uma constante, como no exemplo a seguir:



Caso esse comportamento seja verificado, o que há de se fazer é, simplesmente, escolher duas frequências quaisquer para se trabalhar no algoritmo.

1.1.3 Obtenção das frequências ótimas

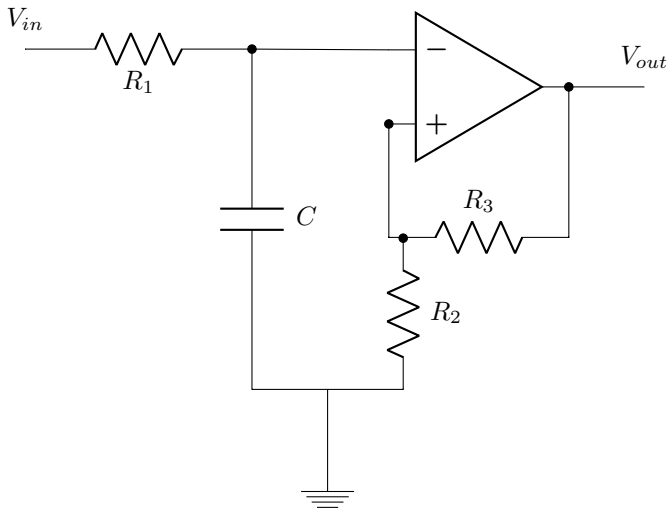
Por meio do circuito da (inserir número da imagem aqui) é medida a tensão/potência dissipada nos terminais do microfone. A curva (próxima à normal descrita acima) é gerada a partir da potência consumida/queda de tensão que é medida entre os terminais do microfone.



Quanto maior a queda de tensão no microfone, maior a energia por ele consumida em relação ao resto do circuito (divisor de tensão), logo, melhor será sua resposta aos estímulos sonoros testados.

1.1.4 Tratamento do sinal analógico

Será utilizado, a priori, um filtro passa-baixa a fim de reduzir o ruído de alta frequência, além de um amplificador não inversor (que garantirá uma clara análise do sinal).



A frequência de corte do circuito acima é dada por

$$f_c = \frac{1}{2 \cdot \pi \cdot R_1 \cdot C} \tag{1.1}$$

E o ganho por

$$\frac{V_{out}}{V_{in}} = G = 1 + \frac{R_3}{R_2} \tag{1.2}$$

Espera-se trabalhar com frequências na faixa audível ($20Hz$ a $20kHz$). Portanto, pode-se colocar uma frequência de corte razoável de 22 kHz a priori, já

que, dessa forma, o ruído de alta frequência filtrado não interferirá nos sinais desejados.

Para tal, usamos um capacitor com capacitância $C = 0.1\mu F$, o que nos resulta:

$$R_1 = \frac{1}{2 \cdot \pi \cdot 0.1 \cdot 10^{-6} \cdot 22 \cdot 10^3} \approx 72\Omega \tag{1.3}$$

Para avaliarmos o ganho, deixamos em R_3 um potenciômetro e, assim, avaliamos, a posteriori, qual o melhor ganho para se projetar (aquele cuja análise posterior for facilitada).

1.2 Escolha de Microcontrolador (μC)

1.2.1 STM32

O microcontrolador foi escolhido na base da seguinte *criteria*:

1) Baixo Custo: O controlador deve ter um custo mínimo, com alta flexibilidade de prototipação. Deve incluir um programador integrado, ou possuir algum de baixo custo e facilmente acessível.

2) Alto suporte: O controlador deve ter suporte de compiladores, ambientes de programação, comunidade de software aberto e pela própria empresa que o fabrica.

3) Performance: A unidade deve ter uma performance que faz processamento de dados e sinais com relativa rapidez, de forma que a sua capacidade de processamento não dite o algoritmo e execução do programa.

Em consideração aos items **1** e **2**, se optou por utilizar algum controlador da família ARM^{TM}, indubtavelmente a família mais utilizada de processadores no mundo, e desta forma, acessível e de bom suporte.

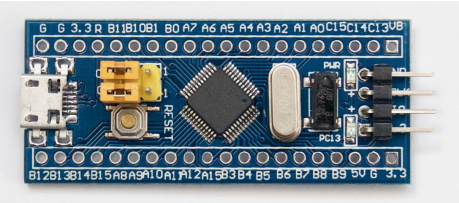
O quesito performance nos faz olhar para as famílias de processadores que a ARM^{TM} atualmente oferece, dentre as quais, os processadores Cortex-M3^{TM} [1] parecem entrar nos quesitos.

De acordo com o página da empresa sobre o processador:

"The Arm Cortex-M3 processor is the industry-leading 32-bit processor for highly deterministic real-time applications."

Depois de uma curta pesquisa de mercado, a placa STM32F103C8 da ST^{TM}, também conhecida como *bluepill*, parece preencher bem os quesitos supraescritos, com um processador Cortex-M3^{TM} de 72MHz de clock interno, é um dos microcontroladores mais potentes da categoria fabricados pela ST^{TM}, o que preenche bem o requisito **3**.

[1] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3



STM32F103C8.

Qualquer processador dessa linha da ST™ pode ser programado e debuggado via uma interface chamada ST-Link™, que também foi de fácil obtenção e baixo custo, e basicamente inclui um outro processador que conversa em tempo real com a placa alvo.

Programador SWD ST-Link.

1.2.2 Suporte

O suporte oferecido tanto pela ARM™ quanto pela ST™ é extenso. A ST™ produz seu próprio software de pré-configuração e geração de código chamado STM32CubeMX™, que é uma *Graphical User Interface* usada para gerar o código inicial, como configuração de clock interno e dos pinos do processador, por exemplo.

A ARM™ também mantém um compilador, um debugger e utilidades de programação para sua linha de processadores, todas as ferramentas baseadas no GCC, GDB e Gnu Binutils.[2]

Como software de upload de código e debugging, podemos usar o programa *openocd* [3], também de código aberto que acessa a interface SWD (*Serial Wire Debug*) do ST-Link™ via terminal, com possibilidade de abrir uma janela de sessão de debugging via GDB.

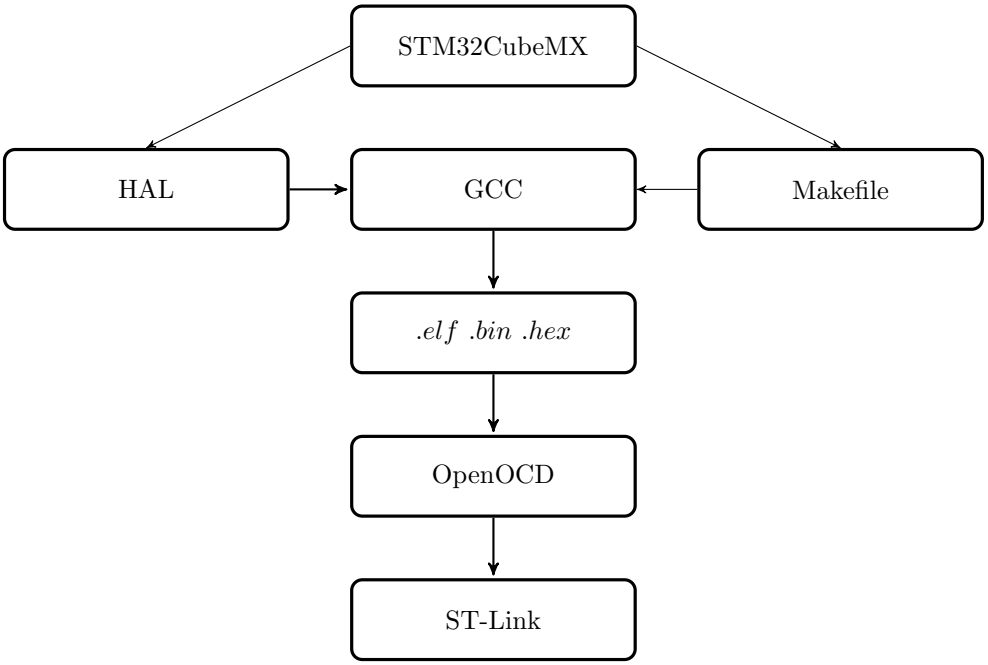
1.2.3 Processo de Desenvolvimento

Desde a última versão, o STM32CubeMX™ consegue produzir um Makefile que pode ser compilado pelo programa *make* chamando o compilador de C cedido pela ARM.

A partir disso, o compilador gera arquivos .hex, .elf e .bin, quaisquer podendo ser inseridos na memoria interna do microcontrolador pelo openocd.

[2] https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain
[3] http://openocd.org/

Para a programação, existe uma HAL - *Hardware Abstration Layer* -, também fornecida pela ST^{TM}, que é incluída nos sources do projeto no momento da inicialização do código pelo STM32CubeMX^{TM} e possuí funções específicas de baixo nível, mas tem as mesmas chamadas em quaisquer processador alvo que queira executar o código, aumentando a portabilidade. Também, a HAL é construída de forma a aumentar o nível de abstração, diminuindo significativamente a complexidade do código, permitindo desenvolvimento rápido e colaborativo, sem necessariamente conhecer todos os artifícios de baixo nível de cada processador.



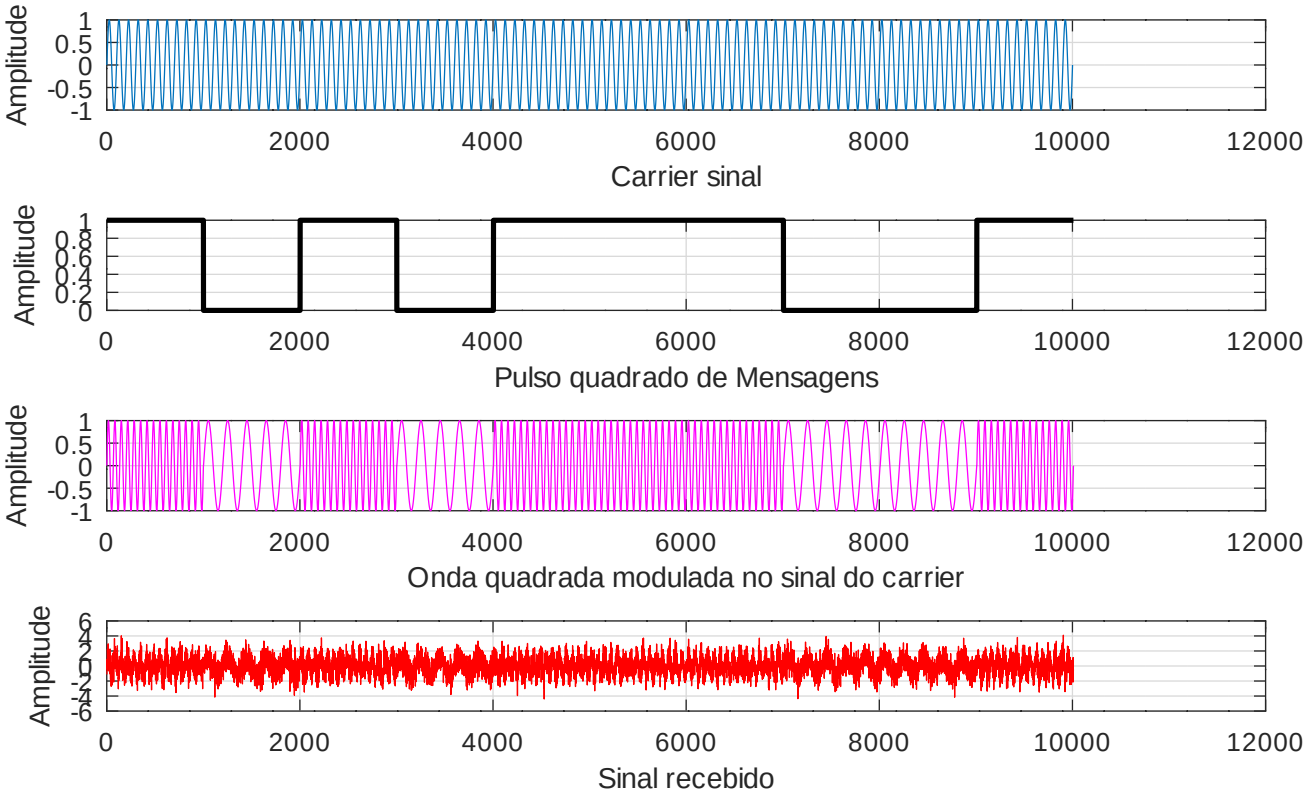
Flowchart de programação e desenvolvimento do firmware.

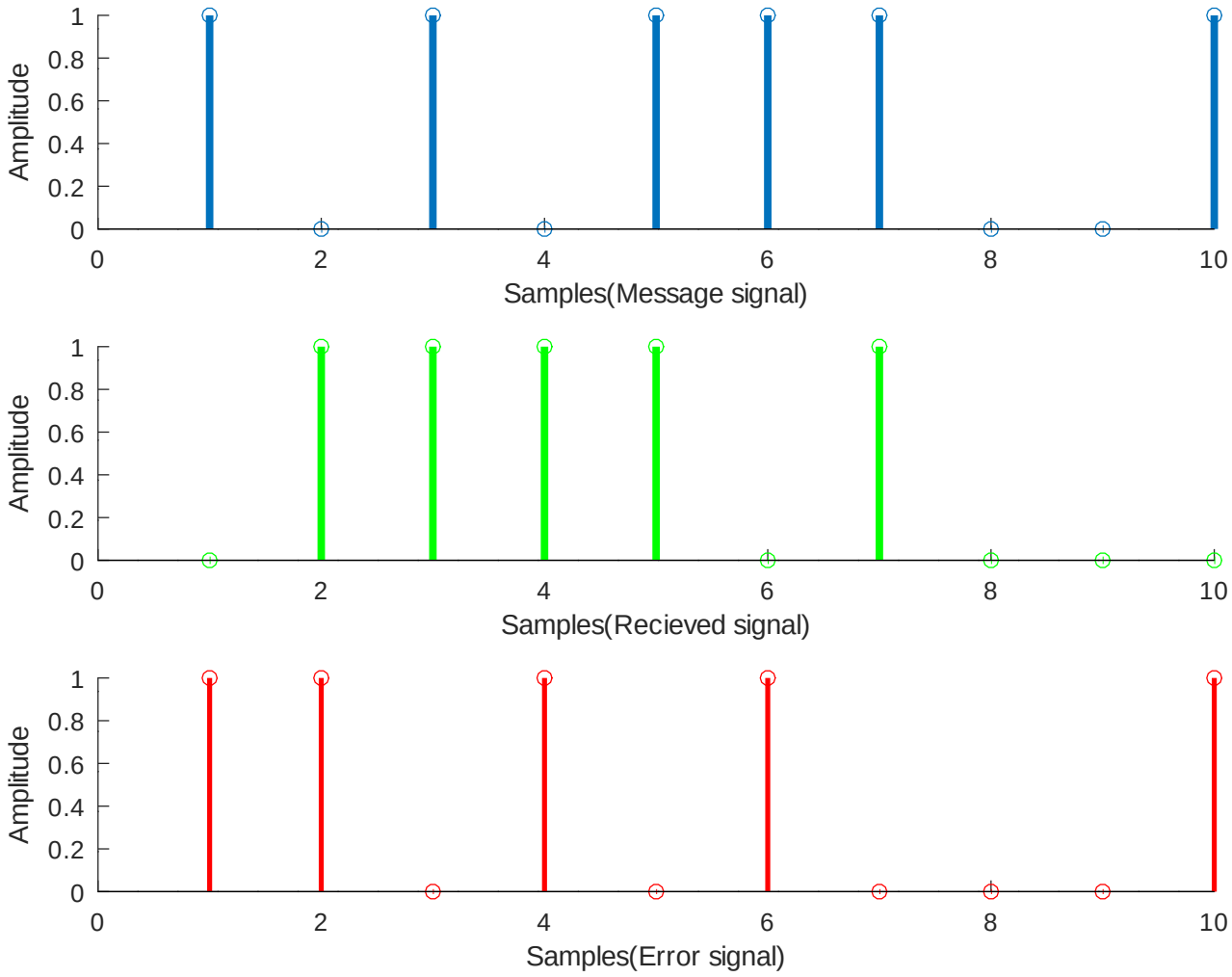
Chapter 2

Title

2.1 Title

2.1.1 graphics


Amplitude
1
0.5
0
-0.5
-1
0
2000
4000
6000
8000
10000
12000
Carrier sinal
Amplitude
1
0.8
0.6
0.4
0.2
0
0
2000
4000
6000
8000
10000
12000
Pulso quadrado de Mensagens
Amplitude
1
0.5
0
-0.5
-1
0
2000
4000
6000
8000
10000
12000
Onda quadrada modulada no sinal do carrier
Amplitude
6
4
2
0
-2
-4
-6
0
2000
4000
6000
8000
10000
12000
Sinal recebido


Amplitude
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
Samples(Message signal)
Amplitude
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
Samples(Recieved signal)
Amplitude
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
Samples(Error signal)

2.1.2 demod

```
function [NORMALIZED_BIT_ERROR_RATE] = fsk_demod(sinal_modulado, ruido)

    onda_transmitida = awgn(sinal_modulado, ruido); %adiciona ruido no sinal

    data = [1 0 1 0 1 1 1 0 0 1]; %apenas necessario pra computar a taxa de erro
    nro_bits = length(data);

    holdup_time = 10;

    frequencia_carrier = 1000;
    periodo_carrier = 1/frequencia_carrier;

    f_sampling = frequencia_carrier * 100;
    periodo_sampling = 1/f_sampling;

    holdup_time = 10;

    tempo_sampling = 0:periodo_sampling:(periodo_carrier*holdup_time);

    negative=0;
    positive=0;

    sampleValue = nro_bits;
    dif_sinal=0;
    nro_zeros=0;
    amostra_de_zeros=[];
    k=1;
    for i=1:10
        for j=1:length(tempo_sampling)
            if(dif_sinal>sampleValue)
                if(onda_transmitida(1,k)>0)
                    positive=1;
                end
                if(onda_transmitida(1,k)<0)
                    negative=1;
                end
            end
            k++;
            dif_sinal=dif_sinal+1;
            if(positive==1 && negative==1)
                nro_zeros++;
                positive=0;
                dif_sinal=0;
                negative=0;
            end
        end
        amostra_de_zeros=[amostra_de_zeros nro_zeros];
        nro_zeros=0;
    end
```

```
    %normalize os vetores dividindo-os pela MALVADA
    amostra_de_zeros=amostra_de_zeros/mean(amostra_de_zeros);

    filtData=[];
    for i=1:length(amostra_de_zeros)
     if(amostra_de_zeros(i)>=1)
         filtData=[filtData 1];
     else
         filtData=[filtData 0];
     end
    end

    [BIT_ERROR_RATE NORMALIZED_BIT_ERROR_RATE]=biterr(data,filtData);
end
```

2.1.3 mod

```
data = [1 0 1 0 1 1 1 0 0 1]; %defina os bits a serem modulados na onda
nro_bits = length(data);

%DEFINIR SINAL CARRIER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%frequência e período da onda carrier
frequencia_carrier = 1000;
periodo_carrier = 1/frequencia_carrier;

%frequência e período que definem a sampling rate(baseado na f e t da carrier)
f_sampling = frequencia_carrier * 100;
periodo_sampling = 1/f_sampling;


holdup_time = 10;
proportional_holdup_time = holdup_time*nro_bits;

t = 0:periodo_sampling:(proportional_holdup_time*periodo_carrier);


onda_carrier = sin(2*pi*t*frequencia_carrier);

%PROCESSO DE MODULAÇÃO

delta_frequencia = 0.5; % o quão violenta vai ser o `amortecimento` nos carregamentos de frequencia
frequencia_alta = frequencia_carrier + (frequencia_carrier*delta_frequencia);
frequencia_baixa = frequencia_carrier - (frequencia_carrier*delta_frequencia);

%definição das frequencias moduladas
carrier_alta = sin(2*pi*tempo_sampling*frequencia_alta); %bit alto
```

```
carrier_baixa = sin(2*pi*tempo_sampling*frequencia_baixa); %bit baixo

sinal_modulado = [];

for i=1:nro_bits
     if(data(i)==1)
         sinal_modulado = [sinal_modulado carrier_alta];
     else
         sinal_modulado = [sinal_modulado carrier_baixa];
     end
 end

ruido = 0.1;
onda_transmitida = awgn(sinal_modulado, ruido); %adiciona ruido no sinal
```