

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>

Nota de Trademark:

Arm<sup>TM</sup>, ARM<sup>TM</sup>, Cortex<sup>TM</sup> são marcas registradas da Arm Limited. Copyright © 1995-2019 Arm Limited (or its affiliates).

STM32CubeMX<sup>TM</sup>, ST<sup>TM</sup>, ST-Link<sup>TM</sup> são marcas registradas da Singapore Technologies Engineering Limited. Copyright © 2019 Singapore Technologies Engineering Ltd.

GCC is Copyright (C) 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008 Free Software Foundation, Inc.

Octave is Copyright © 1996-2016 John W. Eaton.

# 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.

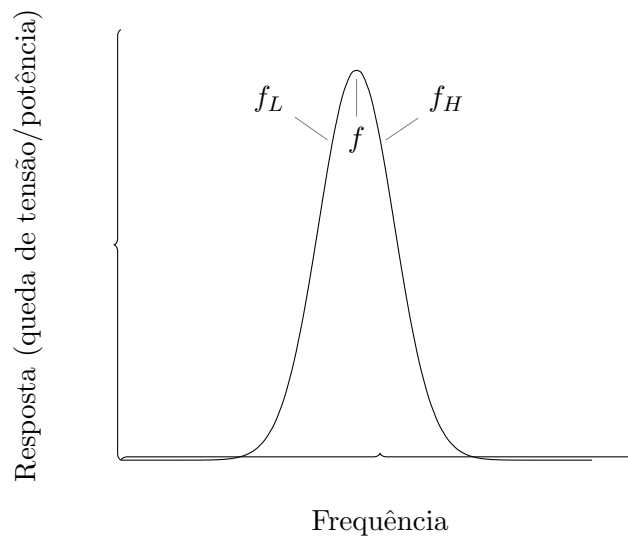
O processo de modulação, em si, consiste na transformação de um sinal PWM em um analógico (que é controlado por um STM32) que, por sua vez, é emitido por um tweeter. O sinal (sonoro) é recebido por um microfone controlado por outro STM32, que realizará o processo de demodulação do sinal.

#### 1.1.2 Algoritmo da modulação (FSK)

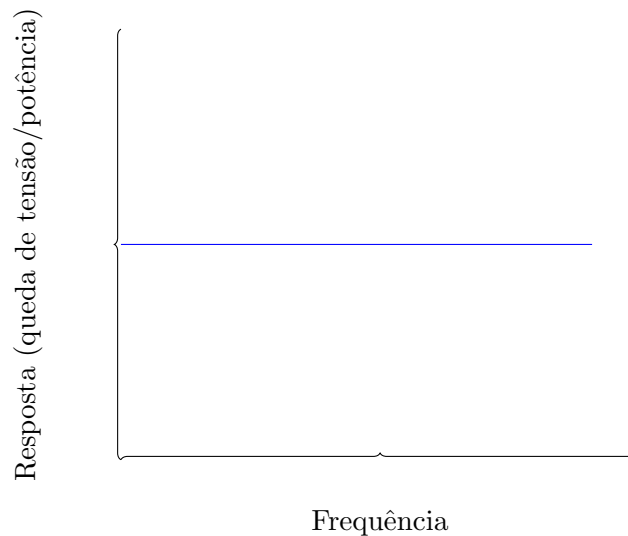
Empiricamente são testadas (com o microfone) diferentes frequências emitidas pelo tweeter e recebidas pelo microfone. A de melhor resposta (dada, no gráfico abaixo, por  $f$ ) é associada à onda portadora (carrier). É, então, estabelecido um desvio  $\Delta f$ , e, a partir desse, são determinadas:

$$\begin{aligned}f_L &= f - \Delta f \\f_H &= f + \Delta f\end{aligned}$$

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:

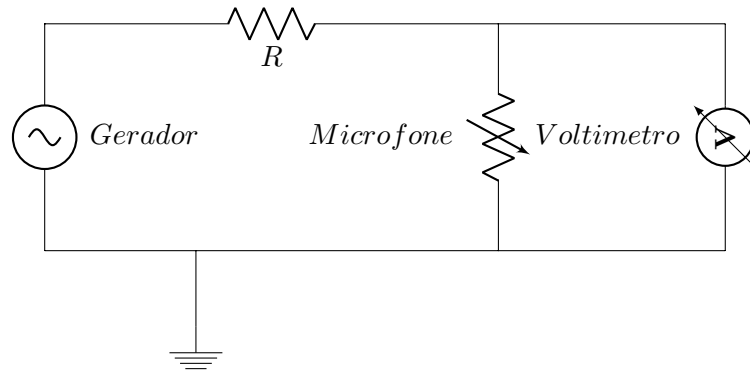


Caso esse comportamento seja verificado, o que há de se fazer é, simplesmente, escolher duas frequências quaisquer para se trabalhar no algoritmo.

### 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

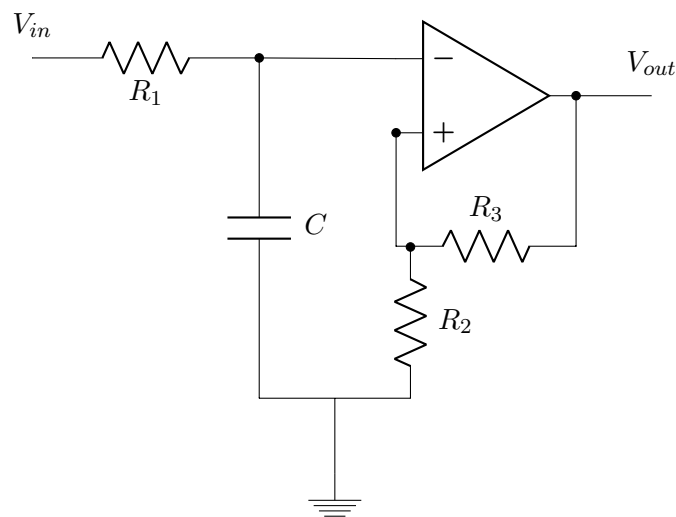
à normal descrita acima) é gerada a partir da potência consumida/queda de tensão que é medida entre os terminais do microfone.



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

$$f_c = \frac{1}{2 \cdot \pi \cdot R_1 \cdot C} \quad (1.1)$$

E o ganho por

$$\frac{V_{out}}{V_{in}} = G = 1 + \frac{R_3}{R_2} \quad (1.2)$$

Espera-se trabalhar com frequências na faixa audível ( $20Hz$  a  $20kHz$ ). Portanto, pode-se colocar uma frequência de corte razoável de  $22\text{ kHz}$  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:

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

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).

### 1.1.5 Circuito amplificador

O núcleo do amplificador será o chip *LM386*, cuja pinagem pode ser vista abaixo:

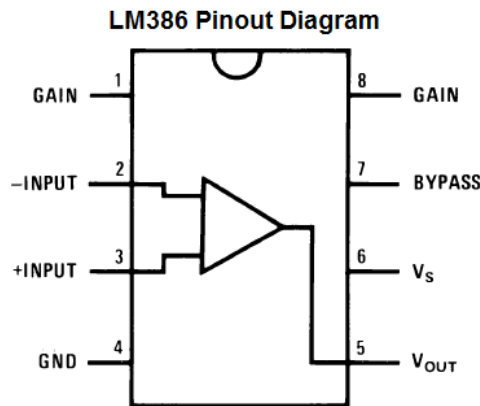
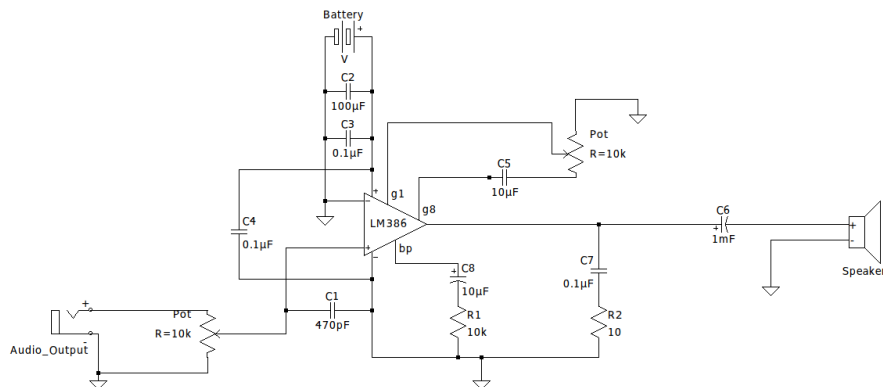


Diagrama com pinagem do LM386

O circuito abaixo será utilizado a fim de se amplificar o sinal emitido pelo modulador, para que esse seja reproduzido pelo transdutor utilizado.<sup>1</sup>

<sup>1</sup><http://www.circuitbasics.com/build-a-great-sounding-audio-amplifier-with-bass-boost-from-the-lm386/>



Circuito amplificador (conectado entre saída do emissor e entrada do speaker/transdutor)

1.  $C_1$  é responsável por filtrar eventuais interferências captadas pelos fios.
2.  $C_2$  e  $C_3$  são responsáveis por desacoplar a fonte do amplificador. o de  $100\mu F$  filtrará ruído de alta frequência, enquanto o de  $0.1\mu F$ , de baixa.
3.  $C_4$  desacopla os pinos de alimentação do chip.
4.  $C_5$ , juntamente com o potenciômetro, tem a função de regular o ganho do *LM386* entre 20 e 200 vezes.<sup>2</sup>
5.  $C_6$  rejeita correntes contínuas.
6.  $C_7$  e  $R_2$  atuam como um passa-alta, a fim de evitar ruídos no som emitido pelo speaker.
7.  $C_8$  e  $R_1$  desacoplam a entrada de áudio (bypass pin).
8. Os potenciômetros têm finalidade de teste (possivelmente serão substituídos por resistores, uma vez devidamente testadas resistências condizentes com o circuito).

## 1.2 Escolha de Microcontrolador ( $\mu C$ )

### 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.

<sup>2</sup><http://www.ti.com/lit/ds/symlink/lm386.pdf>

- 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.

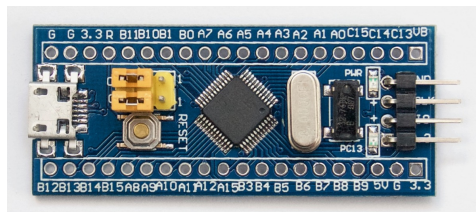
Em consideração aos itens **1** e **2**, se optou por utilizar algum controlador da família ARM<sup>TM</sup>, indubitavelmente a família mais utilizada de processadores no mundo, e desta forma, acessível e de bom suporte.

O quesito performance nos faz olhar para as famílias de processadores que a ARM atualmente oferece, dentre as quais, os processadores Cortex-M3<sup>3</sup> 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, também conhecida como *bluepill*, parece preencher bem os quesitos supracitados, com um processador Cortex-M3 de 72MHz de clock interno, é um dos microcontroladores mais potentes da categoria fabricados pela ST, o que preenche bem o requisito **3**.



STM32F103C8.

Qualquer processador dessa linha da ST pode ser programado e debugado 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.

---

<sup>3</sup><https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3>





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.<sup>4</sup>

Como software de upload de código e debugging, podemos usar o programa *openocd*<sup>5</sup>, 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 memória interna do microcontrolador pelo openocd.

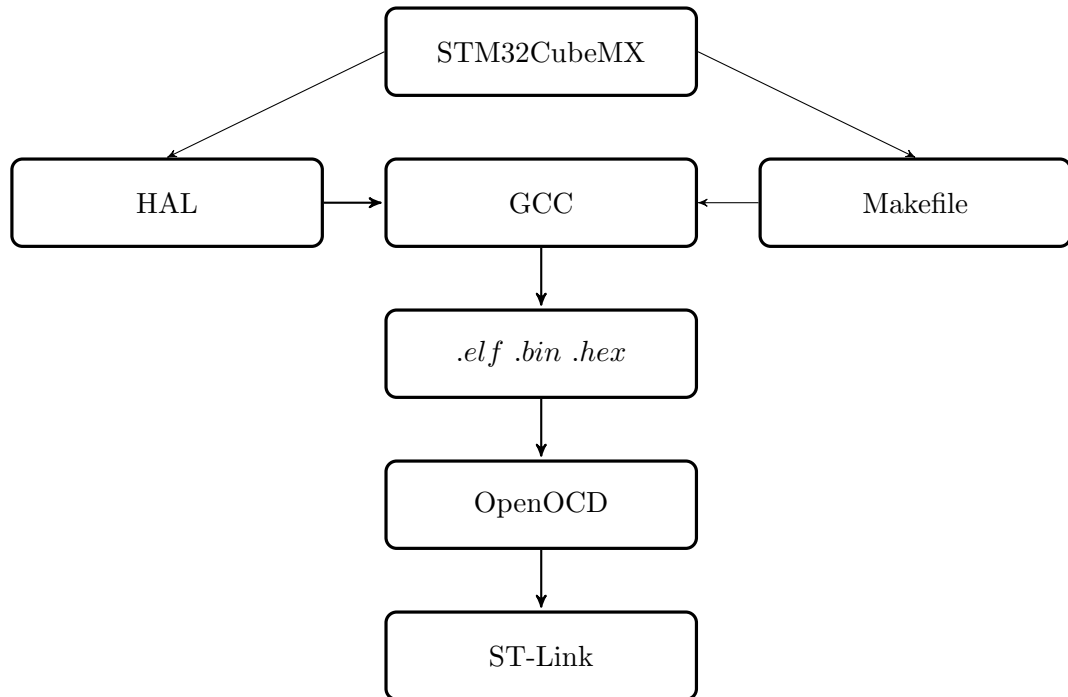
Para a programação, existe uma HAL - *Hardware Abstraction Layer* -, também fornecida pela ST, que é incluída nos sources do projeto no momento da inicialização do código pelo STM32CubeMX e possui funções específicas de baixo nível, mas tem as mesmas chamadas em quaisquer processador alvo que queira 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

---

<sup>4</sup><https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain>

<sup>5</sup><http://openocd.org/>

rápido e colaborativo, sem necessariamente conhecer todos os artifícios de baixo nível de cada processador.



Flowchart de programação e desenvolvimento do firmware.

## Chapter 2

# Simulação e Algoritmo

### 2.1 Simulação usando Octave

Os algoritmos foram programados usando Octave, uma linguagem de manipulação de vetores que possui pacotes específicos para comunicação e sinais digitais. Também possui uma função `agwn`, ou *Additive Gaussian White Noise*, que introduz barulho no sistema e no sinal.

O algoritmo base de simulação foi pego do site da MathWorks, criadora do MATLAB, linguagem compatível com Octave. Dessa base, alteramos o código de forma a cortar excessos e tentando eliminar elementos de aleatoriedade, vistos que são mais difíceis de se obter na plataforma de hardware que pretendemos usar. Também modularizamos o código de forma que podemos fazer  $n$  simulações com padrões diferentes de onda, *sampling rates*, diferentes frequências e tudo entre o meio.

O algoritmo é extremamente bem conhecido e documentado. O algoritmo de modulação compreende em gerar uma onda sinusóide que é modulada em função de um vetor binário. De forma resumida:

$$\mathfrak{F}(V) \text{ onde } V(i) = 1 \vee 0 \text{ para } i \in \mathbb{N} \quad (2.1)$$

e

$$\mathfrak{F} \begin{cases} \delta_+, & V(i) = 0 \\ \delta_-, & V(i) = 1 \end{cases}$$

Defina  $\mathcal{L}$  como um vetor que receberá um sinal senoidal, dependendo de um fator  $\delta_{+/-}$  da função  $\mathfrak{F}$

$$\mathcal{L} = \mathfrak{F}(V) = \sin 2 \cdot \pi \cdot t \cdot f_{\delta_{+/-}} \quad (2.2)$$

Vale reiterar que essa  $f$  é baseada numa frequência da onda *carrier*. Como estamos destacando frequências diferentes para cada sinal,

a "carrier" é só uma alegoria para determinar frequência e período de uma onda fundamental que nunca será gerada. A alternativa seria usar uma onda continua onde o bit 1 ou 0 são gerados a partir de apenas um *shift* de frequência, e não dois como estamos fazendo aqui.

Dessa forma, se constrói uma onda que varia no tempo e é construída com duas frequências de acordo com os bits do vetor que se deseja modular.

Em termos de implementação, é possível direcionar os valores amostrados dessa senóide para um Conversor Digital Analógico, e obter amplificação das mesmas.

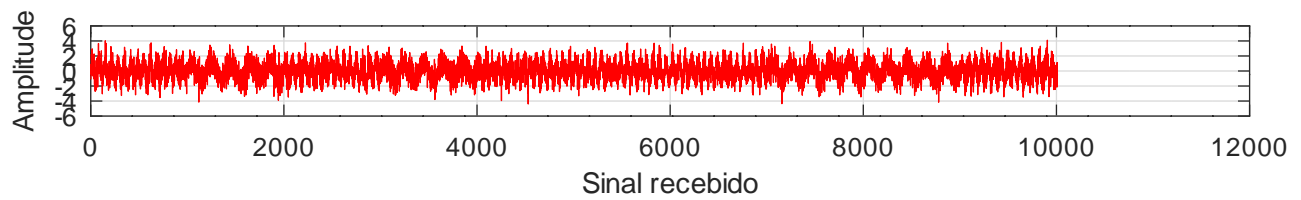
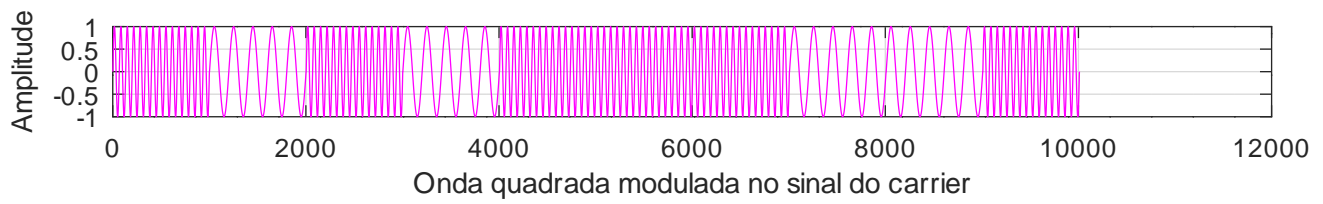
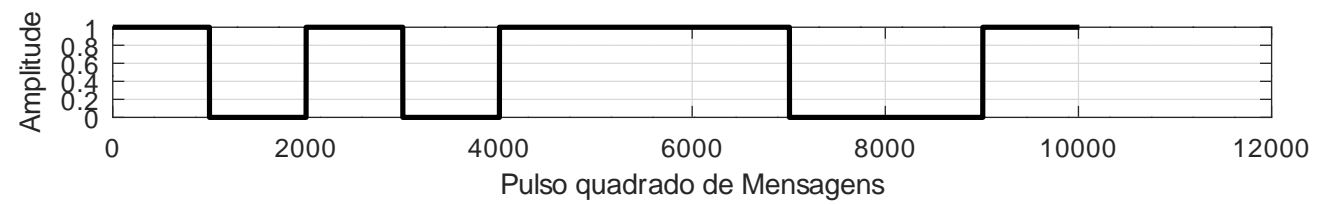
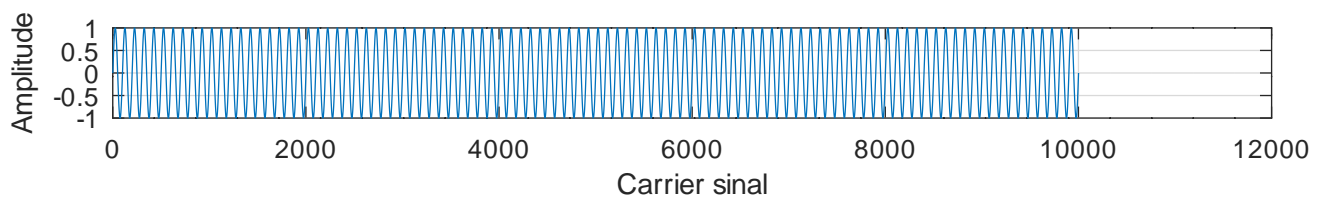
O resultado da simulação pode ser visto da próxima página. O primeiro gráfico representa uma onda pura, não modulada, a "carrier", que como mencionamos, nunca será gerada no contexto do nosso algoritmo.

O segundo gráfico está associado com um pulso quadrado de mensagens que representa o vetor

$$V = [1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1]$$

O terceiro gráfico é a onda já modulada, passada pelo algoritmo citado acima. Perceba os cortes horizontais de frequência gerados pelo vetor do pulso quadrado.

A quarta onda é a terceira com *white noise* adicionado. Podemos usar o white noise como barulho e interferência criado pelo ambiente durante a execução do programa. A razão  $\frac{signal}{noise}$  naquele caso está extremamente alta, da ordem de 100. É perceptível a diferença na onda.



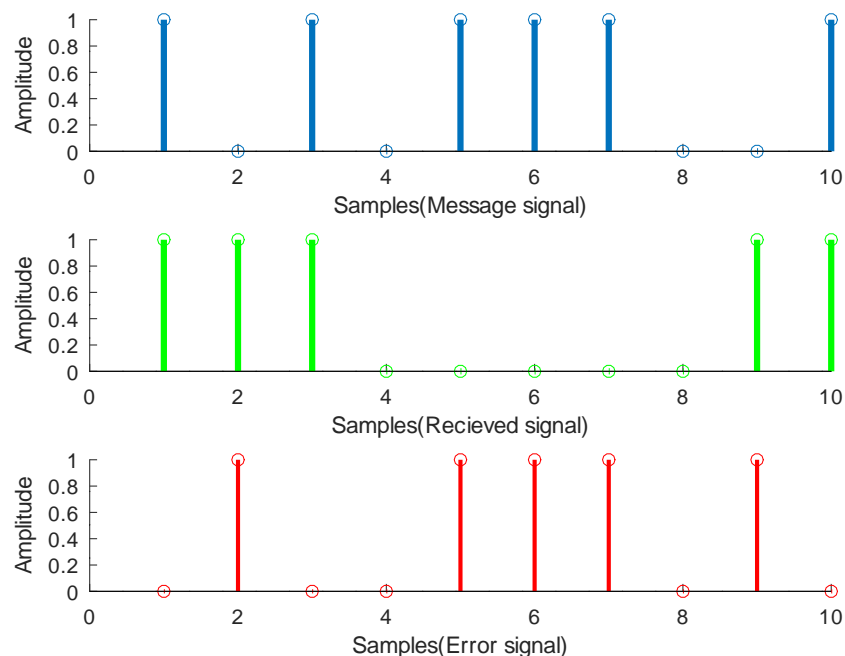
O processo de demodulação do sinal consiste em, basicamente, avaliar quantas vezes o sinal cruza a origem (em relação ao eixo temporal) para que, dessa forma, se consiga gerar uma frequência média para esse.

A partir do número de vezes em que o sinal passa pela origem, é criado um vetor que contem essa quantidade. Esse vetor (em nosso caso, de 10 componentes) é normalizado conforme a média de zeros contida em cada uma das 10 posições (é o processo de reconstrução da onda carrier). Isso é feito a fim de se comparar, após cálculo, com o valor 1, ou seja: se o "número de zeros" normalizados forem maiores do que ou iguais a 1 na posição 0 do vetor, então a posição 0 do vetor assume bit 1, já que houve mais oscilações que a média nesse intervalo de tempo (frequência mais alta). Caso o valor normalizado seja menor que 1, então a componente correspondente no vetor assume valor binário 0.

O primeiro plot retrata o vetor original (aquele que fora emitido).

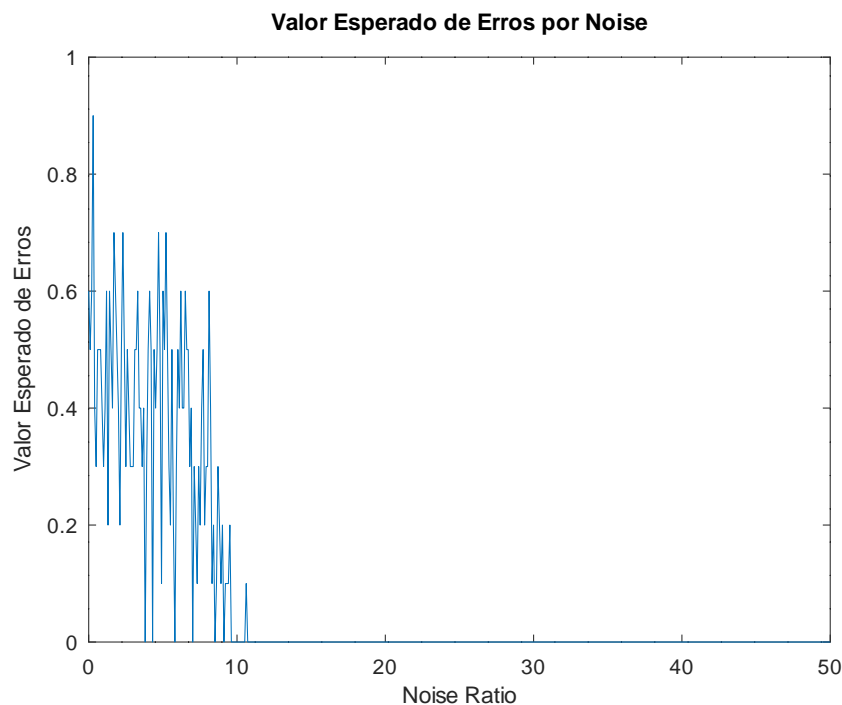
O segundo, o vetor que foi recebido e demodulado.

O terceiro, a comparação final entre o sinal emitido e o sinal percebido. Os erros ocorrem em função, principalmente, do ruído (de alta frequência), que altera a frequência da onda carrier demodulada.



Perceptivelmente, de 10 bits, 5 tiveram erros de transmissão. Isso se dá pelo fato do *noise* introduzido ter sido máximo, garantindo que mesmo que o sinal esteja destruído por interferências, pelo menos 50% dos bits serão transmitidos com sucesso.

É útil, então, saber como o erro de transmissão se comporta em relação à distorção e interferência no sinal. Esse plot apresenta justamente isso, com 500 iterações do mesmo vetor binário sendo modulado e demodulado, mas com a razão de interferência/sinal variando.



O Valor Esperado de Erros (em %) decresce conforme a razão de interferência cai (noise aumenta com o sinal constante).

### 2.1.1 demod

```
1 function [NORMALIZED_BIT_ERROR_RATE] = fsk_demod(sinal_modulado, ruido)
2
3     onda_transmitida = awgn(sinal_modulado, ruido); %adiciona ruido no sinal
4
5     data = [1 0 1 0 1 1 1 0 0 1]; %apenas necessario pra computar a taxa de erro
6     nro_bits = length(data);
7
8     holdup_time = 10;
```

```

9
10     frequencia_carrier = 1000;
11     periodo_carrier = 1/frequencia_carrier;
12
13     f_sampling = frequencia_carrier * 100;
14     periodo_sampling = 1/f_sampling;
15
16     holdup_time = 10;
17
18     tempo_sampling = 0:periodo_sampling:(periodo_carrier*holdup_time);
19
20     negative=0;
21     positive=0;
22
23     sampleValue = nro_bits;
24     dif_sinal=0;
25     nro_zeros=0;
26     amostra_de_zeros=[];
27     k=1;
28     for i=1:10
29         for j=1:length(tempo_sampling)
30             if(dif_sinal>sampleValue)
31                 if(onda_transmitida(1,k)>0)
32                     positive=1;
33                 end
34                 if(onda_transmitida(1,k)<0)
35                     negative=1;
36                 end
37             end
38             k++;
39             dif_sinal=dif_sinal+1;
40             if(positive==1 && negative==1)
41                 nro_zeros++;
42                 positive=0;
43                 dif_sinal=0;
44                 negative=0;
45             end
46         end
47         amostra_de_zeros=[amostra_de_zeros nro_zeros];
48         nro_zeros=0;
49     end
50
51     %normalize os vetores dividindo-os pela MALVADA
52     amostra_de_zeros=amostra_de_zeros/mean(amostra_de_zeros);
53

```



```

54     filtData=[];
55     for i=1:length(amostra_de_zeros)
56         if(amostra_de_zeros(i)>=1)
57             filtData=[filtData 1];
58         else
59             filtData=[filtData 0];
60         end
61     end
62
63     [BIT_ERROR_RATE NORMALIZED_BIT_ERROR_RATE]=biterr(data,filtData);
64 end
65

```

## 2.1.2 mod

```

1  data = [1 0 1 0 1 1 1 0 0 1]; %defina os bits a serem modulados na onda
2  nro_bits = length(data);
3
4  %DEFINIR SINAL CARRIER
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6  %frequência e período da onda carrier
7  frequencia_carrier = 1000;
8  periodo_carrier = 1/frequencia_carrier;
9
10 %frequência e período que definem a sampling rate(baseado na f e t da carrier)
11 f_sampling = frequencia_carrier * 100;
12 periodo_sampling = 1/f_sampling;
13
14
15 holdup_time = 10;
16 proportional_holdup_time = holdup_time*nro_bits;
17
18 t = 0:periodo_sampling:(proportional_holdup_time*periodo_carrier);
19
20
21 onda_carrier = sin(2*pi*t*frequencia_carrier);
22
23 %PROCESSO DE MODULAÇÃO
24
25 delta_frequencia = 0.5; % o quão violenta vai ser o `amortecimento` nos carregamentos de frequencia
26 frequencia_alta = frequencia_carrier + (frequencia_carrier*delta_frequencia);
27 frequencia_baixa = frequencia_carrier - (frequencia_carrier*delta_frequencia);

```

```

28
29 %definição das frequencias moduladas
30 carrier_alta = sin(2*pi*tempo_sampling*frequencia_alta); %bit alto
31 carrier_baixa = sin(2*pi*tempo_sampling*frequencia_baixa); %bit baixo
32
33 sinal_modulado = [];
34
35 for i=1:nro_bits
36     if(data(i)==1)
37         sinal_modulado = [sinal_modulado carrier_alta];
38     else
39         sinal_modulado = [sinal_modulado carrier_baixa];
40     end
41 end
42
43 ruido = 0.1;
44 onda_transmitida = awgn(sinal_modulado, ruido); %adiciona ruido no sinal

```