

Introdução a Modulação Digital com Áudio

⚠ Este código não funciona em máquinas virtuais ou no WSL

Sobre esse Laboratório:

- [Setup](#)
- [Parametros e Bibliotecas](#)
- [Funções e Metodos](#)
- [O Laboratório](#)
 1. [Etapa 1: Compreendendo com a Codificação/Modulação de dados](#)
 2. [Etapa 2: Decodificação](#)
 3. [Etapa 3: Impacto do Ruído na Comunicação](#)
 4. [Etapa 4: Decodificação no mundo real](#)

Setup

⚠ **Atenção** : Execute o código abaixo **apenas uma vez** para realizar a configuração inicial do ambiente.

```
In [1]: !pip install numpy matplotlib soundfile sounddevice scipy
```

Defaulting to user installation because normal site-packages is not writeable

Requirement already satisfied: numpy in /home/bruno/.local/lib/python3.10/site-packages (2.2.6)

Requirement already satisfied: matplotlib in /home/bruno/.local/lib/python3.10/site-packages (3.10.3)

Requirement already satisfied: soundfile in /home/bruno/.local/lib/python3.10/site-packages (0.13.1)

Requirement already satisfied: sounddevice in /home/bruno/.local/lib/python3.10/site-packages (0.5.2)

Requirement already satisfied: scipy in /home/bruno/.local/lib/python3.10/site-packages (1.15.3)

Requirement already satisfied: pyparsing>=2.3.1 in /usr/lib/python3/dist-packages (from matplotlib) (2.4.7)

Requirement already satisfied: cycler>=0.10 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (0.12.1)

Requirement already satisfied: python-dateutil>=2.7 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (2.9.0.post0)

Requirement already satisfied: pillow>=8 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (11.2.1)

Requirement already satisfied: contourpy>=1.0.1 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (1.3.2)

Requirement already satisfied: packaging>=20.0 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (25.0)

Requirement already satisfied: fonttools>=4.22.0 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (4.58.4)

Requirement already satisfied: kiwisolver>=1.3.1 in /home/bruno/.local/lib/python3.10/site-packages (from matplotlib) (1.4.8)

Requirement already satisfied: cffi>=1.0 in /home/bruno/.local/lib/python3.10/site-packages (from soundfile) (1.17.1)

Requirement already satisfied: pycparser in /home/bruno/.local/lib/python3.10/site-packages (from cffi>=1.0->soundfile) (2.22)

Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)

escolhendo a saída e entrada de áudio

```
In [2]: import os

import sounddevice as sd
import numpy as np
from scipy.io import wavfile
```

O comando `sd.query_devices()` listará todas as **entradas** e **saídas** de áudio disponíveis no seu computador.

A partir dessa lista, identifique qual é a sua **entrada** (microfone) e **saída** (alto-falante) de áudio desejada e atribua esses valores às variáveis `input_device` e `output_device`, respectivamente. Por exemplo,

```
5 LG ULTRA WIDE (HD Audio Driver f, MME (0 in, 2 out)
6 Alto-falantes (3- Realtek(R) Au, MME (0 in, 2 out)
> 7 Driver de captura de som primário, Windows
DirectSound (2 in, 0 out)
8 Grupo de microfones (3- Tecnologia Intel® Smart Sound
para microfones digitais), Windows DirectSound (2 in, 0
out)
```

```

    9 Headset (QCY MeloBuds Pro), Windows DirectSound (1
in, 0 out)
< 10 Driver de som primário, Windows DirectSound (0 in, 2
out)
    11 Fones de ouvido (QCY MeloBuds Pro), Windows
DirectSound (0 in, 8 out)
    12 LG ULTRA WIDE (HD Audio Driver for Display Audio),
Windows DirectSound (0 in, 2 out)

```

Para que o laboratório funcione adequadamente no meu computador devo escolher opções 7, 10.

```
In [3]: sd.query_devices()
```

```

Out[3]:  0 sof-hda-dsp: - (hw:0,0), ALSA (2 in, 0 out)
        1 sof-hda-dsp: - (hw:0,3), ALSA (0 in, 2 out)
        2 sof-hda-dsp: - (hw:0,4), ALSA (0 in, 2 out)
        3 sof-hda-dsp: - (hw:0,5), ALSA (0 in, 2 out)
        4 sof-hda-dsp: - (hw:0,6), ALSA (2 in, 0 out)
        5 sof-hda-dsp: - (hw:0,7), ALSA (2 in, 0 out)
        6 sysdefault, ALSA (128 in, 0 out)
        7 samplerate, ALSA (128 in, 0 out)
        8 speexrate, ALSA (128 in, 0 out)
        9 jack, ALSA (2 in, 2 out)
       10 pipewire, ALSA (64 in, 64 out)
       11 pulse, ALSA (32 in, 32 out)
       12 upmix, ALSA (8 in, 0 out)
       13 vdownmix, ALSA (6 in, 0 out)
      * 14 default, ALSA (32 in, 32 out)
       15 Tiger Lake-LP Smart Sound Technology Audio Controller Digital Micro
phone, JACK Audio Connection Kit (2 in, 0 out)
       16 Tiger Lake-LP Smart Sound Technology Audio Controller Headphones St
ereo Microphone, JACK Audio Connection Kit (2 in, 0 out)
       17 Google Chrome, JACK Audio Connection Kit (2 in, 0 out)
       18 Tiger Lake-LP Smart Sound Technology Audio Controller Speaker + Hea
dphones, JACK Audio Connection Kit (0 in, 0 out)
       19 Tiger Lake-LP Smart Sound Technology Audio Controller HDMI / Displa
yPort 1 Output, JACK Audio Connection Kit (2 in, 2 out)
       20 Tiger Lake-LP Smart Sound Technology Audio Controller HDMI / Displa
yPort 2 Output, JACK Audio Connection Kit (2 in, 2 out)
       21 Tiger Lake-LP Smart Sound Technology Audio Controller HDMI / Displa
yPort 3 Output, JACK Audio Connection Kit (2 in, 2 out)

```

```

In [ ]: output_device = 14
        input_device = 14
        sd.default.device = (input_device, output_device)

```

🔗 Execute o código abaixo para gravar um áudio de 3 segundos e, em seguida, reproduzi-lo.

Caso a gravação ou reprodução não funcione corretamente, ajuste os valores das variáveis `input_device` (dispositivo de entrada) e `output_device` (dispositivo de saída) até que o código funcione como esperado.

```

In [4]: DURATION = 3 # duracao em segundos
        SAMPLE_RATE = 44100 # Hz

```

```

FILENAME = "captura.wav"

audio = sd.rec(int(DURATION * SAMPLE_RATE), samplerate=SAMPLE_RATE, channels=2, dtype=np.float32)
sd.wait()
print("Gravação finalizada.")

# Salvando arquivo de audio temporario
max_val = np.max(np.abs(audio))
if max_val > 0:
    scaled = audio / max_val
else:
    scaled = audio
wav_data = np.int16(scaled * 32767)
wavfile.write(FILENAME, SAMPLE_RATE, wav_data)
print(f"Áudio salvo em {FILENAME}")

# Reproduz o áudio salvo
fs, data = wavfile.read(FILENAME)
sd.play(data, fs)
sd.wait()
os.remove(FILENAME)
print('Se você escutou o audio tudo seu sistema está configurado corretamente!')

```

Gravação finalizada.

Áudio salvo em captura.wav

Se você escutou o audio tudo seu sistema está configurado corretamente!

Parametros de configurações

```

In [5]: import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf
import sounddevice as sd
from scipy import signal
import time

```

parametros de codificação

```

In [6]: ##
## configuracoes do audio (detalhes na secção de setup)
##

# output_device = 10
# input_device = 7
output_device = 14
input_device = 14
sd.default.device = (input_device, output_device)

##
## Configurações globais do exercio
##
SAMPLE_RATE = 44100 # Taxa de amostragem do audio
BIT_DURATION = 1.0 # 1 segundo por bit
FREQ_LOW = 440 # bit '0' (Lá)
FREQ_HIGH = 880 # bit '1' (Lá oitava)

```

Funções

Nessa secao vocs encontraram as funcoes utilizadas

```
In [7]: import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf
import sounddevice as sd
from scipy import signal
import time
```

Gerador de tom & visualizacao

```
In [8]: def generate_tone(frequency, duration, sample_rate=SAMPLE_RATE):
        """
        Gera um tom senoidal

        Args:
            frequency: Frequência em Hz
            duration: Duração em segundos
            sample_rate: Taxa de amostragem

        Returns:
            array: Sinal de áudio
        """
        t = np.linspace(0, duration, int(sample_rate * duration), False)
        # Aplica janela para suavizar transições
        tone = np.sin(2 * np.pi * frequency * t)
        # Janela de Hanning para evitar cliques
        window = np.hanning(len(tone))
        return tone * window
```

```
In [9]: def show(data:str, debug):
        if debug==True:
            print(data)
```

```
In [10]: def plot_signal(audio_signal, title, num_bits):
        time_axis = np.linspace(0, len(audio_signal)/SAMPLE_RATE, len(audio_s

        plt.figure(figsize=(12, 4))
        plt.plot(time_axis, audio_signal)
        plt.title(title)
        plt.xlabel('Tempo (s)')
        plt.ylabel('Amplitude')
        plt.grid(True, alpha=0.3)

        for i in range(1, num_bits): #divisões dos bits
            plt.axvline(x=i*BIT_DURATION, color='red', linestyle='--', alpha=

        plt.tight_layout()
        plt.show()
```

Codificadores

```
In [11]: def encode_nrz(data_bits, debug=False):
        """
        Codifica dados usando NRZ
```

```

Args:
    data_bits: string de bits (ex: "10110")

Returns:
    array: Sinal de áudio codificado
"""
audio_signal = np.array([])

show(f"Codificando NRZ: {data_bits}", debug)

for i, bit in enumerate(data_bits):
    if bit == '1':
        freq = FREQ_HIGH
        show(f"Bit {i}: '1' -> {freq} Hz", debug)
    else:
        freq = FREQ_LOW
        show(f"Bit {i}: '0' -> {freq} Hz", debug)

    tone = generate_tone(freq, BIT_DURATION)
    audio_signal = np.concatenate([audio_signal, tone])

return audio_signal

```

In [12]: `def encode_nrzi(data_bits, debug=False):`

```

"""
Codifica dados usando NRZI

Args:
    data_bits: string de bits

Returns:
    array: Sinal de áudio codificado
"""
pass

return '0'

```

In [13]: `def encode_manchester(data_bits, debug=False):`

```

"""
Codifica dados usando Manchester

Args:
    data_bits: string de bits

Returns:
    array: Sinal de áudio codificado
"""
audio_signal = np.array([])

show(f"Codificando Manchester: {data_bits}", debug)

for i, bit in enumerate(data_bits):
    if bit == '1':
        # Bit '1': alto->baixo (primeira metade alta, segunda baixa)
        tone1 = generate_tone(FREQ_HIGH, BIT_DURATION/2)
        tone2 = generate_tone(FREQ_LOW, BIT_DURATION/2)
        show(f"Bit {i}: '1' -> {FREQ_HIGH}Hz -> {FREQ_LOW}Hz", debug)
    else:

```

```

# Bit '0': baixo->alto (primeira metade baixa, segunda alta)
tone1 = generate_tone(FREQ_LOW, BIT_DURATION/2)
tone2 = generate_tone(FREQ_HIGH, BIT_DURATION/2)
show(f"Bit {i}: '0' -> {FREQ_LOW}Hz -> {FREQ_HIGH}Hz", debug)

bit_signal = np.concatenate([tone1, tone2])
audio_signal = np.concatenate([audio_signal, bit_signal])

return audio_signal

```

Decodificadores

Detector de frequência

Neste trecho de código, vamos utilizar a **Transformada Rápida de Fourier (FFT)** para detectar frequências dominantes em segmentos de áudio — uma ferramenta para análise espectral de sinais. Utilizamos esse conceito em sala de aula para ilustrar o conceito de **modulação por divisão de frequência (FDM)**. Esses códigos detectaram os bits **0** e **1** em razão das frequências que estabelessemos.

A função `detect_frequency` tem como objetivo identificar as frequências que compõem um sinal de áudio modulado. Na Figura 1, vemos duas senoides (uma azul e uma vermelha), representando frequências distintas associadas aos bits **'0'** e **'1'**. Essas senoides são combinadas para formar o sinal apresentado na Figura 2, o qual será analisado.

Figura 1

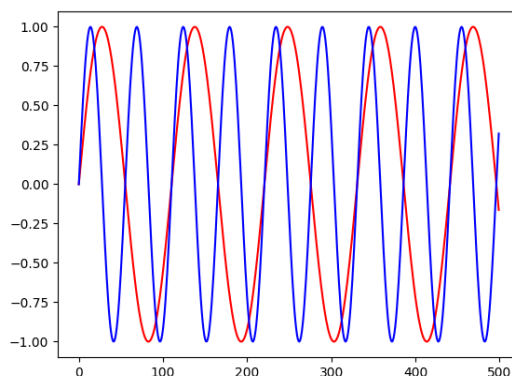
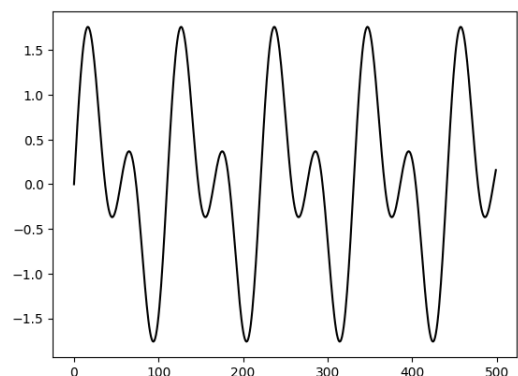
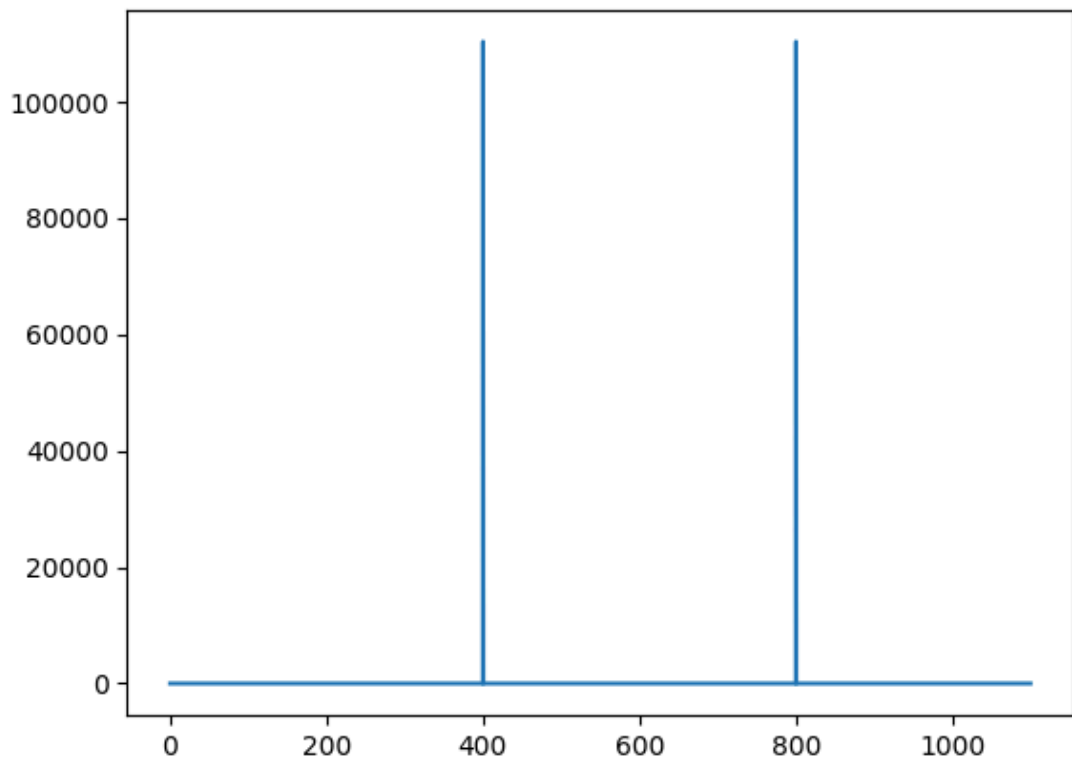


Figura 2




O código utiliza a FFT para detectar os picos de frequência presentes no sinal composto. Esses picos — visíveis no gráfico espectral gerado pela FFT (Figura 3) — correspondem exatamente às componentes originais da modulação, permitindo que cada segmento de áudio seja interpretado como **'0'** ou **'1'** com base em qual frequência está presente.



Assim como vimos em sala de aula, essa abordagem ilustra o processo de demodulação por frequência, fundamental no modelo de transmissão FDM (Frequency Division Multiplexing).

Já a função `frequency_to_bit` interpreta a frequência detectada como um bit binário, comparando-a com um limiar: se for superior ao valor definido, retorna `'1'`; caso contrário, `'0'`. Esse processo simula uma decodificação simples de sinais FDM com base na presença de faixas de frequência distintas.

 **Observação:** valores como `SAMPLE_RATE` e o `threshold` devem estar de acordo com as frequências utilizadas (já configurados no início do notebook) durante a modulação para garantir a correta detecção dos dados transmitidos.

```
In [14]: def detect_frequency(audio_segment, sample_rate=SAMPLE_RATE):
    """
    Detecta a frequência dominante em um segmento de áudio

    Args:
        audio_segment: Segmento de áudio
        sample_rate: Taxa de amostragem

    Returns:
        float: Frequência detectada
    """
    # FFT para análise espectral
    fft = np.fft.fft(audio_segment)
    freqs = np.fft.fftfreq(len(fft), 1/sample_rate)

    # Considera apenas frequências positivas
    magnitude = np.abs(fft[:len(fft)//2])
    freqs_positive = freqs[:len(freqs)//2]
```



```

# Encontra o pico de frequência
peak_idx = np.argmax(magnitude)
detected_freq = abs(freqs_positive[peak_idx])

return detected_freq

def frequency_to_bit(frequency, threshold=660):
    """
    Converte frequência detectada em bit

    Args:
        frequency: Frequência detectada
        threshold: Limiar para decisão (média entre FREQ_LOW e FREQ_HIGH)

    Returns:
        str: '0' ou '1'
    """
    return '1' if frequency > threshold else '0'

```

Decoders

```

In [ ]: def decode_nrz(audio_signal, num_bits, sample_rate=SAMPLE_RATE, debug=False)
        """
        Decodifica sinal NRZ

        Args:
            audio_signal: Sinal de áudio
            num_bits: Número esperado de bits
            sample_rate: Taxa de amostragem

        Returns:
            str: Bits decodificados
        """
        samples_per_bit = int(sample_rate * BIT_DURATION)
        decoded_bits = ""

        show("Decodificando NRZ:", debug)

        for i in range(num_bits):
            start_idx = i * samples_per_bit
            end_idx = start_idx + samples_per_bit

            if end_idx > len(audio_signal):
                show(f"Aviso: Áudio muito curto para {num_bits} bits", debug)
                break

            # Analisa o meio do bit para evitar transições
            mid_start = start_idx + samples_per_bit // 4
            mid_end = end_idx - samples_per_bit // 4
            segment = audio_signal[mid_start:mid_end]

            freq = detect_frequency(segment, sample_rate)
            bit = frequency_to_bit(freq)
            decoded_bits += bit

            show(f"Bit {i}: freq={freq:.1f}Hz -> '{bit}'", debug)

        return decoded_bits

```

```
In [53]: def decode_nrzi(audio_signal, num_bits, sample_rate=SAMPLE_RATE, debug=False)
        """
        Decodifica sinal NRZ com ruído

        Args:
        audio_signal: Sinal de áudio
        num_bits: Número esperado de bits
        sample_rate: Taxa de amostragem

        Returns:
        str: Bits decodificados
        """

        samples_per_bit = int(sample_rate * BIT_DURATION)
        decoded_bits = ""

        show("Decodificando NRZ:", debug)

        for i in range(num_bits):
            start_idx = i * samples_per_bit
            end_idx = start_idx + samples_per_bit

            if end_idx > len(audio_signal):
                show(f"Aviso: Áudio muito curto para {num_bits} bits", debug)
                decoded_bits += '?' * (num_bits - i) # sinaliza erro
                break

            # Analisa o meio do bit para evitar transições
            mid_start = start_idx + samples_per_bit // 4
            mid_end = end_idx - samples_per_bit // 4
            segment = audio_signal[mid_start:mid_end]

            freq = detect_frequency(segment, sample_rate)
            bit = frequency_to_bit(freq)
            decoded_bits += bit

            show(f"Bit {i}: freq={freq:.1f}Hz -> '{bit}'", debug)

        return '0'
```

```
In [17]: def decode_manchester(audio_signal, num_bits, sample_rate=SAMPLE_RATE, debug=False)
        """
        Decodifica sinal Manchester
        """

        samples_per_bit = int(sample_rate * BIT_DURATION)
        decoded_bits = ""

        show("Decodificando Manchester:", debug)

        for i in range(num_bits):
            start_idx = i * samples_per_bit
            end_idx = start_idx + samples_per_bit

            if end_idx > len(audio_signal):
                break

            # Analisa primeira e segunda metade do bit
            mid_point = start_idx + samples_per_bit // 2

            # Primeira metade
```

```

first_half = audio_signal[start_idx + samples_per_bit//8 : mid_po
freq1 = detect_frequency(first_half, sample_rate)
state1 = frequency_to_bit(freq1)

# Segunda metade
second_half = audio_signal[mid_point + samples_per_bit//8 : end_i
freq2 = detect_frequency(second_half, sample_rate)
state2 = frequency_to_bit(freq2)

# Determina o bit baseado na transição
if state1 == '1' and state2 == '0': # Alto -> Baixo
    bit = '1'
    show(f"Bit {i}: {freq1:.1f}Hz -> {freq2:.1f}Hz = alto->baixo
elif state1 == '0' and state2 == '1': # Baixo -> Alto
    bit = '0'
    show(f"Bit {i}: {freq1:.1f}Hz -> {freq2:.1f}Hz = baixo->alto
else: # Erro de decodificação
    bit = '?'
    show(f"Bit {i}: {freq1:.1f}Hz -> {freq2:.1f}Hz = ERRO na tran

decoded_bits += bit

return decoded_bits

```

O Laboratório

```

In [18]: ##
## configuracoes do audio (detalhes na secção de setup)
##
# output_device = 10
# input_device = 7
output_device = 14
input_device = 14
sd.default.device = (input_device, output_device)

##
## Configurações globais do exercio
##
SAMPLE_RATE = 44100 # Taxa de amostragem do audio
BIT_DURATION = 1.0 # 1 segundo por bit
FREQ_LOW = 440 # bit '0' (Lá)
FREQ_HIGH = 880 # bit '1' (Lá oitava)

```

Etapla 1: Compreendendo com a Codificação/Modulação de dados

Nessa seção você deve se familiarizar com algumas das funções de codificação (modulação) vista em sala de aula

```

In [19]: test_bits = "11001"
print(f"Dados originais: {test_bits}\n")

```

Dados originais: 11001

```

In [20]: # Testa cada modulação
print("1. NRZ:")

```

```
nrz_signal = encode_nrz(test_bits, debug=True)

print("\n3. Manchester:")
manchester_signal = encode_manchester(test_bits, debug=True)
```

1. NRZ:

Codificando NRZ: 11001

Bit 0: '1' -> 880 Hz

Bit 1: '1' -> 880 Hz

Bit 2: '0' -> 440 Hz

Bit 3: '0' -> 440 Hz

Bit 4: '1' -> 880 Hz

3. Manchester:

Codificando Manchester: 11001

Bit 0: '1' -> 880Hz -> 440Hz

Bit 1: '1' -> 880Hz -> 440Hz

Bit 2: '0' -> 440Hz -> 880Hz

Bit 3: '0' -> 440Hz -> 880Hz

Bit 4: '1' -> 880Hz -> 440Hz

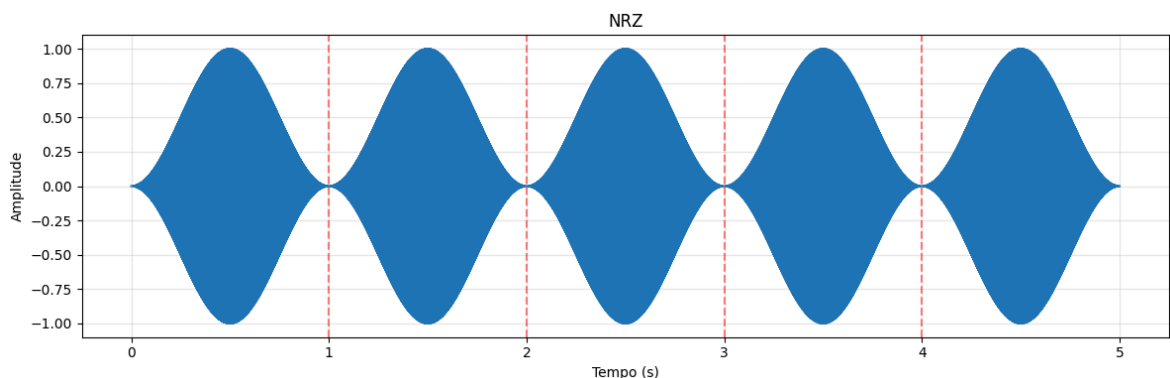
Escutando os dados como a nossa proposta é utilizar notas musicais (som) para representar **0** e **1**. Vamos escutalos.

```
In [21]: sd.play(manchester_signal, SAMPLE_RATE)
sd.wait()
```

```
In [22]: sd.play(nrz_signal, SAMPLE_RATE)
sd.wait()
```

Você pode visualizar a modulação utilizando plots, com abaixo

```
In [23]: plot_signal(nrz_signal, 'NRZ', len(test_bits))
```



Esse modulação foi utilizada nos primórdios da internet (internet discada)



A internet discada utilizava modulação analógica para transportar dados digitais pela linha telefônica. Nos primeiros modems — como os que seguiram o padrão Bell 103 — a técnica predominante foi a Frequency Shift Keying (FSK), onde duas frequências distintas (uma para o bit 0 e outra para o bit 1) eram usadas para representar os dados.

Para quem não teve a oportunidade de utilizar a internet discada o video abaixo ilustra bem o processo de modulação utilizada

Som da conexão discada (Dial-up)



Etapa 2: Decodificação

Nesta etapa vocês irão recuperar os dados originais a partir do sinal de áudio recebido. Este é o papel do receptor em um sistema de comunicação.

O primeiro passo para a decodificação (demodulação) em um mundo ideal é salvar o resultado da modulação em um arquivo de audio (.wav)

```
In [24]: # Dados de teste
test_data = "1010100000001111110000010101010111000"

print(f"Criando arquivos de teste para: {test_data}")

# NRZ
nrz_signal = encode_nrz(test_data)
sf.write('teste_nrz.wav', nrz_signal, SAMPLE_RATE)
print("\t ✓ Arquivo teste_nrz.wav criado")

# Manchester
manchester_signal = encode_manchester(test_data)
sf.write('teste_manchester.wav', manchester_signal, SAMPLE_RATE)
print("\t ✓ Arquivo teste_manchester.wav criado")
```

Criando arquivos de teste para: 1010100000001111110000010101010111000
 ✓ Arquivo teste_nrz.wav criado
 ✓ Arquivo teste_manchester.wav criado

```
In [25]: original_data = test_data

print(f"\nDados originais: {original_data}")
print(f"Número de bits: {len(original_data)}\n")
```

Dados originais: 1010100000001111110000010101010111000
 Número de bits: 37

Para evitar efeitos de atenuação e interferências vamos realizar a decodificação diretamente do arquivo de audio.

```
In [26]: # Testa decodificação NRZ
print("1. Decodificando NRZ:")
nrz_audio, _ = sf.read('teste_nrz.wav')
decoded_nrz = decode_nrz(nrz_audio, len(original_data))
print(f"Original: {original_data}")
print(f"Decodificado: {decoded_nrz}")
print(f"Correto: {original_data == decoded_nrz}\n")
```

1. Decodificando NRZ:
 Original: 1010100000001111110000010101010111000
 Decodificado: 1010100000001111110000010101010111000
 Correto: True

```
In [27]: # Testa decodificação Manchester
print("3. Decodificando Manchester:")
manchester_audio, _ = sf.read('teste_manchester.wav')
decoded_manchester = decode_manchester(manchester_audio, len(original_data))
print(f"Original: {original_data}")
print(f"Decodificado: {decoded_manchester}")
print(f"Correto: {original_data == decoded_manchester}")
```

3. Decodificando Manchester:
 Original: 1010100000001111110000010101010111000
 Decodificado: 1010100000001111110000010101010111000
 Correto: True

```
In [30]: # Decodificacao aquivo de dados 6
```

```

audio, _ = sf.read('dados_codificados/dados_6_44100hz.wav')
num_bits = len(audio) // SAMPLE_RATE
print(f"numero de bits estimado: {num_bits}")
bits_manchester = decode_manchester(audio, num_bits)
bits_nrz = decode_nrz(audio, num_bits)
print(f"decode manchester: {bits_manchester}")
print(f"decode nrz: {bits_nrz}")

def bits_to_ascii(bits):
    chars = []
    for i in range(0, len(bits), 8):
        byte = bits[i:i+8]
        if len(byte) == 8 and '?' not in byte:
            chars.append(chr(int(byte, 2)))
        else:
            chars.append('?')
    return ''.join(chars)

print("Manchester ->", bits_to_ascii(bits_manchester))
print("NRZ ->", bits_to_ascii(bits_nrz))

```

```

numero de bits estimado: 15
decode manchester: 010111001000001
decode nrz: 101000110111110
Manchester -> \?
NRZ -> f?

```

Etapa 3: Impacto do Ruído na Comunicação

Simulação de decodificação em condições adversas, o metodo abaixo simula a adição de ruído ao sinal.

```

In [31]: def adicionar_ruído(audio_signal, snr_db=-12):
        """
        Adiciona ruído gaussiano ao sinal

        Args:
            audio_signal: Sinal original
            snr_db: Relação sinal-ruído em dB

        Returns:
            array: Sinal com ruído
        """
        # Calcula potência do sinal
        signal_power = np.mean(audio_signal ** 2)

        # Calcula potência do ruído baseada no SNR
        snr_linear = 10 ** (snr_db / 10)
        noise_power = signal_power / snr_linear

        # Gera ruído gaussiano
        noise = np.random.normal(0, np.sqrt(noise_power), len(audio_signal))

        return audio_signal + noise

```

Para adicionar ruido utilize um valor **negativo** no `snr_db` . Por exemplo se você quiser um ruido de 3db utilize `snr_db=-3` no metodo.

A baixo veja a mensagem original

```
In [40]: original_bits = "00111000"
```

```
In [33]: snr=-3
```

```
clean_signal = encode_nrz(original_bits)

noisy_signal = adicionar_ruido(clean_signal, snr)
decoded = decode_nrz(noisy_signal, len(original_bits))
print(f" Original: {original_bits}")
print(f" Decodificado: {decoded}")
print(f" Correto: {original_bits == decoded}\n")
```

```
Original: 00111000
Decodificado: 00111000
Correto: True
```

```
In [69]: import numpy as np
import matplotlib.pyplot as plt

snr_values = np.arange(-65, 0, 1)

erros_nrz = []
erros_manchester = []

original_bits = '0' * 8

for snr in snr_values:
    signal_nrz = encode_nrz(original_bits)
    signal_manchester = encode_manchester(original_bits)

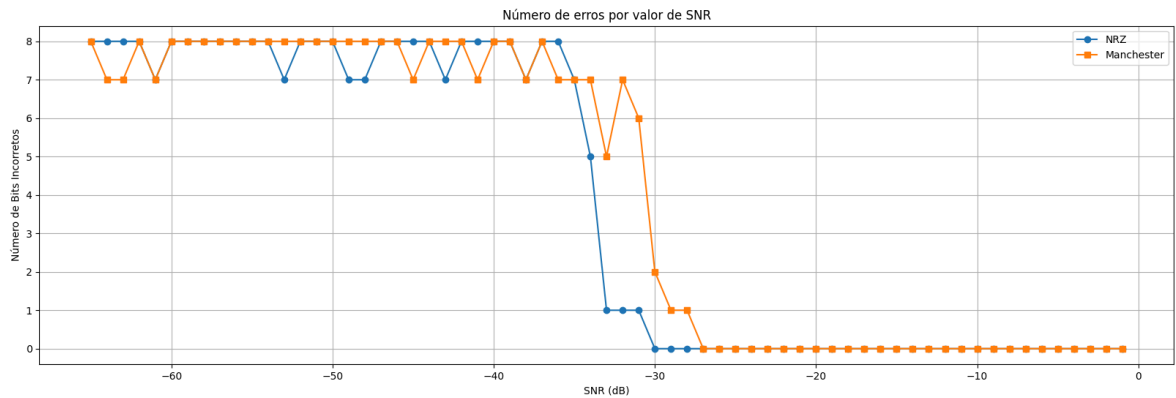
    noisy_nrz = adicionar_ruido(signal_nrz, snr)
    noisy_manchester = adicionar_ruido(signal_manchester, snr)

    decoded_nrz = decode_nrz(noisy_nrz, len(original_bits))
    decoded_manchester = decode_manchester(noisy_manchester, len(original_bits))

    # Contar erros
    erro_nrz = sum(o != d for o, d in zip(original_bits, decoded_nrz))
    erro_manchester = sum(o != d for o, d in zip(original_bits, decoded_manchester))

    erros_nrz.append(erro_nrz)
    erros_manchester.append(erro_manchester)

plt.figure(figsize=(20, 6))
plt.plot(snr_values, erros_nrz, label='NRZ', marker='o')
plt.plot(snr_values, erros_manchester, label='Manchester', marker='s')
plt.xlabel('SNR (dB)')
plt.ylabel('Número de Bits Incorretos')
plt.title('Número de erros por valor de SNR')
plt.grid(True)
plt.legend()
plt.show()
```

A3.1 : A partir de que nível de ruído, para cada modulação, o sistema começa a falhar?

a) Identifique o valor de SNR onde os primeiros bits são comprometidos

- Para a modulação **Manchester**, os primeiros erros ocorrem a partir de aproximadamente **SNR = -27 dB**.
- Para a modulação **NRZ**, os primeiros erros ocorrem a partir de aproximadamente **SNR = -30 dB**.

b) Identifique o valor de SNR onde todos os bits são comprometidos

- Para a modulação **Manchester**, todos os bits estão errados a partir de **SNR = -36 dB**.
- Para a modulação **NRZ**, todos os bits estão errados a partir de **SNR = -35 dB**.

Etapa 4: Decodificação no mundo real

```
In [70]: def capturar_do_microfone(duracao_segundos):
    """
    Captura áudio do microfone

    Args:
        duracao_segundos: Duração da captura

    Returns:
        array: Áudio capturado
    """
    print(f"Iniciando captura por {duracao_segundos} segundos...")
    print("Reproduza o áudio no seu celular AGORA!")

    # Captura áudio
    audio_capturado = sd.rec(
        int(duracao_segundos * SAMPLE_RATE),
        samplerate=SAMPLE_RATE,
        channels=1
    )
    sd.wait() # Aguarda terminar a captura
```

```
print("Captura concluída!")
return audio_capturado.flatten()
```

```
In [71]: import random
import csv

def gerar_string_binaria(n):
    return ''.join(random.choice('01') for _ in range(n))

def gerar_questao(n, start=8, stop=16):
    dados = []
    for i in range(n):
        n_bits = random.randrange(start, stop)
        msg = gerar_string_binaria(n_bits)
        encoder = random.choice([encode_manchester, encode_nrz])
        modulacao = encoder.__name__
        nome = f"dados_{i}_{SAMPLE_RATE}hz.wav"
        sinal = encoder(msg)
        sf.write(nome, sinal, SAMPLE_RATE)

        linha = {
            'arquivo': nome,
            'msg': msg,
            'n_bits': n_bits,
            'modulacao': modulacao
        }
        dados.append(linha)
        # print(n_bits, msg, modulacao, nome, len(sinal))
    # print(dados):

    nome_arquivo = 'gabarito.csv'

    # Escrita no arquivo CSV
    with open(nome_arquivo, mode='w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=dados[0].keys())
        writer.writeheader()
        writer.writerows(dados)

gerar_questao(50)
```

Para este exercício, você deverá utilizar um **segundo dispositivo**, como por exemplo, seu **celular**.

1. Copie o arquivo de áudio `dados_ar.wav` para o segundo dispositivo.
2. Este áudio contém uma **mensagem de 5 bits**, codificada utilizando o esquema **Manchester**.
3. O desafio consiste em **decodificar essa mensagem utilizando apenas o microfone do seu computador**.

Não abra o arquivo diretamente no computador — apenas reproduza o áudio no segundo dispositivo.



Certifique-se de que os [procedimentos de configuração](#) foram seguidos corretamente e que o microfone do seu computador está

funcionando adequadamente.

Por fim, **execute o código abaixo** (referente à etapa de escuta por microfone) e reproduza o áudio no segundo dispositivo para tentar decodificar a mensagem.

```
In [76]: # test_data = "10110"
# Captura áudio

duracao = 5 * BIT_DURATION + 1 # +1 segundo de margem
audio_capturado = capturar_do_microfone(duracao)

# Salva captura para análise
sf.write('captura_microfone.wav', audio_capturado, SAMPLE_RATE)

# Tenta decodificar
print("\nTentando decodificar...")
decoded = decode_manchester(audio_capturado, 5)

print(f"Original: ?????")
print(f"Capturado: {decoded}")
```

Iniciando captura por 6.0 segundos...
Reproduza o áudio no seu celular AGORA!
Captura concluída!

Tentando decodificar...
Original: ?????
Capturado: 10110