Universidade de Brasília
Departamento da Ciência da Computação
Teleinformática e Redes 1

Simre;Camfe

Simulador de Redes: Camadas Física e Enlace

Grupo: 03

Henrique França Mat: 242039130
José Antônio de Andrade Mat: 232013031
Rafael Sapienza Mat: 232006215

Professor:
Marcelo Antônio Marotta

25 de novembro de 2025

1 Introdução

A comunicação de dados em redes de computadores depende de uma série de procedimentos organizados em camadas, cada uma responsável por uma parte específica do processo de transmissão. Entre essas etapas, destacam-se a camada física e a camada de enlace, fundamentais para garantir que a informação seja corretamente convertida, modulada, transmitida, enquadrada, analisada e reconstruída no destino. O objetivo deste trabalho é desenvolver um simulador capaz de representar, de maneira modular, o funcionamento dessas duas camadas utilizando técnicas clássicas de modulação e protocolos de enquadramento, detecção e correção de erros.

Na camada física, o simulador implementa modulações digitais do tipo NRZ-Polar, Manchester e Bipolar, além de modulações por portadora como ASK, FSK, QPSK e 16-QAM, reproduzindo a forma como sinais elétricos ou ondas são utilizados para transportar bits no meio físico.

Já na camada de enlace, o simulador incorpora diferentes protocolos de enquadramento — incluindo contagem de caracteres, FLAGS com inserção de bytes e FLAGS com inserção de bits — que delimitam os quadros e evitam ambiguidades durante a recepção. Em seguida, são aplicados mecanismos de detecção de erros, como bit de paridade, Checksum e CRC-32, além de um método de correção de erros baseado no código de Hamming, permitindo identificar ou corrigir falhas introduzidas no fluxo de bits durante a transmissão.

O sistema integra essas funcionalidades em uma interface gráfica, permitindo visualizar passo a passo o processo de envio e recepção de mensagens. Assim, o simulador oferece uma compreensão prática e unificada dos principais mecanismos que tornam a comunicação de dados confiável.

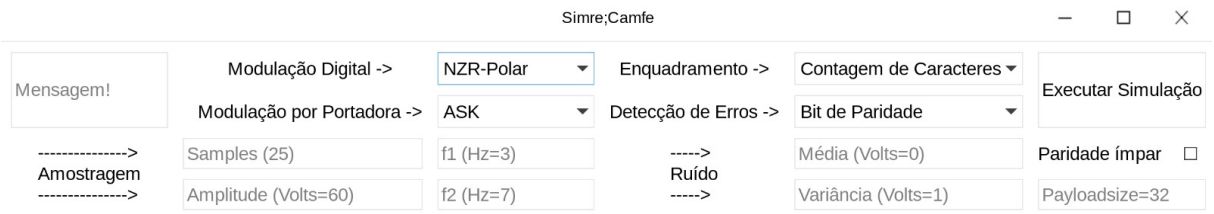

Figura 1: GUI do simulador

1.1 Instalação

Para executar o projeto, deve-se:

1. Instalar Python 3.x no sistema.
2. Instalar GTK no sistema.
3. Criar um Virtual Environment para o python.
4. Usar o pip para instalar: PyGObject (gtk), matplotlib, numpy e socket.
5. Executar o arquivo `InterfaceGrafica.py`.

2 Implementação

2.1 Camada Física

O funcionamento da Camada Física é bem simples do ponto de vista completo do protocolo. Todas os métodos estabelecidos no trabalho (no arquivo `CamadaFisica.py`) recebem uma lista de booleanos e parâmetros relacionados ao método de codificação, e retornam uma lista de valores de tensão que simulam uma onda eletromagnética. Similarmente, na decodificação, recebem uma lista de amostras de tensão e convertem para o valor aproximado mais provável (seja este verdadeiro ou falso). Segue exemplo do método na Listagem 1 com Amplitude Shift Keying.

Listagem 1: (De)Modulação ASK

```
import numpy as np
import numpy.typing as npt

def ASK_modulation(digitals: list[bool], *,
                   smplcnt: int = 220,
                   f: float = 100,
                   amp: float = 500,
                   lowwave: bool = False) -> npt.NDArray:

    # Variaveis basicas
    result: npt.NDArray = np.empty(0)
    period: np.float64 = np.pi * 2 * f

    # Pre-calculos para eficiencia
    ## Cria uma lista com smplcnt valores de 0 ate 2*pi*f e usa-os para construir uma onda
    rOne: npt.NDArray = np.sin(np.arange(0, period, period / smplcnt)) * amp
    ## Apenas zeros
    rZero: npt.NDArray = np.zeros(smplcnt)

    # Criacao de sinal
    for b in digitals:
        # Checa se b eh 0/1, e entao inverte a codificacao se tal foi requisitado
        result = np.append(result, rOne if (b ^ lowwave) else rZero)

    return result

def ASK_demodulation(signal: npt.NDArray, *,
                   smplcnt: int = 220,
                   f: float = 100,
                   amp: float = 500,
                   lowwave: bool = False) -> list[bool]:

    # Pre-calculos para eficiencia
    period: np.float64 = np.pi * 2 * f
    ## Valor esperado para onda "1"
    eOne: npt.NDArray = np.sin(np.arange(0, period, period / smplcnt)) * amp
    ## Valor esperado para onda "0"
    eZero: npt.NDArray = np.zeros(smplcnt)

    # Itera pelos chunks de samples de cada codigo (smplcnt)
```

```
    result: list[bool] = []
    for i in range(0, len(signal), smplcnt):
        # Checa a diferenca absoluta acumulada entre o sinal recebido
        # e as ondas esperadas.
        absdiffOne: np.float64 = np.sum(np.abs(signal[i:i+smplcnt] - eOne))
        absdiffZero: np.float64 = np.sum(np.abs(signal[i:i+smplcnt] - eZero
))

        # O menor valor eh escolhido, seguindo escolha da inversao de
codificacao
        result.append(bool(absdiffOne < absdiffZero) ^ lowwave)

    return result
```

No caso de modulações digitais, os métodos recebem uma lista de booleanos e retornam uma lista de inteiros, representando a devida modulação. Em geral, tais valores são 1, 0 e -1. Para cada modulação, um método construtor distinto deve ser aplicado mas, em geral, sua formação depende de valores anteriores, diferente dos métodos de portadora que todos apenas se interessam no valor do bit atual. Segue exemplo da (de)modulação bipolar, na Listagem 2.

Para a geração de valores de tensão nas modulações de portadora, usa-se uma função seno com os devidos parâmetros fornecidos (amplitude, frequência e fase) e divide-se um período completo de acordo com os parâmetros de amostragem. Especificamente, um período completo é dividido de tal forma que, para N amostras, o intervalo entre elas seja $2\pi/N$. Por exemplo, para uma onda de 1Hz, com 4 amostras teremos a lista de tensão $[0, A, 0, -A]$ onde A é o valor máximo de tensão. Cada símbolo é gerado individualmente de acordo com o protocolo escolhido, e então utilizado na formação da lista final de valores de tensão.

Para a demodulação, compara blocos de amostras do sinal recebido e um conjunto de ondas de referência pré-computadas para determinar qual é o símbolo recebido. Utiliza-se como critério a diferença absoluta acumulada e, se necessário, a energia do sinal.

Listagem 2: (De)Modulação Bipolar

```
def bipolar(bit_string: list[bool]) -> list[int]:
    # Lista de resultados
    sinal_codificado = []
    # Valor inicial da codificacao
    aux = 1

    for bit in bit_string:
        if not bit:
            # Zero
            sinal_codificado.append(0)
        else:
            # Um (+-1)
            sinal_codificado.append(aux)
            aux = -aux  # Trocar polaridade apos todo "1"

    return sinal_codificado

def bipolar_decoder(bit_string: list[int]) -> list[bool]:
    # Lista de resultados
    sinal_decodificado = []
    # Valor inicial da codificacao
    aux = 1

    for bit in bit_string:
        if bit > -0.5 and bit <= 0.5:
```

```
            # Zero
            sinal_decodificado.append(False)
        elif bit > 0.5:
            # Um (+1)
            if aux == -1:
                # Erro na codificacao
                return None
            sinal_decodificado.append(True)
            aux = -aux
        else:
            # Um (-1)
            if aux == 1:
                # Erro na codificacao
                return None
            sinal_decodificado.append(True)
            aux = -aux

    return sinal_decodificado
```

No sistema, a Camada Física entra logo antes da mensagem ser enviada para o socket — momento em que todo o protocolo da Camada de Enlace já foi aplicado — para simular o trajeto de transmissão. Ruído é adicionado logo em seguida e, após o socket receber a mensagem, tal é decodificada pelo mesmo método escolhido pelo usuário. Por prática, mesmo que o sinal esteja irreconhecível devido ao erro, todos métodos de portadora retornarão algum resultado que pode ser, então, analizado pela Camada de Enlace. A aplicação dos métodos para simulação está disposta na Listagem 3.

Listagem 3: Aplicação da Camada Física

```
import numpy as np
import numpy.typing as npt
import threading

# ...

class Programa(Gtk.Window):
    # ...
    def run(self,
            message,
            digital_mod,
            analog_mod,
            framing_method,
            error_method,
            errorcurve_mean,
            errorcurve_variance,
            parity_status) -> bool:
        # ...

        # Codificar por portadora
        amsg: npt.NDArray = np.empty(0)
        if analog_mod == GUI_ASK:
            amsg = cf.ASK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
    amp=GUI_AMP)
        elif analog_mod == GUI_FSK:
            amsg = cf.FSK_modulation(dmsg, (GUI_F1, GUI_F2), smplcnt=
    GUI_SMPLCNT, amp=GUI_AMP)
        elif analog_mod == GUI_BPSK:
            amsg = cf.BPSK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
```

```
    amp=GUI_AMP)
28          elif analog_mod == GUI_QPSK:
29              amsg = cf.QPSK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
    amp=GUI_AMP)
30          elif analog_mod == GUI_16QAM:
31              amsg = cf.QAM_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
    amp=GUI_AMP)
32          else:
33              return False
34          if amsg is None:
35              return False
36
37          # ...
38
39          # Adicionar ruido
40          amsg = utils.samples_addnoise(amsg, average=errorcurve_mean, spread
    =errorcurve_variance)
41          if amsg is None:
42              return False
43
44          # ...
45
46          # Enviar mensagem
47          if not tm.send_to_server(amsg):
48              return False
49
50          # Receber mensagem
51          receiver.flag_ready.wait(timeout=5)
52          rmsg: npt.NDArray = receiver.interpreted_data
53          if rmsg is None:
54              return False
55
56          # ...
57
58          # Decodificar mensagem
59          if analog_mod == GUI_ASK:
60              rmsg = cf.ASK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
     amp=GUI_AMP)
61          elif analog_mod == GUI_FSK:
62              rmsg = cf.FSK_demodulation(rmsg, (GUI_F1, GUI_F2), smplcnt=
    GUI_SMPLCNT, amp=GUI_AMP)
63          elif analog_mod == GUI_BPSK:
64              rmsg = cf.BPSK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1
    , amp=GUI_AMP)
65          elif analog_mod == GUI_QPSK:
66              rmsg = cf.QPSK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1
    , amp=GUI_AMP)
67          elif analog_mod == GUI_16QAM:
68              rmsg = cf.QAM_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
     amp=GUI_AMP)
69          else:
70              return False
71          if rmsg is None:
72              return False
73
74  # ...
```

2.2 Camada de Enlace

A camada de enlace foi estruturada logicamente em dois componentes: o transmissor e o receptor. O transmissor recebe um trem de bits representado por uma lista de valores booleanos (`False` para 0 e `True` para 1), contendo a mensagem original a ser enviada. A etapa inicial do processamento consiste em segmentar o fluxo de bits em blocos de carga útil, tarefa realizada pela função `split_bitstream_into_payloads`. Como resultado, obtém-se uma lista de listas de booleanos, em que cada lista interna representa um bloco. Cada bloco possui, por padrão, no máximo 32 bits, valor que pode ser ajustado pelo usuário na interface gráfica. O último bloco pode conter menos de 32 bits, porém seu tamanho deve obrigatoriamente ser par, um requisito necessário para a aplicação do *checksum*.

Para garantir essa condição, caso o último bloco tenha um número ímpar de bits, a função `add_padding_and_padding_size` insere um bit 0 ao final do bloco. Depois, ela acrescenta um bloco adicional de 2 bits que indica se houve padding: “00” quando nenhum bit foi inserido e “01” quando um bit 0 foi adicionado ao pacote anterior.

Em seguida, aplica-se um protocolo de detecção de erros (EDC) por meio da função `add_EDC`. O método específico é definido pelo usuário na interface gráfica e pode ser: bit de paridade, *checksum* ou CRC-32. A função `add_EDC` opera de maneira iterativa: para cada bloco de carga útil, ela invoca uma função correspondente ao protocolo selecionado, aplicando-o individualmente a cada bloco. Esse processo é repetido sequencialmente até que todos os blocos tenham sido processados e estejam devidamente protegidos pelo código de detecção de erros.

A título de exemplo, segue, na Listagem 4, a definição da função que adiciona o protocolo CRC-32 a um bloco de carga útil. O processo inicia com o acréscimo de 32 bits zero ao payload original. Utiliza o polinômio gerador `0x104C11DB7` (padrão IEEE 802.3). O algoritmo executa então uma divisão polinomial bit a bit: percorre todo o payload estendido, realizando operações `XOR` com o polinômio gerador quando o bit mais significativo (MSB) do resto atual é `1`, e efetua deslocamentos sucessivos.

Listagem 4: Implementação do protocolo CRC-32

```
def crc32_insert(payload: List[bool]) -> List[bool]:

    # Adiciona 32 bits 0 ao padding
    padding: List[bool] = [False] * 32
    dividend: List[bool] = payload + padding

    # Converte polinomio gerador em lista de booleanos
    divisor: List[bool] = int_to_bool_list(num=GENERATOR_POLYNOMIAL, size
    =33)

    # Inicia divisor com primeiros 33 bits do dividendo
    remainder: List[bool] = dividend[:33]

    # Divisao polinomial
    for i in range(len(divisor), len(dividend)):
        if remainder[0]:
            remainder = [a ^ b for a, b in zip(remainder, divisor)]
        remainder = remainder[1:] + [dividend[i]]

    if remainder[0]:
        remainder = [a ^ b for a, b in zip(remainder, divisor)]
    remainder = remainder[1:]  # Tira overflow do resto

    crc_bits: List[bool] = remainder
```

```
24
25      payload.extend(crc_bits)
26
27      return payload
```

Após a inserção dos códigos de detecção de erros (EDC), aplica-se a camada de correção através da função `add_ECC`. Esta função invoca iterativamente a função `hamming_insert` para cada bloco de dados. A função `hamming_insert`, cuja definição encontra-se na Listagem 5, implementa o protocolo Hamming para correção de erros, seguindo quatro etapas principais:

1. **Calcula o número de bits de paridade** necessários com base no tamanho dos dados, utilizando a relação $2^r \geq m + r + 1$, onde m é o número de bits de dados e r o número de bits de paridade.

2. **Insere *placeholders* para os bits de paridade** nas posições correspondentes a potências de 2 (1, 2, 4, 8, ...). Estes *placeholders* são inicializados com o valor `False` (bit 0). Esta inicialização é crucial porque:

   - O bit `False` (0) atua como **elemento neutro** da operação `XOR`
   - Permite que os bits de dados sejam processados sem interferência

3. **Calcula cada bit de paridade** usando operações `XOR`. É fundamental compreender que:

   - A operação `XOR` (^) implementa a **soma binária módulo 2**
   - Esta operação equivale à **paridade par**: resultado é 1 se houver número ímpar de 1's, 0 se houver número par de 1's
   - Cada bit de paridade cobre um padrão específico de posições:
     - **P1 (posição 1)**: bits 1, 3, 5, 7, 9, 11, ...
     - **P2 (posição 2)**: bits 2, 3, 6, 7, 10, 11, 14, 15, ...
     - **P4 (posição 4)**: bits 4, 5, 6, 7, 12, 13, 14, 15, ...
     - **P8 (posição 8)**: bits 8, 9, 10, 11, 12, 13, 14, 15, 24, ...

4. **Substitui os *placeholders*** iniciais pelos valores de paridade calculados, completando a palavra-código.

O resultado final é uma palavra-código de Hamming completa, onde cada bit de paridade protege um subconjunto específico de bits, permitindo detectar e corrigir erros individuais na transmissão.

Listagem 5: Implementação do código de Hamming para correção de erros

```
1  def hamming_insert(data_bits: List[bool]) -> List[bool]:
2
3      data_size: int = len(data_bits)
4      parity_bits_count: int = floor(log2(data_size)) + 1
5      # Ajusta numero de bits de paridade ate encontrar
6      # a condicao 2^r >= m + r + 1
7      while log2(data_size + parity_bits_count) > parity_bits_count:
8          parity_bits_count += 1
9
10     # Insere placeholder nas posicoes 1, 2, 4, 8, ...
11     position: int = 1
12     for _ in range(parity_bits_count):
```

```
        data_bits.insert(position - 1, False)
        position *= 2

    # Calcula bits de paridade
    step: int = 1
    for _ in range(parity_bits_count):
        parity: bool = False

        for j in range(step, len(data_bits) + 1, step * 2):
            for bit in data_bits[j - 1 : j - 1 + step]:
                parity ^= bit

        # Armazena bits de paridade
        data_bits[step - 1] = parity

        # Move para proxima posicao (2, 4, 8, ...)
        step *= 2

    return data_bits
```

A função `add_framing_protocol` aplica o método de enquadramento selecionado, suportando contagem de caracteres, *flags* com inserção de bits e de bytes. Para protocolos baseados em bytes, a função `apply_8bit_padding` garante o alinhamento a bytes através da adição de bits de preenchimento e um campo de 8 bits indicando sua quantidade. A Listagem 6 apresenta as implementações das funções de enquadramento por contagem de caracteres e alinhamento.

Listagem 6: Funções de enquadramento por contagem de caracteres e alinhamento de dados

```
def apply_8bit_padding(frame_body: List[bool]) -> List[bool]:

    # Se len(frame_body) ja for multiplo de 8
    # queremos que padding_size = 0 no lugar de 8
    # Por isso, adicionamos um modulo extra
    padding_size:int = (8 - len(frame_body) % 8) % 8

    frame_body += [False] * padding_size

    # Codifica padding size como uma lista de 8 booleanos
    padding_bits: List[bool] = int_to_bool_list(padding_size, 8)

    frame_body += padding_bits

    return frame_body

def add_char_count_flag_to_frame(frame_body: List[bool]) -> List[bool]:

    # Aplica padding
    frame_body = apply_8bit_padding(frame_body)

    # Calcula numero de bytes
    num_of_bytes: int = len(frame_body) // 8

    # Codifica numero de bytes como um cabecalho
    header: List[bool] = int_to_bool_list(num_of_bytes, 8)

    frame_body = header + frame_body

    return frame_body
```

Por fim, a função `add_padding_for_4bit_alignment` assegura que o número total de bits seja múltiplo de quatro antes que os quadros sejam enviados à camada física. Esse alinhamento é necessário para permitir a correta utilização de protocolos de modulação como QPSK e 16-QAM, cujos símbolos representam múltiplos fixos de 2 ou 4 bits. A função adiciona também um bloco final de quatro bits que codifica a quantidade de preenchimento aplicada.

No receptor, o processo inverso é realizado. O trem de bits recebido pode conter erros devido ao ruído inserido na camada física. A primeira etapa, realizada pela função `remove_padding_for_4bit_alignment`, remove os bits que foram utilizados para ajustar o tamanho da mensagem a um múltiplo de quatro.Na segunda, `remove_framing_protocol` retira os bits relacionados ao enquadramento e devolve a lista de pacotes correspondentes. Nessa etapa, também é removido o preenchimento inserido nos métodos de enquadramento baseados em bytes.

Posteriormente, a função `ECC_fix_corrupted_bits` aplica o código de Hamming para corrigir erros de um único bit presentes nos pacotes, e a função `remove_ECC` descarta os bits de redundância utilizados pelo ECC. A função `find_corrupted_frames` percorre os pacotes e gera uma lista indicando quais deles foram identificados como corrompidos pelos protocolos de detecção de erros.

Em seguida, a função `remove_EDC` remove os bits de detecção de erros dos pacotes já corrigidos. Por fim, a função `remove_paddings` verifica no último pacote quantos bits de preenchimento haviam sido adicionados no transmissor. Caso o último pacote esteja corrompido, ele é simplesmente descartado; caso contrário, a função descarta o último pacote e utiliza o valor armazenado nesse último pacote para remover os bits excedentes do penúltimo pacote. O resultado final é o trem de bits que representa a mensagem recebida pelo destinatário

2.3 Interface de Usuário

A Interface de Usuário consiste em uma janela principal a qual permite a inserção de todos os parâmetros desejados de uma simulação (presentes no arquivo `InterfaceGrafica.py`). Seja a escolha de modulação, protocolo de enlace, códigos de erro, quantidade de samples da onda, amplitude, frequência, etc.. Todos os blocos estão devidamente descritos. Para realizar a simulação, basta clicar no botão "Executar simulação", e janelas extras — cada qual com um gráfico relevante — são abertas para análise da simulação.

Por dentro do código, a janela principal consiste apenas de uma janela simples do GTK com um único objeto, uma grade. Outros campos de entrada são, então, organizados na grade, a fim de facilitar a organização e visualização para o usuário. Os outros campos consistem de texto, entrade de texto, dropdown e checkboxes. Segue um exemplo de construção da grade e do dropdown de escolhas para codificação de portadora na Listagem 7

Listagem 7: Construção do simulador

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

# ...

class Programa(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self, title="Simulador de Camadas Física e Enlace")
        # self.set_default_size(1600, 900)

        # Layout basico
```

```
        self.grid = Gtk.Grid()
        self.grid.set_column_spacing(15)
        self.grid.set_row_spacing(10)
        self.grid.set_border_width(10)
        self.add(self.grid)

        # Entradas de input simples
        self.input_section()
        self.digmod_section()
        self.algmod_section()
        self.framing_section()
        self.errordetct_section()
        self.errorcurve_section()
        self.extraparam_smpls()

        # Botao Run()
        self.run_button = Gtk.Button(label="Executar Simula ão")
        self.run_button.connect("clicked", self.on_run_clicked)
        self.grid.attach(self.run_button, 9, 0, 2, 2)

    # ...

    def algmod_section(self):
        # Analog modulation options
        analog_label = Gtk.Label(label="Modula ão por Portadora ->")
        self.grid.attach(analog_label, 3, 1, 1, 1)

        self.algmod_option = Gtk.ComboBoxText()
        self.algmod_option.append_text(GUI_ASK)
        self.algmod_option.append_text(GUI_FSK)
        self.algmod_option.append_text(GUI_BPSK)
        self.algmod_option.append_text(GUI_QPSK)
        self.algmod_option.append_text(GUI_16QAM)
        self.algmod_option.set_active(0)
        self.grid.attach(self.algmod_option, 4, 1, 2, 1)

# ...
```

As janelas dos gráficos são ainda mais simples. Usando do `FigureCanvas` disponibilizado pelo backend GTK do matplotlib, as figuras são colocadas em janelas individuais, para permitir a movimentação e comparação entre gráficos. Tanto a versão digital como analógica da visualização de gráficos, como também o display de texto, está implementada na Listagem 8.

Listagem 8: Output de texto e gráficos

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as
    FigureCanvas

# ...

class Programa(Gtk.Window):

    # ...

    def analog_plot(self, signal, title):
```

```
        mod = self.algmod_option.get_active_text()
        graph = plt.Figure()
        ax = graph.add_subplot()
        ax.plot(signal)
        ax.set_title(f"Sinal Analógico")
        ax.set_xlabel("Amostras")
        ax.set_ylabel("Leitura de tensão")

        canvas = FigureCanvas(graph)
        canvas.set_size_request(800, 400)
        win = Gtk.Window(title=f"Analógico - {title}")
        win.add(canvas)
        win.show_all()

    def digital_plot(self, dsignal, title):
        graph = plt.Figure()
        ax = graph.add_subplot()

        mod = self.digmod_option.get_active_text()

        if title == GUI_MAN:
            amnt_smpl = len(dsignal) // 2
        else:
            amnt_smpl = len(dsignal)

        x = np.linspace(0, amnt_smpl, amnt_smpl, endpoint=False)

        ax.plot(x, dsignal, drawstyle="steps-pre")
        ax.set_title(f"Sinal Digital")
        ax.set_xlabel("Amostras")
        ax.set_ylabel("Leitura de tensão")
        ax.set_xticks(np.arange(0, amnt_smpl + 1))
        ax.set_xlim(0, amnt_smpl)

        canvas = FigureCanvas(graph)
        canvas.set_size_request(800, 400)
        win = Gtk.Window(title=f"Digital - {title}")
        win.add(canvas)
        win.show_all()

    def text_show(self, message):
        # Display de texto
        label = Gtk.Label(label=message)
        scrlb = Gtk.ScrolledWindow()
        scrlb.add(label)
        scrlb.set_size_request(600, 600)
        win = Gtk.Window(title=f"Output de texto")
        win.add(scrlb)
        win.show_all()

# ...
```

Por fim, o funcionamento interno consiste do uso de sinais, funcionalidade própria do GTK. Um sinal é enviado ao requisitar a execução da simulação, e então os valores providenciados são verificados e, por fim, a simulação é realizada.

3 Membros

3.1 Henrique França

As contribuições realizadas iniciam-se pela camada física, com a implementação das modulações digitais NRZ-Polar, Manchester e Bipolar. Foi desenvolvida tanto a modulação quanto a desmodulação de cada técnica, assegurando que os sinais digitais fossem representados corretamente.

Na camada de enlace, foram implementados os métodos de enquadramento por Contagem de Caracteres e por FLAGS com inserção de bytes. Esses protocolos foram integrados de forma consistente ao restante da estrutura do simulador.

3.2 José Antônio de Andrade

Suas contribuições consistem, principalmente, da implementação da interface gráfica. Realizou todo o sistema de funcionamento e interação com o usuário, desde cheques relacionados à atribuição de valores até a projeção de gráficos na tela. Também, neste escopo, construiu o sistema de conversa via sockets presente no trabalho.

Adicionalmente, realizou os protocolos de (de)codificação de portadora ASK, FSK e BPSK, e ajudou na produção deste relatório.

3.3 Rafael Sapienza

Na camada física, implementou os esquemas de modulação QPSK e 16-QAM. Também implementou os algoritmos de demodulação desses protocolos.

No desenvolvimento da camada de enlace, desenvolveu uma função que recebe o trem bruto de bits e o segmenta em payloads de 32 bits, adicionando um pacote de padding quando o último bloco não possui o tamanho necessário. Além disso, implementou os protocolos de detecção de erros de checksum e de CRC32, bem como o mecanismo de correção de erros (ECC) baseado no código de Hamming, incluindo as rotinas de cálculo e inserção dos bits de redundância. Para o enquadramento, implementou o método baseado em flags com inserção de bits, abrangendo as funções de enquadramento e de desenquadramento.

4 Conclusão

A principal característica dessas simulações, mesmo com otimizações, é a demanda computacional excessiva quando implementada por software. Devidamente, esses protocolos foram feitos para serem realizados com circuitos físicos, principalmente em relação à Camada Física. Mesmo assim, o programa pode ser útil para a análise de sistemas sob influência de ruído constante, para testar os limites e configurações necessárias para a transmissão de mensagens.

A Camada Física não apresentou dificuldades em sua implementação, mas a Camada de Enlace forneceu diversos problemas durante o desenvolvimento. Foi necessário testar diversos casos extremos para identificar problemas que surgiram. Notavelmente, a aplicação de padding e sua devida remoção foram campos de tediosa implementação. Por sua vez, a Interface de Usuário foi outra área problemática, principalmente para introduzir os gráficos da biblioteca matplotlib de forma correta.

De todo modo, o trabalho providenciou uma visão mais ampla dos diversos protocolos e algoritmos por trás da comunicação comum do dia a dia, não só como são realizados e produzidos, mas também das diversas dificuldades presentes no estabelecimento de um protocolo tanto eficiente quanto prático.