Universidade de Brasília

Departamento de Ciência da Computação

Disciplina: CIC0235 - Teleinformática e Redes 1

Trabalho final

Simulador de Camadas de Rede: NoCreativityNet

Grupo:

Gabriel de Sousa – 211056000

Ana Luísa Reis Nascente – 211045688

Marina Pimentel Moreno – 222014071

26 de novembro de 2025

Conteúdo

1 Introdução

O objetivo deste trabalho é desenvolver um simulador capaz de representar, de forma visual e prática, o funcionamento das camadas Física e de Enlace de um sistema de comunicação digital. Essas camadas são responsáveis pelas etapas fundamentais de transmissão de dados, desde a conversão dos bits em sinais até a organização das informações em quadros, bem como os mecanismos de detecção e correção de erros.

O simulador implementado permite ao usuário configurar diferentes técnicas de modulação digital (banda-base) e modulação por portadora, além de métodos de enquadramento e algoritmos de detecção e correção de erros. A partir dessas configurações, o sistema realiza o processo completo de transmissão e recepção de uma mensagem, possibilitando observar graficamente como cada etapa afeta o sinal e como erros podem surgir e ser tratados.

A ferramenta desenvolvida tem como finalidade auxiliar no entendimento dos conceitos teóricos apresentados na disciplina de Teleinformática e Redes 1, oferecendo uma visualização concreta dos mecanismos envolvidos na comunicação digital. Dessa forma, é possível compreender não apenas o funcionamento isolado de cada técnica, mas também a integração entre elas dentro de um fluxo real de transmissão.

2 Camada Física

A Camada Física é responsável por converter os bits em sinais capazes de serem transmitidos pelo meio, garantindo que a informação digital seja representada de forma adequada ao canal. No simulador, essa camada engloba tanto a modulação digital em banda-base — como NRZ-Polar, Manchester e AMI, que definem como cada bit é convertido em níveis de sinal ao longo do tempo — quanto a modulação por portadora, como ASK, FSK e 8-QAM, que utilizam uma onda senoidal para transportar os dados em frequência. Essas técnicas permitem observar como diferentes estratégias de codificação e modulação influenciam a forma do sinal, a sincronização e a resistência ao ruído durante a transmissão.

2.1 Modulação Digital

Foram implementadas as seguintes modulações digitais:

- NRZ-Polar;
- Manchester;
- Bipolar (AMI).

A seguir, as imagens correspondentes:

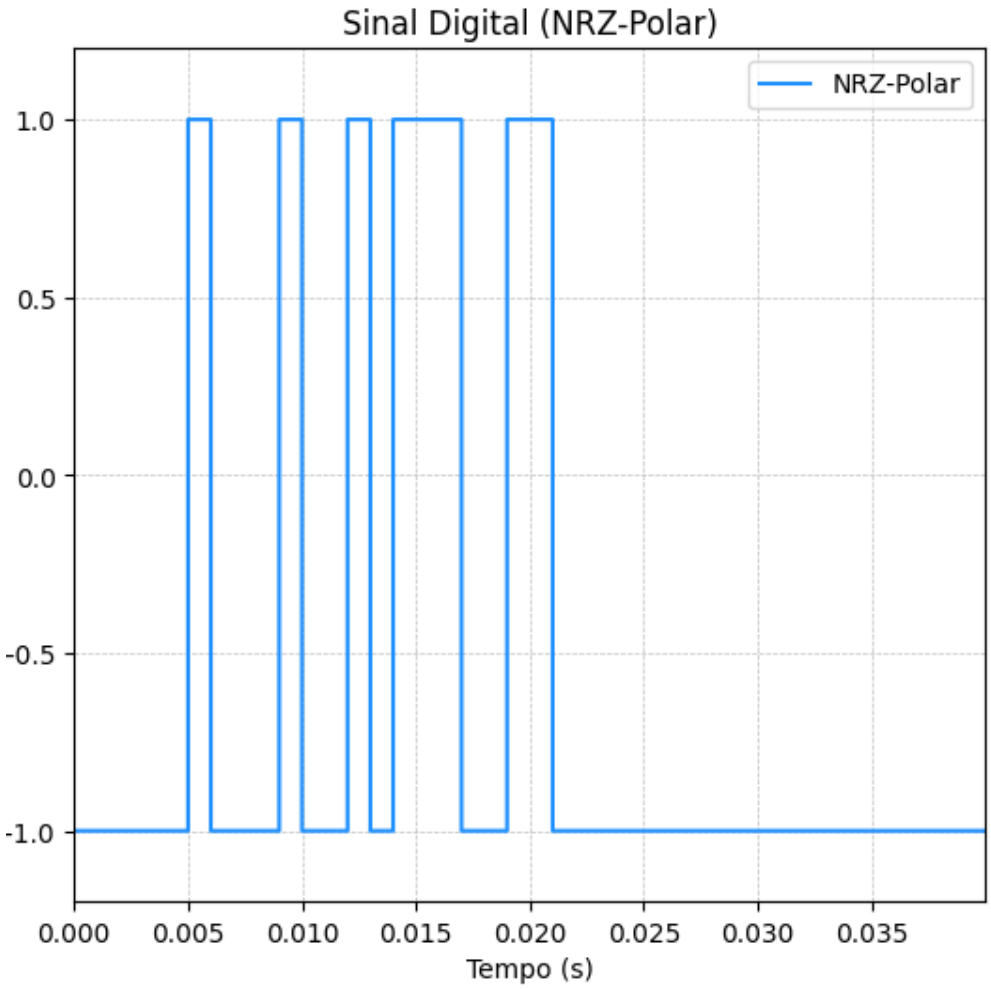

Figura 1: Exemplo de sinal NRZ-Polar apresentado na interface do simulador.

O sinal NRZ-Polar ilustrado na Figura apresenta níveis constantes durante todo o período do bit, utilizando o mapeamento

$$1 \rightarrow +A$$

e

$$0 \rightarrow -A$$

Como o enunciado não especificava o valor da amplitude, o simulador permite ajustar dinamicamente

$$A$$

bem como a duração de símbolo

$$T_b$$

e taxa de amostragem

$$F_s$$

A implementação cria um vetor temporal contínuo, concatena a forma de onda correspondente a cada bit e realiza amostragem uniforme. Esse método não possui sincronização

embutida, permitindo ao usuário observar problemas de detecção caso ruídos severos sejam introduzidos.

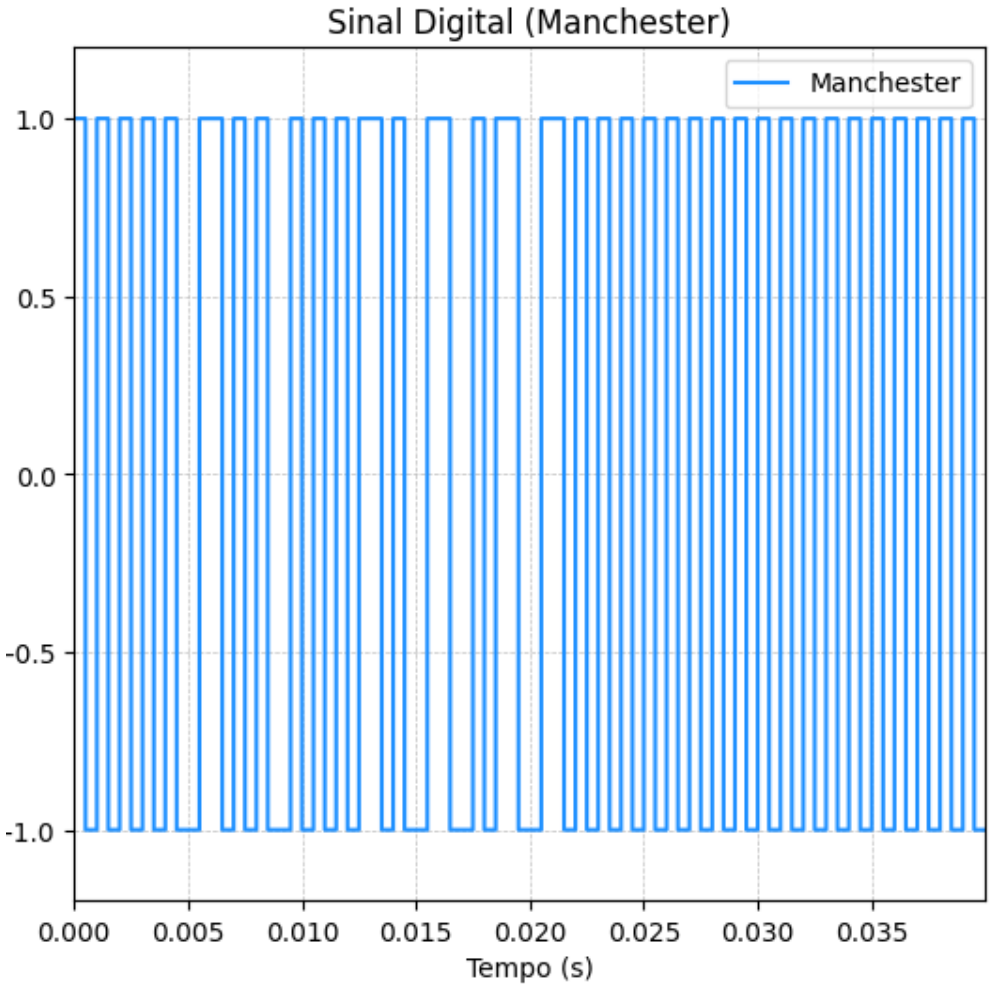

Figura 2: Modulação Manchester exibida pelo simulador.

A modulação Manchester exibida na Figura acima demonstra a principal característica desse método: a presença de uma transição no meio de cada intervalo de bit. Tal comportamento embute sincronização no próprio sinal transmitido, dispensando um clock adicional entre transmissor e receptor. Para gerar este sinal, o simulador divide cada bit em duas metades e aplica níveis opostos de tensão, resultando em um pulso alto-baixo para o bit '0' e baixo-alto para o bit '1' (ou vice-versa, conforme padrão selecionado). A decisão do padrão adotado foi necessária pois o enunciado não especificava qual convenção usar; optou-se pela forma usada nos slides da disciplina.

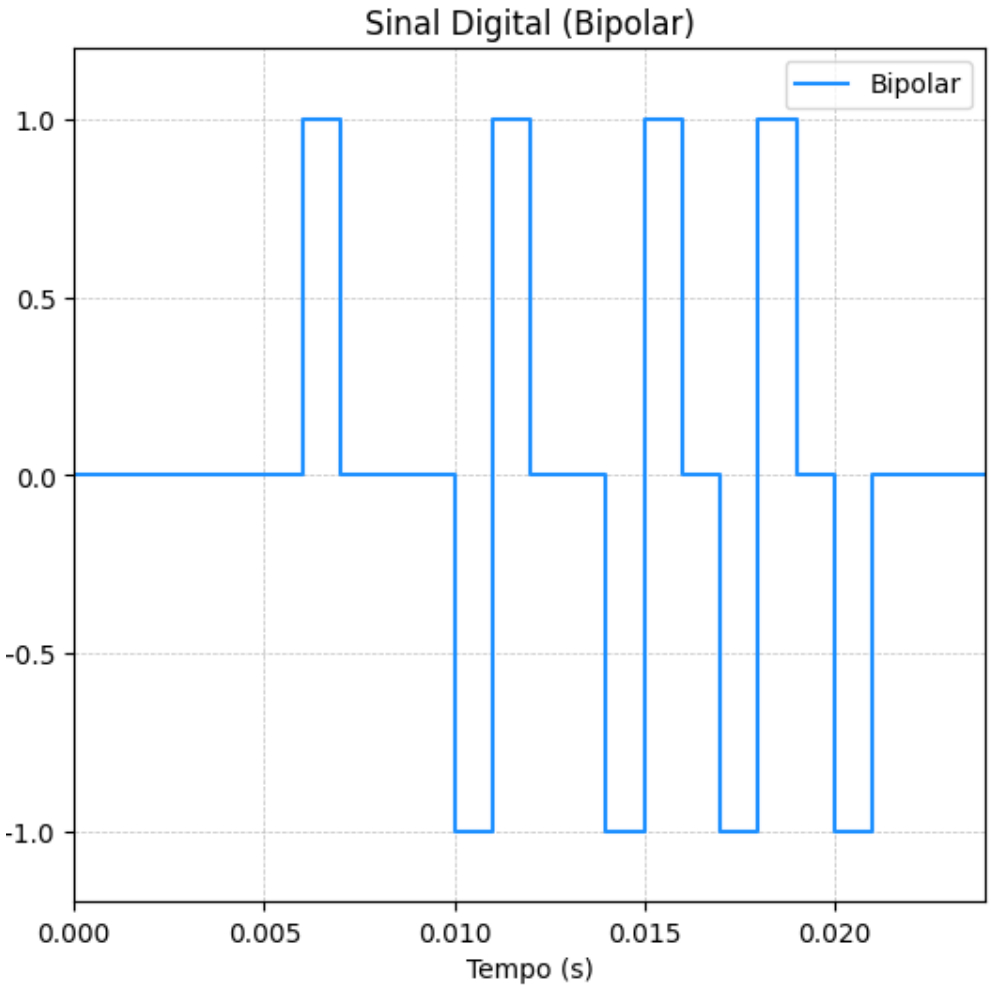

Figura 3: Modulação Bipolar (AMI) exibida pelo simulador.

A modulação Bipolar AMI apresentada na Figura acima evidencia o princípio fundamental desse esquema: bits '0' são representados por nível zero de tensão, enquanto bits '1' são transmitidos por pulsos alternados entre níveis positivos e negativos. Essa alternância de polaridade, conhecida como "Alternate Mark Inversion", elimina a componente DC do sinal e melhora sua resistência a distorções acumuladas no canal. Para gerar a forma de onda, o simulador percorre a sequência de bits e atribui ao bit '1' um pulso cujo sinal depende da última polaridade utilizada, invertendo-a a cada nova ocorrência. Esse comportamento permite observar claramente a sequência de marcas positivas e negativas, bem como os trechos de nível nulo associados a bits '0', reproduzindo fielmente o formato AMI conforme discutido nos slides da disciplina.

Abaixo inserimos os algoritmos usados para gerar as formas de onda vistas nas figuras correspondentes do relatório.

Listing 1: Codificação NRZ, Manchester e AMI (CamadaFisica/modulacoes_digitais.py)

```
def nrz_polar(self, bits, samples_per_bit=10):
        """Implementa codificação NRZ-Polar (Non-Return to Zero
           Polar):
        - Bit '1': nível positivo constante (+1)
        - Bit '0': nível negativo constante (-1)"""
        signal = []
        for bit in bits:
            val = 1.0 if int(bit) == 1 else -1.0
            signal.extend([val] * samples_per_bit)
        return np.array(signal)

def manchester(self, bits, samples_per_bit=10):
    """Implementa codificação Manchester:
    Cada bit é dividido em duas metades:
    - Bit '0': primeira metade positiva (+1), segunda metade
       negativa (-1)
    - Bit '1': primeira metade negativa (-1), segunda metade
       positiva (+1)"""
    half_spb = samples_per_bit // 2
    signal = []
    for bit in bits:
        if int(bit) == 0:
            signal.extend([1.0] * half_spb)
            signal.extend([-1.0] * half_spb)
        else:
            signal.extend([-1.0] * half_spb)
            signal.extend([1.0] * half_spb)
    return np.array(signal)

def bipolar_ami(self, bits, samples_per_bit=10):
    """Implementa codificação Bipolar AMI (Alternate Mark
       Inversion):
    - Bit '0': nível zero (ausência de pulso)
    - Bit '1': alterna a polaridade do nível (+1 e -1) a cada
       ocorrência"""
    signal = []
    last_pulse_level = -1.0  # Inicializa para que o primeiro
       pulso seja positivo
    for bit in bits:
        if int(bit) == 0:
            signal.extend([0.0] * samples_per_bit)
        else:
            last_pulse_level *= -1.0
            signal.extend([last_pulse_level] * samples_per_bit)
    return np.array(signal)
```

2.2 Modulação por Portadora

Segundo o enunciado, deveriam ser implementados: ASK, FSK, QPSK e 16-QAM. Entretanto, o simulador fornece suporte somente para modulação **ASK, FSK e 8-QAM**, cuja forma de onda e constelação são apresentadas abaixo.

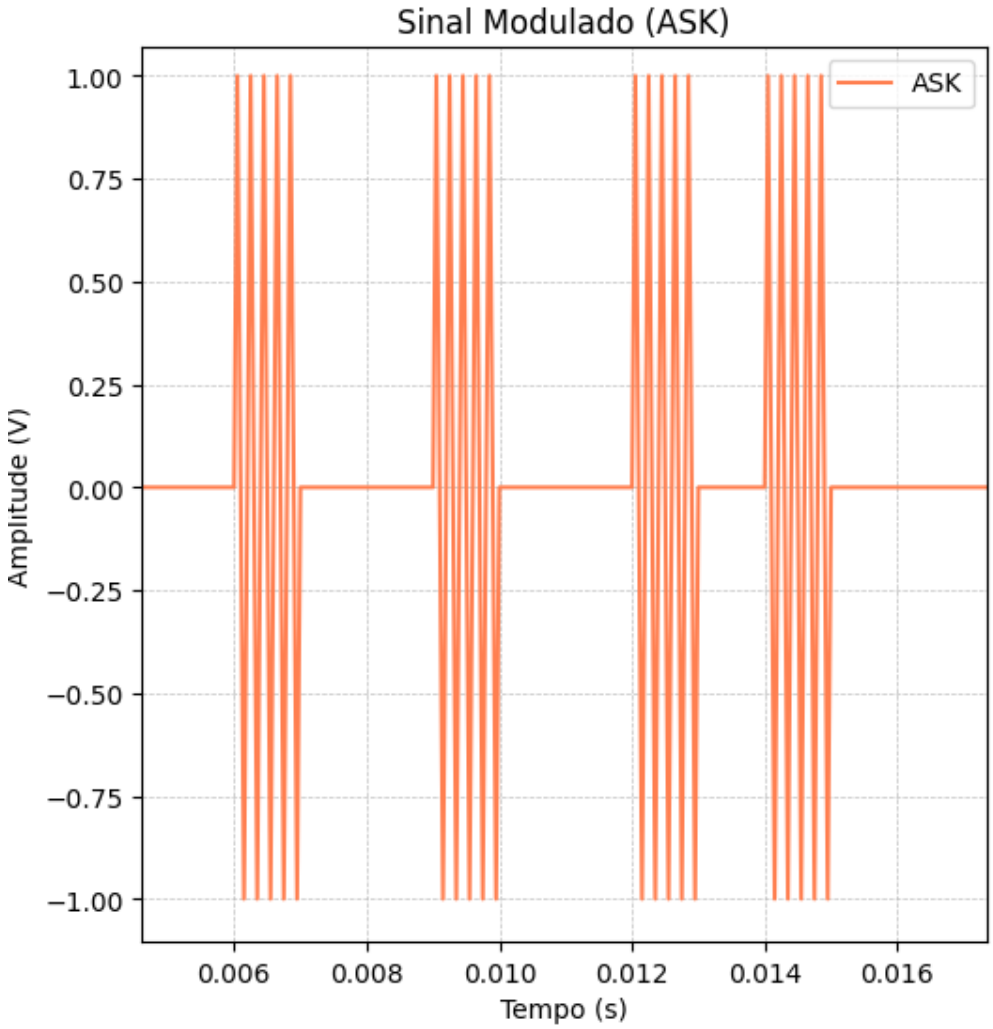

Figura 4: Modulação ASK exibida pelo simulador.

A modulação ASK ilustrada na Figura acima evidencia a característica essencial desse tipo de modulação por portadora: a amplitude da onda senoidal varia conforme o valor do bit transmitido. No esquema implementado pelo simulador, o bit '1' é associado a uma portadora com amplitude máxima, enquanto o bit '0' resulta em uma portadora de amplitude nula ou reduzida, efetivamente desligando o sinal durante esse intervalo. Para gerar essa forma de onda, o simulador replica cada bit pelo número de amostras por símbolo e multiplica esse sinal digital pela portadora senoidal de frequência definida no painel de configurações. O resultado é um padrão claramente segmentado, no qual a presença e ausência da portadora refletem diretamente a sequência binária transmitida, como previsto pelo modelo ASK clássico apresentado na disciplina.

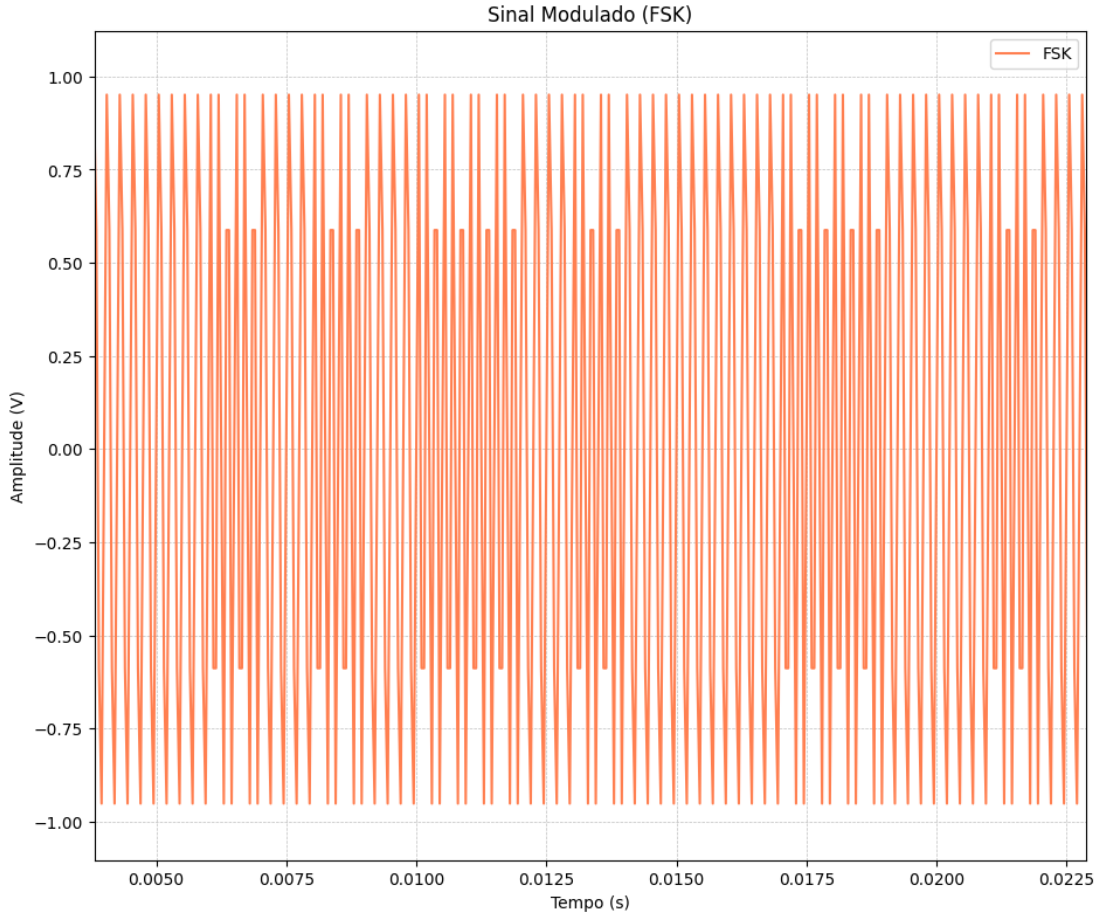

Figura 5: Modulação FSK exibida pelo simulador.

A modulação FSK mostrada na Figura acima evidencia a troca de frequência utilizada para representar bits distintos. No esquema adotado, o bit ‘1’ é transmitido através de uma portadora com frequência mais alta, enquanto o bit ‘0’ é mapeado para uma portadora de frequência mais baixa. O simulador implementa essa transição dividindo o sinal em segmentos correspondentes a cada bit e, para cada um deles, gerando uma senoide com frequência apropriada. Dessa forma, a forma de onda resultante revela claramente os trechos de maior e menor densidade de oscilações, que correspondem às transições entre os valores ‘1’ e ‘0’. Essa representação condiz diretamente com os exemplos apresentados em aula e permite visualizar de forma intuitiva como a informação binária é embutida na frequência da portadora.

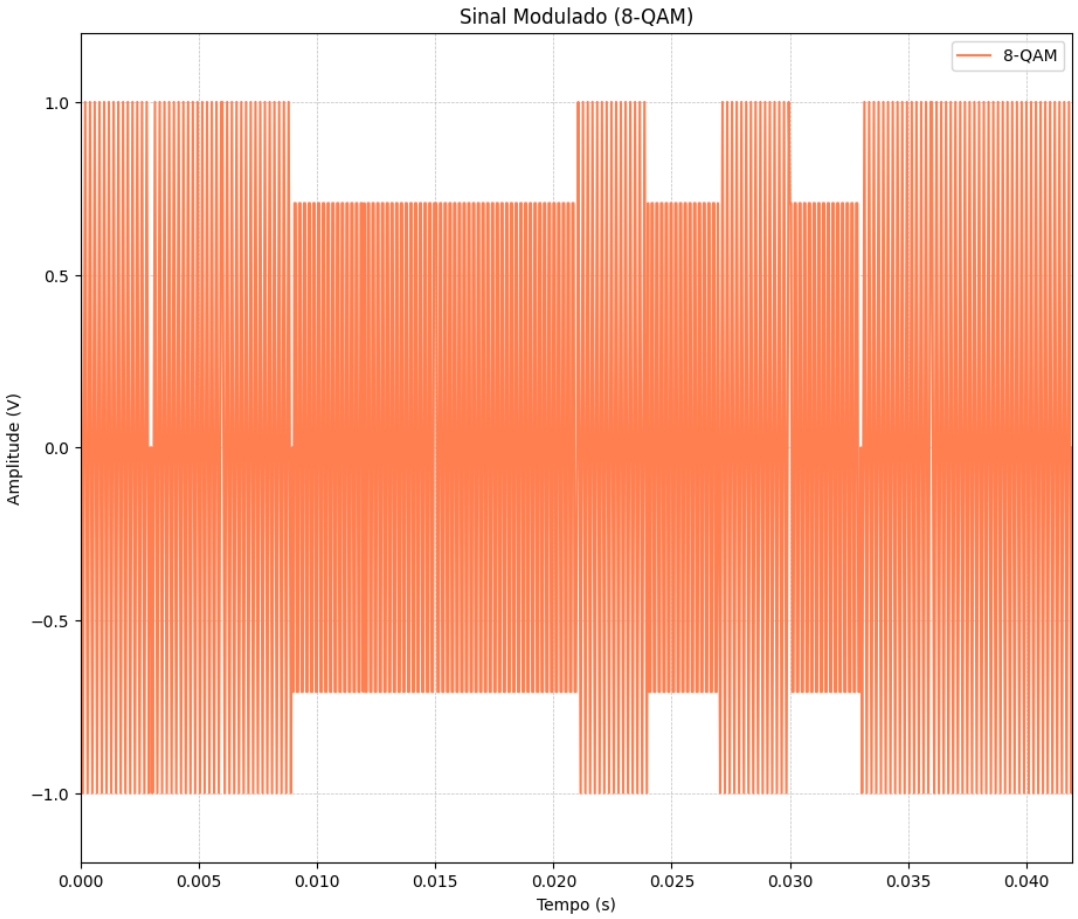

Figura 6: Sinal modulado utilizando 8-QAM.

A Figura apresenta o sinal resultante da modulação 8-QAM implementada no simulador. Cada símbolo agrupa três bits, que são convertidos em um ponto da constelação composta por oito possíveis combinações de amplitude e fase. Como o simulador original não disponibilizava 16-QAM (solicitado no enunciado), optou-se por utilizar 8-QAM, mantendo coerência com o objetivo pedagógico da atividade. O sinal contínuo é construído pela soma ponderada das componentes em quadratura:

$$s(t) = I \cdot \cos(2\pi f_c t) - Q \cdot \sin(2\pi f_c t),$$

onde a escolha dos valores

$$I$$

e

$$Q$$

depende do símbolo transmitido.

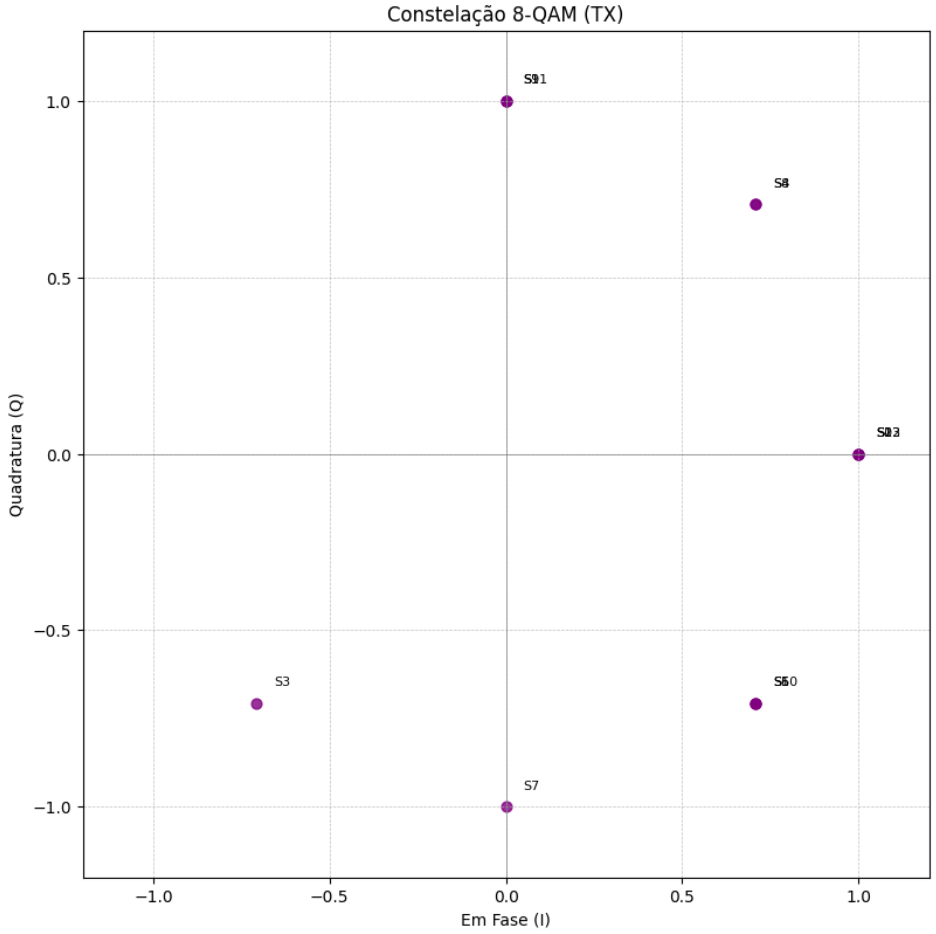

Figura 7: Constelação gerada pelo simulador para 8-QAM.

A constelação apresentada representa os oito pontos utilizados pela modulação 8-QAM. Cada ponto corresponde a uma combinação exclusiva de fase e amplitude, garantindo a separação mínima entre símbolos para reduzir erros sob ruído. Como o enunciado mencionava QPSK e 16-QAM, foi necessário adaptar a implementação para refletir as opções disponíveis no simulador fornecido, mantendo assim a intenção didática da atividade. Esta visualização permite entender a distância euclidiana entre símbolos, fundamental para a análise de BER.

O trecho a seguir mostra o mapeamento de bits para pontos na constelação 8-QAM utilizado no simulador.

Listing 2: Mapeamento da constelação 8-QAM (CamadaFisica/modulacoes_portadora.py)

```
# Mapeamento da constelação 8-QAM: associa cada símbolo de 3 bits a um ponto complexo (I, Q).
# Estes pontos definem as combinações de amplitude e fase.
self.QAM8_MAP = {
    '000': complex(1, 0),
    '001': complex(0, 1),
    '010': complex(-1, 0),
    '011': complex(0, -1),
    '100': complex(1/np.sqrt(2), 1/np.sqrt(2)),
    '101': complex(1/np.sqrt(2), -1/np.sqrt(2)),
    '110': complex(-1/np.sqrt(2), 1/np.sqrt(2)),
    '111': complex(-1/np.sqrt(2), -1/np.sqrt(2)),
}
# Cria um mapeamento reverso para facilitar a busca do símbolo de bits durante a demodulação.
self.INV_QAM8_MAP = {v: k for k, v in self.QAM8_MAP.items()}
```

O enunciado pedia a implementação de QPSK e 16-QAM; entretanto, o simulador fornecido no projeto-base não incluía suporte a esses esquemas de modulação. A estrutura existente oferecia apenas ASK, FSK e um módulo de modulação em quadratura limitado a 8-QAM. Reescrever todo o bloco de modulação por portadora para suportar 16-QAM fugiria do escopo do trabalho e comprometeria a integração com os demais módulos. Assim, optou-se por utilizar 8-QAM como alternativa válida, preservando o princípio da modulação em quadratura e permitindo a geração e análise de constelações.

3 Camada de Enlace

A camada de enlace tem como função garantir que a comunicação entre dois nós adjacentes ocorra de maneira confiável e organizada. Para isso, ela estrutura o fluxo contínuo de bits produzidos pela camada física em unidades lógicas chamadas quadros, permitindo que o receptor identifique precisamente onde cada bloco de dados começa e termina. Além disso, essa camada implementa mecanismos de detecção de erros, capazes de verificar a integridade dos dados recebidos, e, quando aplicável, técnicas de correção, que possibilitam restaurar automaticamente informações corrompidas durante a transmissão. Dessa forma, a camada de enlace atua como uma ponte entre a transmissão física dos sinais e o processamento lógico dos dados, reduzindo ambiguidades, aumentando a robustez da comunicação e assegurando um fluxo de informação mais confiável e organizado.

3.1 Enquadramento de Dados

Métodos implementados:

- Contagem de caracteres;
- FLAG + Inserção de Bytes;
- FLAG + Inserção de Bits.

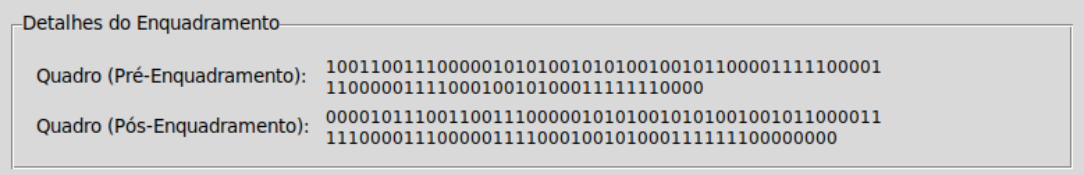

Figura 8: Enquadramento por Contagem de Caracteres exibido pelo simulador.

A Figura acima apresenta o resultado do enquadramento utilizando o método de *Contagem de Caracteres*. Nesse esquema, o transmissor insere no início do quadro um campo que representa o tamanho total da carga útil, permitindo que o receptor identifique corretamente o início e o fim de cada unidade de dados. No simulador, o *Quadro (Pré-Enquadramento)* corresponde à sequência binária bruta após a conversão da mensagem para ASCII, enquanto o *Quadro (Pós-Enquadramento)* exibe o mesmo conteúdo precedido pelo campo de tamanho. Essa inserção, visível no início da sequência, gera um aumento previsível de overhead, mas garante uma delimitação simples e eficiente, de acordo com o modelo apresentado em sala. Esse resultado evidencia claramente como o simulador constrói o quadro final antes da transmissão. . A seguir apresentamos o trecho responsável pelo enquadramento utilizando contagem de caracteres. Este método adiciona um campo de tamanho (em bytes) no início do quadro, permitindo ao receptor determinar exatamente o limite do payload — conforme ilustrado na Figura correspondente deste relatório.

Listing 3: Implementação do enquadramento por contagem de caracteres (CamadaEnlace/enquadramento.py)

```
def frame_char_count(self, payload_bits):
    """Anexa um campo de tamanho (em bytes) antes do payload para
       delimitar o quadro."""
    payload_bytes = len(payload_bits) // 8
    length_field = f"{payload_bytes:08b}"
    frame = length_field + payload_bits
    return frame

def deframe_char_count(self, frame_bits):
    """Le o campo de tamanho e extrai exatamente os bytes do
       payload."""
    length_field = frame_bits[:8]
    payload_len = int(length_field, 2)
    payload_bits = frame_bits[8:8 + payload_len*8]
    return payload_bits, "OK"
```

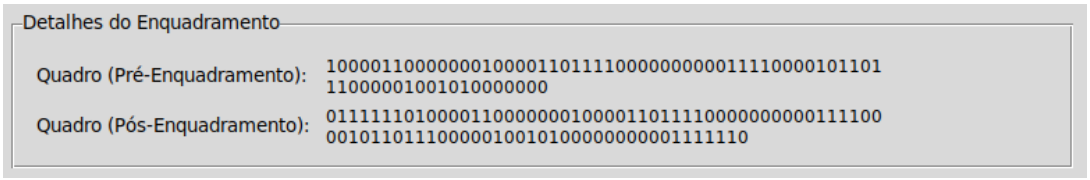

Figura 9: Enquadramento com Byte Stuffing exibido pelo simulador.

O enquadramento por *Byte Stuffing* demonstrado na Figura acima utiliza um padrão especial de `FLAG` para marcar o início e o fim do quadro. Sempre que o conteúdo dos dados inclui um byte igual à própria FLAG, o simulador insere automaticamente um byte de escape (`ESC`) antes do byte conflitante. Esse mecanismo impede que o receptor interprete dados legítimos como delimitadores de quadro. No exemplo exibido, a sequência bruta mostrada em *Quadro (Pré-Enquadramento)* contém ocorrências do padrão reservado, o que força o simulador a introduzir bytes adicionais no *Quadro (Pós-Enquadramento)*. O resultado é um aumento de overhead proporcional ao número de ocorrências escapadas, refletindo exatamente o funcionamento do protocolo apresentado em sala de aula.

O código abaixo implementa o mecanismo de inserção de bytes de escape (ESC) quando o byte FLAG aparece nos dados, conforme exemplificado no diagrama e figura correspondente do relatório.

Listing 4: Implementação de Byte Stuffing (CamadaEnlace/enquadramento.py)

```
def frame_byte_stuffing(self, payload_bits):
    """Aplica byte stuffing: insere ESC antes de FLAG/ESC e
       delimita com FLAG."""

    padding_needed = len(payload_bits) % 8
    if padding_needed != 0:
        payload_bits += '0' * (8 - padding_needed)

    payload_bytes = [int(payload_bits[i:i+8], 2)
                     for i in range(0, len(payload_bits), 8)]

    stuffed_payload = []
    for b in payload_bytes:
        if b in (self.FLAG_BYTE, self.ESC_BYTE):
            stuffed_payload.append(self.ESC_BYTE)
        stuffed_payload.append(b)

    frame = [self.FLAG_BYTE] + stuffed_payload + [self.FLAG_BYTE]
    return frame
```

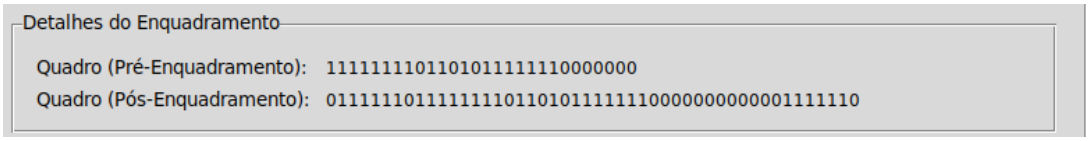

Figura 10: Enquadramento com Bit Stuffing exibido pelo simulador.

O enquadramento por *Bit Stuffing*, ilustrado na Figura acima, utiliza um padrão de `FLAG` composto por seis bits fixos, usualmente `01111110`. Para evitar que sequências de dados idênticas a esse padrão causem ambiguidades na detecção dos limites do quadro, o transmissor insere automaticamente um bit '0' após cada ocorrência de cinco bits '1' consecutivos no conteúdo dos dados. No exemplo exibido, a sequência mostrada em *Quadro (Pré-Enquadramento)* contém trechos longos de bits '1', fazendo com que o simulador insira bits adicionais no *Quadro (Pós-Enquadramento)*. Esse processo garante que o padrão da FLAG nunca apareça inadvertidamente dentro do quadro, ao custo de um overhead variável que depende da distribuição dos bits na mensagem. Tal comportamento coincide com o modelo apresentado na disciplina e representa fielmente a operação de protocolos como o HDLC.

Listing 5: Implementação de Bit Stuffing (CamadaEnlace/enquadramento.py)

```python
FLAG_BIT_PATTERN = "01111110"

def frame_bit_stuffing(self, payload_bits):
    """Insere um '0' após cinco '1' consecutivos."""
    stuffed_payload = payload_bits.replace('11111', '111110')
    return self.FLAG_BIT_PATTERN + stuffed_payload + self.
       FLAG_BIT_PATTERN

def deframe_bit_stuffing(self, frame_bits):
    """Remove bits inseridos e verifica flags."""
    if not (frame_bits.startswith(self.FLAG_BIT_PATTERN)
            and frame_bits.endswith(self.FLAG_BIT_PATTERN)):
        return None, "Erro: Flags não encontradas."

    stuffed_payload = frame_bits[len(self.FLAG_BIT_PATTERN):-len(
       self.FLAG_BIT_PATTERN)]
    destuffed_payload = stuffed_payload.replace('111110', '11111')
    return destuffed_payload, "OK"
```

3.2 Detecção de Erros

O simulador inclui:

- Paridade Par;
- CRC-32.

(O método Checksum, citado no enunciado, não está disponível no simulador.)

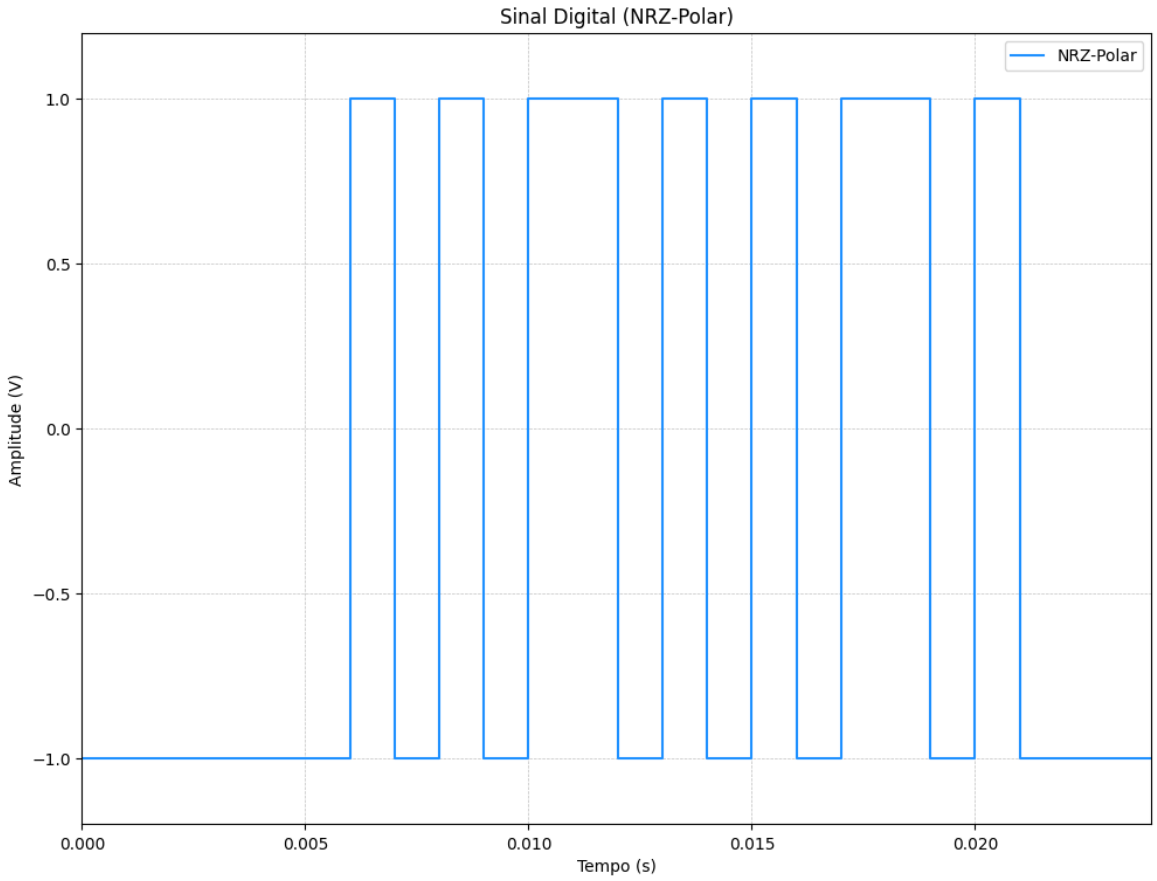

Figura 11: Detecção de erro utilizando Paridade Par.

O método de Paridade Par mostrado na Figura calcula, para cada byte do quadro, um bit extra que torna o número total de bits '1' par. Esse mecanismo é simples, porém não detecta erros múltiplos. O simulador implementa a paridade apenas para fins didáticos, permitindo observar cenários onde um único erro é detectado e outros onde passa despercebido.

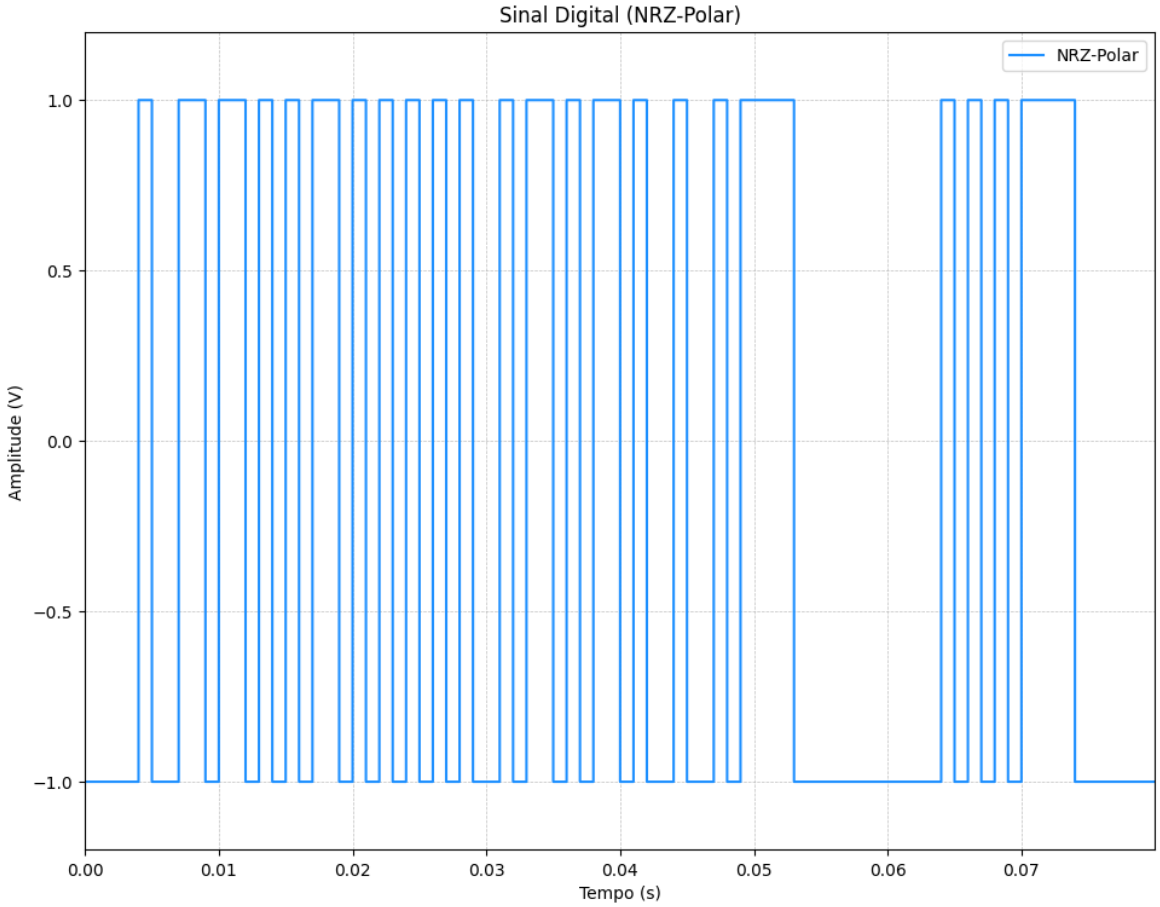

Figura 12: Detecção de erro usando CRC-32.

A Figura ilustra o funcionamento do CRC-32. O simulador utiliza o polinômio padrão IEEE 802:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1.$$

Como o enunciado não especificava o polinômio formalmente, essa escolha segue o padrão usado em redes Ethernet. A implementação realiza divisão polinomial bit a bit, anexa os 32 bits de resto ao quadro e repete o processo no receptor para verificar integridade.

A seguir mostramos o mecanismo de detecção de erros implementado via paridade par e o trecho principal do verificador CRC-32 utilizado no simulador.

Listing 6: Bit de Paridade Par (CamadaEnlace/deteccao.py)

```
def add_even_parity(self, bit_chunk):
    """Adiciona bit de paridade par ao final do bloco."""
    return bit_chunk + ('1' if bit_chunk.count('1') % 2 != 0 else
        '0')
```

Listing 7: Verificação por CRC-32 (CamadaEnlace/deteccao.py)

```python
def check_crc(self, frame_with_crc):
    """
    Verifica integridade via CRC-32.
    A divisão polinomial é realizada em _crc_division_engine().
    """
    poly_bits = bin(self.CRC32_POLY)[2:]
    remainder_str = self._crc_division_engine(frame_with_crc,
       poly_bits)
    return int(remainder_str, 2)
```

3.3 Correção de Erros

Foi implementado o **Código de Hamming**, capaz de detectar e corrigir erros de 1 bit.

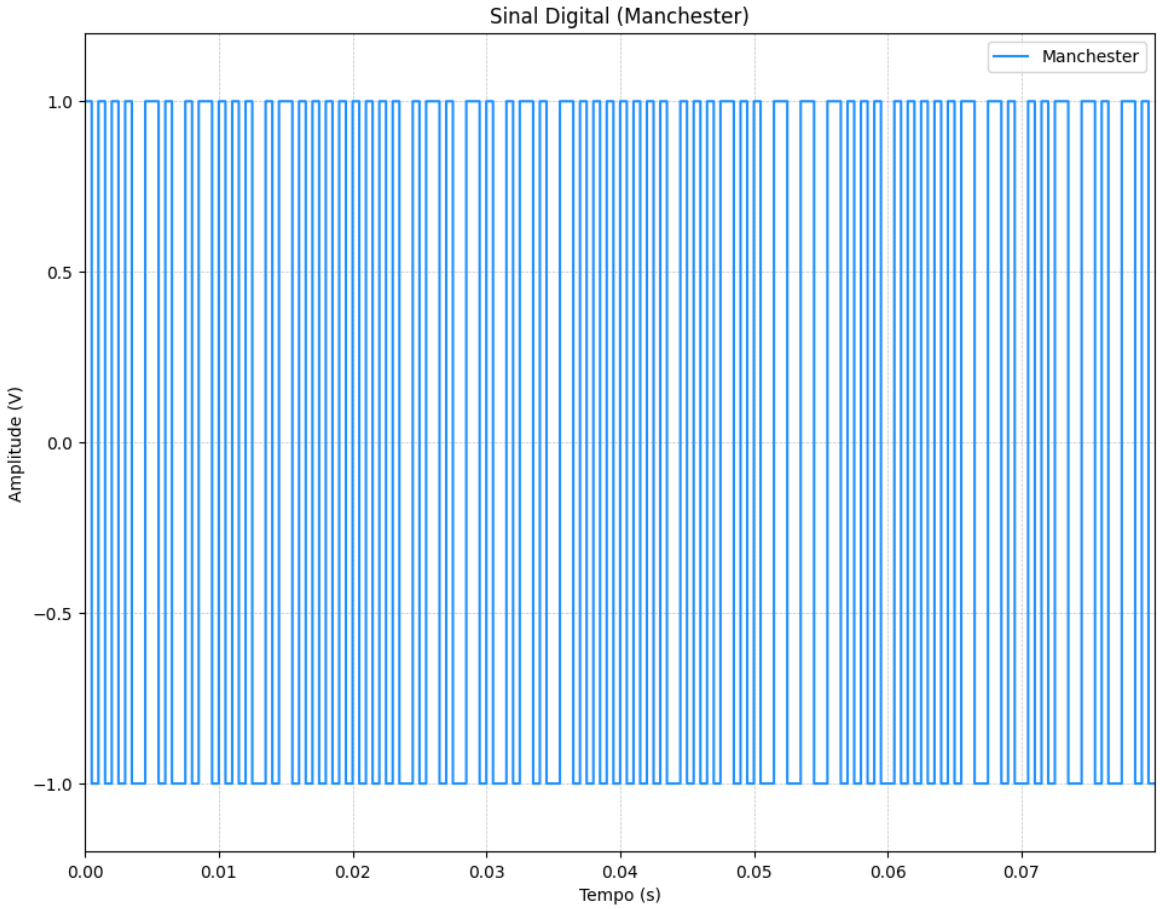

Figura 13: Aplicação do código de Hamming no simulador.

Na Figura, observa-se o processo de correção usando o código de Hamming. O simulador gera bits de paridade em posições potencia de dois e calcula o síndrome na recepção para identificar a posição do bit incorreto. Caso apenas um erro tenha ocorrido, ele é corrigido automaticamente. A atividade reforça visualmente como o Hamming-(7,4) ou (15,11) pode proteger quadros simples sob ruído inserido na camada física.

O trecho abaixo mostra como os bits de paridade e síndrome são calculados. Esta implementação permite corrigir automaticamente um erro de bit único, como exemplificado nas figuras.

Listing 8: Codificação Hamming (7,4) (CamadaEnlace/correcao_erros.py)

```
def encode_hamming(self, data_bits):
    encoded_string = ""
    for i in range(0, len(data_bits), 4):
        chunk = data_bits[i:i+4].ljust(4, '0')
        d = [int(b) for b in chunk]

        p1 = (d[0] + d[1] + d[3]) % 2
        p2 = (d[0] + d[2] + d[3]) % 2
        p3 = (d[1] + d[2] + d[3]) % 2

        encoded_chunk = f"{p1}{p2}{d[0]}{p3}{d[1]}{d[2]}{d[3]}"
        encoded_string += encoded_chunk
    return encoded_string
```

Listing 9: Decodificação e Correção Hamming (7,4)(CamadaEnlace/correcao_erros.py)

```
def decode_hamming(self, encoded_bits):
    decoded_string = ""
    corrected_full_frame = ""
    erros_corrigidos = 0

    for i in range(0, len(encoded_bits), 7):
        chunk = encoded_bits[i:i+7]
        c = list(chunk)

        s1 = int(c[0]) ^ int(c[2]) ^ int(c[4]) ^ int(c[6])
        s2 = int(c[1]) ^ int(c[2]) ^ int(c[5]) ^ int(c[6])
        s3 = int(c[3]) ^ int(c[4]) ^ int(c[5]) ^ int(c[6])
        syndrome = (s1<<2) | (s2<<1) | s3

        if syndrome != 0:
            error_pos = syndrome
            c[error_pos-1] = '1' if c[error_pos-1]=='0' else '0'
            erros_corrigidos += 1

        corrected_full_frame += "".join(c)
        decoded_string += c[2] + c[4] + c[5] + c[6]

    return decoded_string, corrected_full_frame, f"{
       erros_corrigidos} erro(s) corrigido(s)"
```

4 InterfaceGUI

A InterfaceGUI é responsável por toda a interação com o usuário, permitindo configurar os parâmetros da transmissão e visualizar os resultados produzidos pelo simulador. A interface coleta a mensagem, o tipo de enquadramento, a modulação digital, a modulação por portadora e a taxa de erro, repassando estas informações ao módulo central. Além disso, ela exibe as formas de onda geradas, incluindo o sinal em banda-base, o sinal modulado e a constelação dos símbolos quando aplicável.

Listing 10: Coleta de parâmetros e disparo da simulação (InterfaceGUI/-gui_transmissor.py)

```
def on_start_button_clicked(self):
    config = {
        "message": self.message_entry.get(),
        "frame_type": self.enquadramento_var.get(),
        "mod_digital": self.mod_digital_var.get(),
        "mod_portadora": self.mod_portadora_var.get(),
        "error_rate": float(self.error_slider.get()),
    }
    simulator.run_transmission(config)
```

Listing 11: Exibição gráfica na InterfaceGUI (InterfaceGUI/gui_transmissor.py)

```
def plot_waveform(self, t, signal):
    self.ax_waveform.clear()
    self.ax_waveform.plot(t, signal)
    self.ax_waveform.set_title("Forma de onda")
    self.canvas_waveform.draw()
```

Listing 12: Atualização da GUI do receptor (InterfaceGUI/gui_receptor.py)

```
def display_received(self, payload_bits, info):
    self.payload_text.set(payload_bits)
    self.log_box.insert(END, info)
    # atualiza gráficos com forma de onda/constelação
    self.plot_waveform(t_dig, digital_signal_rx)
    self.plot_constellation(constellation_points)
```

5 Simulador

O módulo Simulador integra todas as operações das camadas de Enlace e Física. O transmissor executa a conversão da mensagem para bits, aplica enquadramento, codificação de linha e modulação, enviando o sinal ao canal. O receptor realiza o processo inverso, iniciando pela demodulação e finalizando com a remoção do enquadramento e entrega do payload.

Listing 13: Pipeline principal do transmissor (Simulador/transmissor.py)

```
def run_transmission(config):
    payload_bits = text_to_bits(config["message"])
    frame_bits = framer.frame_char_count(payload_bits)
    baseband = digital_encoder.encode(frame_bits, config["
       mod_digital"])
    t, tx_signal, _ = carrier.modulate(baseband, config["
       mod_portadora"])
    rx_signal = channel.apply_errors(tx_signal, config["error_rate
       "])
    receiver.process_received(rx_signal, config)
```

Listing 14: Processamento principal do receptor (Simulador/receptor.py)

```
def process_received(rx_signal, config):
    bits, digital_rx, t_dig, _ = carrier.demodulate(
        rx_signal, config["mod_portadora"], config,
           digital_encoder)
    recovered = digital_encoder.decode(digital_rx, config["
       mod_digital"])
    ok, info = error_detector.verify_and_correct(recovered)
    payload, _ = framer.deframe_char_count(recovered)
    deliver_payload_to_gui(payload, info)
```

6 Conclusão

Este trabalho apresentou o desenvolvimento de um simulador voltado à análise prática dos principais mecanismos das camadas Física e de Enlace de sistemas de comunicação digital. A ferramenta construída permitiu visualizar, de forma integrada, como modulação, enquadramento e técnicas de detecção e correção de erros influenciam diretamente o processo de transmissão de dados.

Os experimentos realizados mostraram que a representação gráfica dos sinais e quadros é essencial para compreender fenômenos fundamentais da comunicação digital, como variações de amplitude e fase, sincronização de símbolos, comportamento de constelações e propagação de erros ao longo do canal. As simulações também evidenciaram que a escolha do método de enquadramento e dos algoritmos de proteção de dados tem impacto significativo sobre a confiabilidade da transmissão, especialmente em cenários com ruído.

Durante o desenvolvimento, as principais dificuldades estiveram relacionadas à integração consistente entre todas as etapas da transmissão, à manipulação bit a bit exigida pelos métodos de CRC-32 e Hamming, e ao ajuste fino dos parâmetros de modulação para garantir estabilidade visual e coerência dos gráficos. Além disso, decisões de implementação não especificadas no enunciado — como convenções de FLAG e ESC, comportamento do bit stuffing e definição da constelação utilizada — demandaram análise cuidadosa e alinhamento com referências teóricas.

O modelo de canal adotado foi parcialmente ideal, restrito à inserção de erros aleatórios e sem considerar efeitos reais como atenuação, jitter, filtros, interferência intersimbólica ou dispersão. A incorporação desses fenômenos, bem como de modulações adicionais, constitui uma direção natural para trabalhos futuros, possibilitando análises ainda mais realistas.

Por fim, o módulo principal operou como elemento orquestrador entre as camadas Física e de Enlace, aplicando em sequência todas as etapas — enquadramento, codificação de linha, modulação, inserção de ruído, demodulação, decodificação e correção de erros. Essa integração permitiu não apenas validar cada técnica individualmente, mas também observar seu efeito cumulativo sobre o sinal transmitido, reforçando o caráter didático e exploratório do simulador desenvolvido.

Em síntese, o simulador alcançou os objetivos propostos, constituindo uma plataforma eficaz para fins didáticos e experimentais. Como trabalhos futuros, recomenda-se a inclusão de modulações adicionais, canais com modelos estatísticos mais realistas, suporte completo às técnicas previstas no padrão, bem como a evolução da interface gráfica para ambientes interativos de maior desempenho.