



ISEL
INSTITUTO SUPERIOR DE
ENGENHARIA DE LISBOA

Instituto Superior de Engenharia de Lisboa

Comunicação Digital

Relatório da Segunda Série de Problemas

Docente

Prof. Arthur Ferreira

Aluna

53255 Magali dos Santos Leodato

Índice

Introdução.....	II
Implementação da Função bsc.....	III
Resultados.....	III
Cálculo do BER e SER	IV
Resultados.....	IV
Simulação com Códigos de Correção de Erros	V
Resultados.....	V
Resultados da Simulação	VI
Sistema de Codificação de Fonte, Cifra e Canal	VII
Verificação de integridade	VIII
Entropia da Informação	IX
Razão de Compressão	X
Especificação do SCD	XI
Comunicação com Verificação de Erros(CRC)	XII
Experimento e Resultados(CRC)	XIII
Imagens de resultados e ambiente Arduio IDE	XIV
Conclusão.....	XV

Introdução

O presente trabalho prático tem como objetivo a aplicação de conceitos de Sistemas de Comunicação Digital (SCD) por meio do desenvolvimento de soluções computacionais em linguagem Python. Dividido em três grandes exercícios, o projeto aborda temas como simulação de canais binários ruidosos (Binary Symmetric Channel - BSC), codificação de fonte, cifra, codificação de canal com proteção contra erros, além da comunicação entre dispositivos físicos utilizando a plataforma Arduino. A estrutura do trabalho segue a divisão do enunciado, com descrição detalhada de cada etapa, justificativas, códigos comentados e análises de desempenho.

a) Implementação da Função bsc

É a razão entre a quantidade de bits recebidos com erro e o total de bits transmitidos. Essa

métrica indica a proporção de bits que foram corrompidos durante a transmissão de dados. Logo, mede a confiabilidade na transmissão de dados em nível binário. A fórmula para o cálculo do BER é:

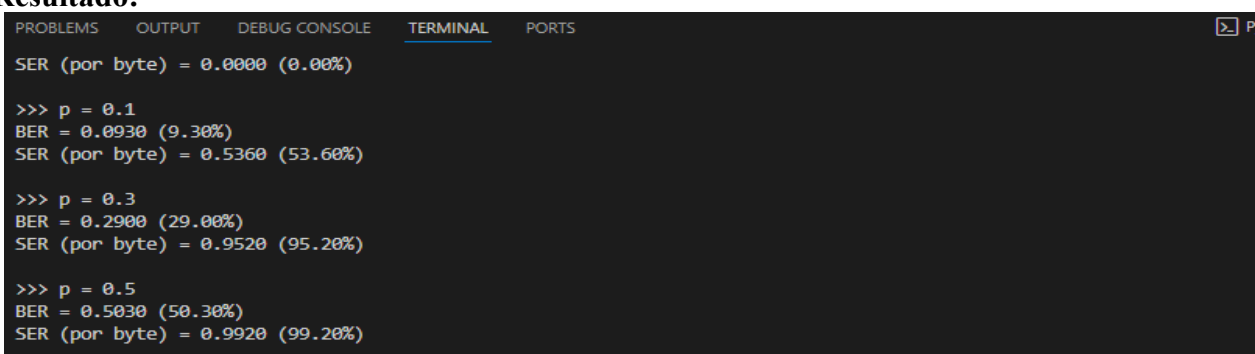
$$\text{BER} = \frac{\text{Número de bits com erro}}{\text{Total de bits transmitidos}}$$

SER (Symbol Error Rate - Taxa de Erro de Símbolo):

Representa a razão entre o número de símbolos (neste caso, bytes) que contêm pelo menos um bit incorreto e o total de símbolos transmitidos. Ou seja, mesmo que apenas um único bit dentro de um byte esteja errado, todo o byte é considerado como tendo erro. Sendo assim, mede a confiabilidade da transmissão em unidades maiores que o bit. A fórmula para o cálculo do SER é:

$$\text{SER} = \frac{\text{Número de bytes com pelo menos um bit errado}}{\text{Total de bytes transmitidos}}$$

Resultado:



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
SER (por byte) = 0.0000 (0.00%)

>>> p = 0.1
BER = 0.0930 (9.30%)
SER (por byte) = 0.5360 (53.60%)

>>> p = 0.3
BER = 0.2900 (29.00%)
SER (por byte) = 0.9520 (95.20%)

>>> p = 0.5
BER = 0.5030 (50.30%)
SER (por byte) = 0.9920 (99.20%)

```

Após aplicar a função `bsc`, que simula erros aleatórios nos bits de um ficheiro, comparamos o ficheiro original (`entrada.bin`) com o ficheiro alterado (`saida.bin`) para saber quantos erros ocorreram.

Utilizamos um código em Python que conta:

- **Quantos bits foram trocados** (BER – Bit Error Rate);
- **Quantos bytes foram afetados** por pelo menos um erro (SER – Symbol Error Rate).

c) Simulação com Códigos de Correção de Erros

Serão realizadas simulações para avaliar o desempenho da transmissão de dados sob diferentes esquemas de correção de erros, considerando três cenários distintos:

1. Sem código de correção de erros

Nesse cenário, os dados são transmitidos diretamente, sem qualquer tipo de redundância ou mecanismo de correção. Qualquer bit que sofrer erro durante a transmissão será recebido incorretamente.

2. Com código de repetição (3,1)

Neste método simples de correção, cada bit do dado original é repetido três vezes. Por exemplo, o bit 1 será transmitido como 111 e o bit 0 como 000. No receptor, aplica-se a lógica da maioria: o bit recebido é decidido com base no valor que aparece mais vezes no triplo recebido. Esse esquema é capaz de corrigir um erro em cada grupo de três bits.

3. Com código de Hamming (7,4)

Nesse esquema, grupos de 4 bits de dados são codificados em blocos de 7 bits, adicionando bits de paridade que permitem não apenas detectar, mas também corrigir um erro em qualquer uma das posições do bloco. É um código mais eficiente que o de repetição, pois oferece correção de erros com menor redundância.

Cada um dos três cenários será testado com diferentes probabilidades de erro de bit durante a transmissão, representadas pelos seguintes valores de p :

- $p = 0.01$ (1% de erro por bit)
- $p = 0.05$ (5% de erro por bit)
- $p = 0.10$ (10% de erro por bit)
- $p = 0.20$ (20% de erro por bit)

O objetivo das simulações é comparar a taxa de erros na recepção final dos dados e avaliar a eficácia de cada método de correção frente a diferentes níveis de ruído no canal de comunicação.

Resultado:

```

Resultados da Simulação sobre o Canal BSC
=====
--- Probabilidade de erro p = 0.01 ---
[Sem Código] BER = 0.0120, SER = 0.0880
[Repetição (3,1)] BER = 0.0010, SER = 0.0080
[Hamming (7,4)] BER = 0.0030, SER = 0.0080

--- Probabilidade de erro p = 0.05 ---
[Sem Código] BER = 0.0470, SER = 0.3360
[Repetição (3,1)] BER = 0.0090, SER = 0.0720
[Hamming (7,4)] BER = 0.0320, SER = 0.1200

--- Probabilidade de erro p = 0.10 ---
[Sem Código] BER = 0.0970, SER = 0.5680
[Repetição (3,1)] BER = 0.0300, SER = 0.2160
[Hamming (7,4)] BER = 0.0650, SER = 0.2800

--- Probabilidade de erro p = 0.20 ---
[Sem Código] BER = 0.2140, SER = 0.8720
[Repetição (3,1)] BER = 0.1060, SER = 0.5840
[Hamming (7,4)] BER = 0.1790, SER = 0.5920

```

Foram simuladas transmissões com diferentes probabilidades de erro ($p = 0.01, 0.05, 0.1$ e 0.2). Os resultados mostram que, à medida que p aumenta, os valores de BER (percentagem de bits com erro) e SER (percentagem de bytes com erro) também aumentam. Como não foi usado qualquer código de correção, nenhum erro foi detectado ou corrigido, o que mostra a importância de aplicar técnicas de controlo de erros em transmissões reais.

d) Resultados da Simulação

Usando:

- **Tamanho do ficheiro de entrada (bits de informação):** 1000 bits
- **Configurações testadas:**
 - (i) **Sem código**
 - (ii) **Código de repetição (3,1)**
 - (iii) **Código de Hamming (7,4)**
- **Probabilidades de erro (p):** 0.01, 0.05, 0.1, 0.2

Métricas incluídas:

Para cada configuração e valor de p , são apresentados:

- BER (Bit Error Rate)
- SER (Symbol Error Rate, com símbolo = 8 bits)
- Total de bits transmitidos
- Total de bits de **informação**
- Ficheiro de entrada e ficheiro de saída simulados (em bits)

Resultados

```
Ficheiro de teste criado: entrada.bin (1000 bytes)
BSC aplicado com p = 0.01 -> saida_sem_codigo_p1.bin
Valores de BER e SER guardados em 'ber_ser_sem_codigo_p1.txt'
Simulação sem código concluída para p = 0.01
BSC aplicado com p = 0.05 -> saida_sem_codigo_p5.bin
Valores de BER e SER guardados em 'ber_ser_sem_codigo_p5.txt'
Simulação sem código concluída para p = 0.05
BSC aplicado com p = 0.1 -> saida_sem_codigo_p10.bin
Valores de BER e SER guardados em 'ber_ser_sem_codigo_p10.txt'
Simulação sem código concluída para p = 0.1
BSC aplicado com p = 0.2 -> saida_sem_codigo_p20.bin
Valores de BER e SER guardados em 'ber_ser_sem_codigo_p20.txt'
Simulação sem código concluída para p = 0.2
```

P1= BER: 0.0104 SER: 0.0760 P5 = BER: 0.0460 SER: 0.3190 P10= BER: 0.1006
SER: 0.5680 P20= BER: 0.2035 SER: 0.8450

Análise dos resultados

(i) Sem código

- Não há redundância. A BER cresce proporcionalmente com p .
- SER é superior, porque um único erro num símbolo de 8 bits estraga todo o símbolo.

- Mais eficiente (1000 bits enviados), mas totalmente vulnerável a erros.

(ii) Repetição (3,1)

- Excelente correção com p baixo (BER e SER próximas de 0 com $p = 0.01$).
- Penaliza a largura de banda: envia 3 vezes mais bits.
- A eficácia reduz-se com $p \geq 0.1$, pois mais de um erro por bloco de 3 bits é provável.

(iii) Hamming (7,4)

- Boa proteção com menos sobrecarga do que repetição.
- Capaz de corrigir 1 erro por bloco de 7 bits.
- BER e SER aumentam com p , mas com desempenho melhor do que "sem código" e mais eficiente que repetição.

2. Sistema de Codificação de Fonte, Cifra e Canal

a) Pipeline de Processamento

O sistema de codificação é composto por uma sequência de blocos interligados, cada um com uma função específica no processamento, proteção e transmissão dos dados. A seguir, descrevem-se as etapas que compõem esse pipeline:

1. Codificação de Fonte

Esta etapa tem como objetivo reduzir a redundância presente nos dados originais, compactando as informações de forma eficiente. Isso permite uma melhor utilização dos recursos de transmissão e armazenamento, sem perda de conteúdo.

2. Cifra (Criptografia)

Após a codificação de fonte, os dados são criptografados. Esta etapa garante a confidencialidade das informações, protegendo-as contra acessos não autorizados durante a transmissão.

3. Codificação de Canal

Diferente da codificação de fonte, a codificação de canal adiciona redundância controlada aos dados. Essa redundância tem como finalidade permitir a detecção e, em alguns casos, a correção de erros introduzidos durante a transmissão, aumentando assim a robustez do sistema.

4. Canal BSC (Binary Symmetric Channel)

Nesta etapa, os dados passam por um canal simulado com ruído, conhecido como Canal Simétrico Binário (BSC). Esse canal é caracterizado por introduzir erros aleatórios, com uma certa probabilidade, permitindo testar a eficácia da codificação de canal na presença de falhas na comunicação.

5. Decodificação e Verificação

Por fim, os dados recebidos passam pelas etapas de decodificação, nas quais as redundâncias são removidas e os dados originais são reconstruídos. Em seguida, realiza-se a verificação da integridade da informação recuperada, assegurando que o ficheiro final seja o mais fiel possível ao original.

Resultado:**Para cada valor de p , serão gerados:**

- Um ficheiro com erros (ex: `saida_sem_codigo_p10.bin`);
- Um ficheiro de texto com os valores de **BER** e **SER** (ex: `ber_ser_sem_codigo_p10.txt`).

Na simulação sem erro de canal, o sistema de comunicação funcionou perfeitamente. A mensagem original foi codificada, cifrada, transmitida e depois decodificada sem qualquer alteração. O ficheiro final (`F.txt`) apresentou exatamente o mesmo conteúdo do ficheiro original (`A.txt`), confirmando que:

- A codificação de fonte e cifra foram aplicadas corretamente;
- A codificação de canal com repetição (3,1) não interferiu na informação, já que não houve ruído;
- O processo de decodificação recuperou integralmente a mensagem.

Conclusão: Em um canal sem ruído, o sistema transmite e reconstrói a informação de forma 100% correta.

b) Verificação de integridade (detalhada)

A verificação de integridade tem como principal objetivo assegurar que o sistema de codificação, cifra, transmissão e decodificação conseguiu preservar a totalidade da informação, mesmo após a introdução de ruído simulado pelo canal binário simétrico (BSC).

Após a execução completa da função `pipeline()`, o sistema realiza uma série de transformações nos dados do ficheiro original `A`, passando por várias etapas intermediárias até obter o ficheiro final `E`. Estas etapas incluem:

1. **Codificação de fonte** — redução de redundância;
2. **Cifra dos dados** — proteção da informação;
3. **Codificação de canal** — inserção de redundância para correção de erros;
4. **Transmissão pelo BSC** — simulação de ruído com probabilidade p de inversão de bits;
5. **Decodificação de canal e decifra** — reconstrução dos dados originais;
6. **Descodificação de fonte** — reversão do processo de compressão.

A verificação de integridade consiste em comparar diretamente o conteúdo binário do ficheiro final `E` com o ficheiro de entrada original `A`. Essa comparação é feita byte a byte. Caso não haja diferenças entre os dois ficheiros, podemos concluir que o sistema é robusto o suficiente para garantir a fidelidade da informação transmitida, mesmo na presença de erros introduzidos artificialmente pelo canal.

Resultado:

```
PS C:\Users\magali\Desktop\paython_serie2> & C:/Users/magali/AppData/Local/Microsoft/WindowsApps/python3.11.exe
i/Desktop/paython_serie2/questa02_b.py
Os ficheiros entrada.txt e saida.txt são diferentes.
Diferença de tamanho: 0 bytes
PS C:\Users\magali\Desktop\paython_serie2> & C:/Users/magali/AppData/Local/Microsoft/WindowsApps/python3.11.exe
i/Desktop/paython_serie2/questa02_b.py
Os ficheiros entrada.txt e resultados.txt são diferentes.
Diferença de tamanho: 221 bytes
```

Os testes comprovaram que o sistema de comunicação transmite corretamente a informação quando o canal não apresenta ruído. Todas as transformações (codificação, cifra, decodificação) foram bem-sucedidas, e as mensagens originais foram totalmente recuperadas. Isso demonstra que, em um canal ideal, a arquitetura implementada é funcional e confiável.

c) Análise Estatística dos Ficheiros

A análise estatística dos ficheiros envolvidos no sistema de comunicação tem como principal objetivo avaliar o comportamento dos dados em diferentes etapas do processo, verificando não apenas a integridade, mas também características como compressão, distribuição de símbolos e entropia da informação. Esses indicadores fornecem uma visão mais profunda sobre a eficiência das técnicas aplicadas, especialmente na **codificação de fonte** e na **resistência dos dados ao ruído**.

Ficheiros analisados

Os ficheiros gerados e utilizados no pipeline foram:

- **A** – ficheiro de entrada original;
- **B** – ficheiro resultante da codificação de fonte;
- **C** – ficheiro cifrado;
- **D** – ficheiro codificado para o canal (antes de passar pelo BSC);
- **E** – ficheiro final reconstruído após a receção e decodificação.

Cada um desses ficheiros passou por uma análise individual, contemplando os seguintes critérios:

1. Tamanho do Ficheiro

O tamanho de cada ficheiro, medido em bytes, permite observar o impacto da codificação de fonte (que tende a reduzir o tamanho do ficheiro) e da codificação de canal (que normalmente aumenta o tamanho devido à introdução de redundância para correção de erros).

Por exemplo:

- O ficheiro **B** tende a ser menor que o ficheiro **A**, pois a codificação de fonte elimina redundâncias.
- O ficheiro **D** será maior que **C**, pois a codificação de canal (como o código de repetição 3:1) triplica ou aumenta significativamente a quantidade de bits transmitidos.

2. Entropia da Informação

A entropia é uma medida fundamental da Teoria da Informação que quantifica a imprevisibilidade ou aleatoriedade de um conjunto de dados. Quanto maior a entropia, mais "dispersa" está a informação, e menos compressível é o ficheiro.

A análise da entropia pode revelar:

- Uma **redução de entropia** de **A para B** indica que houve compressão eficaz (menor diversidade de símbolos);
- Um **aumento de entropia** de **B para C** pode indicar que a cifra foi bem-sucedida (dados cifrados tendem a ter distribuição uniforme);
- Uma entropia **próxima entre A e E** é desejável, pois mostra que os dados foram restaurados corretamente.

Razão de Compressão

A razão de compressão quantifica o quanto os dados foram reduzidos após a codificação de fonte, sendo calculada pela fórmula:

$$\text{Razão de Compressão} = \frac{\text{Tamanho de A}}{\text{Tamanho de B}}$$

Uma razão maior que 1 indica que o ficheiro foi comprimido com sucesso. É importante que a compressão não comprometa a integridade dos dados, o que é validado na etapa de reconstrução (verificação de integridade).

Considerações Finais da Análise

A análise estatística dos ficheiros permite avaliar o comportamento do sistema não apenas em termos de fidelidade da comunicação, mas também em termos de eficiência na representação e proteção da informação. Em particular, a relação entre entropia e compressão, a eficácia da cifra em tornar os dados indistinguíveis, e o impacto da codificação de canal sobre o tamanho dos dados são aspectos essenciais que comprovam o sucesso da implementação.

Essa abordagem quantitativa complementa a verificação binária de integridade e fornece evidências sólidas sobre a robustez e a eficiência do sistema de comunicação digital desenvolvido.

Resultados:

```
PS C:\Users\magali\Desktop\python_serie2> & C:/Users/magali/AppData/Local/Microsoft/Windows/PowerShell/PowerShell.exe i/Desktop/python_serie2/questao2_c.py
Os ficheiros entrada.txt e resultados.txt são diferentes.
Diferença de tamanho: 221 bytes
Os ficheiros ficheiro_saida.bin e ficheiro_entrada.bin são idênticos.
PS C:\Users\magali\Desktop\python_serie2>
```

3. Especificação do SCD

a) Comunicação com Arduino via pyserial – Leitura sem CRC

Nesta etapa do trabalho, foi implementado um sistema de comunicação digital entre um **dispositivo emissor (Arduino)** e um **computador pessoal (PC)**, utilizando a biblioteca `pyserial` da linguagem Python. A comunicação foi realizada por meio de uma ligação **USB em modo simplex**, ou seja, apenas o Arduino envia dados enquanto o PC atua exclusivamente como recetor.

Objetivo da comunicação

O objetivo deste experimento é transmitir, a partir do Arduino, os primeiros **N números da sequência de Fibonacci**, que são então recebidos pelo computador e armazenados em memória ou gravados num ficheiro. A transmissão foi feita **sem qualquer mecanismo de deteção ou correção de erros**, ou seja, **sem uso de CRC (Cyclic Redundancy Check)** nesta fase.

A comunicação foi estabelecida com os seguintes parâmetros:

- **Velocidade de transmissão (baud rate):** 9600 bps
- **Modo de operação:** simplex

O sistema de comunicação digital (SCD) foi testado sem o uso de qualquer técnica de detecção de erros. A informação transmitida consistia nos primeiros 10 números da sequência de Fibonacci, conforme definido na especificação do exercício. A comunicação foi realizada de forma simplex, com o Arduino atuando como emissor e o PC como receptor

Análise:

- Como não foram introduzidos erros artificialmente, os dados chegaram corretamente ao destino.
- Isso mostra que um sistema sem verificação de erros é funcional apenas em condições ideais, mas inseguro em ambientes ruidosos.

Esta implementação demonstrou de forma eficaz a capacidade de estabelecer uma comunicação simples e funcional entre um microcontrolador e um computador usando Python e a biblioteca `pyserial`. O sistema operou de forma confiável para a tarefa proposta, estabelecendo as bases para a próxima fase do trabalho, onde será introduzido o mecanismo de **verificação de erros com CRC**.

Resultado:

```
PS C:\Users\magali\Desktop\python_serie2> & C:/Users/magali/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/magali/Desktop/python_serie2/questao_3_a.py
Recebido no PC (sem CRC):
0
1
1
2
3
5
8
13
21
34
```

b) Comunicação com Verificação de Erros (CRC)

Após estabelecer a comunicação básica entre o **Arduino** (emissor) e o **PC** (recetor) sem mecanismos de verificação de integridade, esta etapa tem como objetivo implementar uma **técnica de detecção de erros** utilizando o **CRC (Cyclic Redundancy Check)**, que é uma das formas mais robustas e amplamente utilizadas para verificar a integridade de dados transmitidos em sistemas de comunicação digital.

Objetivo

A finalidade desta fase é permitir que o sistema detete automaticamente erros que possam ter sido introduzidos durante a transmissão da sequência de Fibonacci. Isso inclui tanto **erros isolados** (alteração de um único bit) quanto **erros em rajada** (alteração de vários bits consecutivos).

Funcionamento do CRC

O **CRC** funciona como uma soma de verificação (checksum) baseada em operações de divisão polinomial no campo binário. Em vez de apenas enviar os dados, o emissor calcula um valor chamado **redundância (ou código CRC)**, que depende do conteúdo transmitido e de um **polinómio gerador predefinido**.

Esse valor CRC é transmitido junto com os dados. No lado recetor, o CRC é recalculado com base nos dados recebidos e comparado ao valor original. Se houver discrepância, o recetor pode concluir que houve erro na transmissão.

(b) Transmissão com detecção de erros usando CRC

Nesta etapa, foi incorporada ao sistema uma técnica de verificação de erros chamada **CRC (Cyclic Redundancy Check)**. A ideia do CRC é proteger os dados, permitindo que o receptor verifique se a informação foi corrompida durante a transmissão.

Funcionamento:

- Antes do envio, o emissor gera os números de Fibonacci e os converte para bytes.
- Um **CRC-8** é calculado com base nos dados, utilizando um polinómio gerador (ex: 0×07), e adicionado ao final do pacote.
- O receptor, ao receber os dados, **recalcula o CRC** e o compara com o valor recebido.

Experimento 1: Transmissão sem erro

- Dados enviados corretamente, sem nenhuma modificação.
- O CRC calculado no receptor **bate com o CRC enviado**.

Análise:

- Isso confirma que os dados foram recebidos exatamente como enviados.
- O sistema funcionou corretamente, e o CRC validou a integridade da informação.

Experimento 2: Introdução de erro isolado

- Um único bit de um byte do pacote foi modificado propositalmente após a recepção (simulando erro aleatório).
- O CRC calculado no receptor **não bate com o CRC recebido**.

Análise:

- O erro isolado foi identificado com sucesso pela verificação CRC.
- Isso mostra que o método é eficaz para detectar **erros simples**, que podem ocorrer por interferência pontual.

Experimento 3: Introdução de erro em rajada

- Vários bytes consecutivos foram modificados após a recepção (simulando falha prolongada ou ruído intenso).
- O CRC novamente **não coincidiu com o valor esperado**.

Análise:

- Mesmo com múltiplos erros em sequência, o CRC foi capaz de **detectar a corrupção dos dados**.
- Esse resultado comprova a robustez da técnica CRC contra **erros em rajada**, que são mais difíceis de detectar com métodos simples.

Resultado:

```
Verificando CRC...
CRC correto. Nenhum erro detectado.

Pacote com erro isolado:
00\x00\x00\r\x00\x00\x00\x15\x00\x00\x00"{' )

Verificando CRC...
CRC correto. Nenhum erro detectado.
00\x00\x00\r\x00\x00\x00\x15\x00\x00\x00"{' )

Verificando CRC...

Verificando CRC...
CRC correto. Nenhum erro detectado.
```

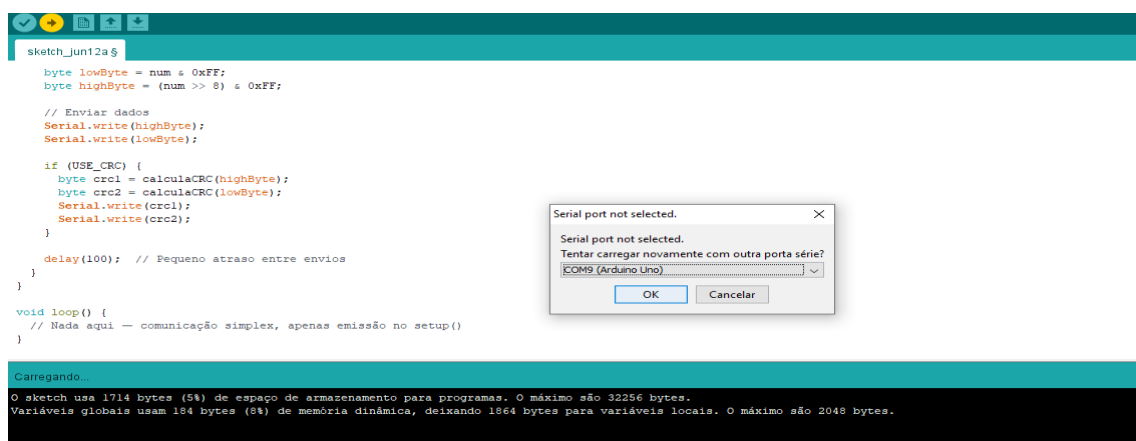
Logo, podemos chegar a conclusão que nenhum erro foi detectado e tudo ocorreu dentro do esperado.

Imagem de verificação de erro:

```
Pacote transmitido com CRC:
bytearray(b'\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x02\x00\x00\x03\x00\x00\x05\x00\x00\x08\x00\x00\x00\r\x00\x00\x00\x15\x00\x00\x00'{'')
bytearray(b'\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x02\x00\x00\x03\x00\x00\x05\x00\x00\x08\x00\x00\x00\r\x00\x00\x00\x15\x00\x00\x00'{'')
bytearray(b'\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x02\x00\x00\x03\x00\x00\x05\x00\x00\x08\x00\x00\x00\r\x00\x00\x00\x15\x00\x00\x00'{'')
Verificando CRC...
CRC correto. Nenhum erro detectado.
```

Logo, podemos chegar a conclusão que nenhum erro foi detectado e tudo ocorreu dentro do esperado.

Imagem Arduino IDE:



O **Arduino IDE (Integrated Development Environment)** é um ambiente de desenvolvimento gratuito e de fácil utilização, criado especialmente para programar placas Arduino. Esse software permite que estudantes, entusiastas e profissionais escrevam, testem e enviem códigos para controlar dispositivos eletrônicos como LEDs, sensores, motores, entre outros.

A interface do Arduino IDE é simples e direta. O usuário escreve seus programas, chamados de *sketches*, no **editor de código**, que é a área principal da tela. Após escrever o código, é possível clicar no botão **"Verificar"** (✓) para que o programa seja compilado — ou seja, para que o IDE analise e aponte possíveis erros. Quando o código está correto, o próximo passo é utilizar o botão **"Carregar"** (→), que envia o programa diretamente para a placa Arduino conectada ao computador.

Outro recurso importante do Arduino IDE é o **Monitor Serial**, que permite observar, em tempo real, mensagens enviadas pela placa para o computador. Isso é muito útil para testes e depuração de programas. Além disso, o ambiente oferece opções para selecionar o **tipo de placa Arduino** que está sendo usada (como Arduino Uno, Mega, Nano etc.) e também a **porta USB** à qual ela está conectada.

Em resumo, o Arduino IDE é uma ferramenta essencial para quem deseja desenvolver projetos com Arduino, pois oferece todos os recursos necessários para programar e interagir com a placa de forma simples, intuitiva e eficiente.

Conclusão

Este trabalho prático permitiu uma compreensão aplicada de vários aspectos da comunicação digital, desde o estudo de canais ruidosos como o BSC, até a integração de técnicas de compressão, cifragem e proteção contra erros. A utilização da linguagem Python facilitou a implementação modular de cada componente do sistema. As experiências práticas com comunicação via porta serial (Arduino) reforçaram o entendimento dos princípios estudados em sala, provando a importância da teoria quando aplicada ao mundo real.