Área Departamental de Engenharia de Eletrónica e Telecomunicações e de Computadores

Módulo 2 do Trabalho Prático

Autores:	49487	Ricardo Duarte António Rovisco
	49504	Jorge Filipe de Medeiros Palácios da Silva
	49508	João Miguel Castanheira Mota

Relatório para a Unidade Curricular de Comunicação Digital da Licenciatura em Engenharia Informática e de Computadores.

Professor: Engenheiro Vitor Fialho

19 – 06 – 2023

Índice

1. Introdução

No âmbito da unidade curricular Comunicação Digital, 2ºAno/4ºSemestre- lecionada pelo docente Vitor Fialho foi proposta a realização de um relatório referente ao segundo trabalho pratico. Neste segundo trabalho foi realizado uma série de exercícios com o objetivo de consolidar a matéria lecionada em aula, Codificação de Canal, Banda Base, Sinais, Sistemas e Banda Canal.

2. Primeiro exercício

No primeiro exercício foi nos designado a tarefa de usar o exercício 5 onde usando o conhecimento sobre Codificação de Canal codificámos uma função que implementa o modelo *Binary Symmetric Channel(BSC)*, figura 1 e 2, onde recebe uma sequência binária e o valor pretendido para o BER como parâmetro e retorna a sequência binária resultante da passagem do canal.

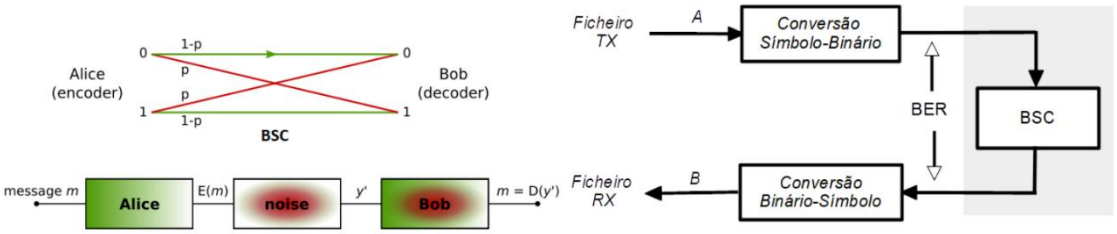

Figura 1 e 2 – Modelo de canal binário simétrico e a sua aplicação sobre o ficheiro

Nesta tarefa foi nos dados os valores de probabilidade de erro de bit $p \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$ com o objetivo de calcular diversas BER's de 4 ficheiros à nossa escolha. A BER_1 entre a entrada e a saída do BSC, sem controlo de erros, BER_2 após a aplicação do código de repetição (3,1) sobre o BSC, modo de correção, BER_3 após a aplicação do código de *hamming* (7,4) sobre o BSC, em modo de correção também. Foi enviado em anexo todos os inputs e outputs de cada ficheiro tendo lá também valores importantes indicando a BER utilizada, número total de bits que passam pelo BSC, a BER depois do BSC e Número de símbolos errados após a saída do BSC.

Para implementar o *Binary Symmetric Channel* foi necessário antes criar duas funções de conversão, uma de *string* para binário e outra de binário para *string* uma vez que o BSC apenas aceita valores em código binário.

A BER apesar de ser calculada da mesma forma independentemente do controlo de erros, esta varia consoante o seu controlo de erros. A BER é calculada apartir dos bits errados, onde os vamos contar e dividir pelo número de bits enviados.

Como mencionado anteriormente, foi desenvolvido o código de controlo de erros *hamming* (7,4) onde o 7 significa que são 7 bits totais do código, 4 deles são de mensagem e por exclusão de partes os outros 3 são de paridade. Neste código os bits de paridade possuem fórmulas para o seu cálculo, designadas como equações de paridade:

$$b_0 = m_1 \oplus m_2 \oplus m_3$$
$$b_1 = m_0 \oplus m_1 \oplus m_3$$
$$b_2 = m_0 \oplus m_2 \oplus m_3$$

$\mathbf{c} = [m_0\ m_1\ m_2\ m_3\ b_0\ b_1\ b_2]$

Figura 3 – Equações de paridade para código *hamming* (7,4)

Para ser possível identificar os erros, foi necessário observar o valor do síndroma gerador que identifica o bit errado na palavra de código gerada, sendo mais tarde feita a correção do mesmo.

Síndroma	Padrão de Erro	Observações
000	0000000	Ausência de erro
011	1000000	1.º bit em erro
110	0100000	2.º bit em erro
101	0010000	3.º bit em erro
111	0001000	4.º bit em erro
100	0000100	5.º bit em erro
010	0000010	6.º bit em erro
001	0000001	7.º bit em erro

Figura 4 – Tabela de síndromas para código de *hamming* (7,4)

Noutra opção, onde se calcula a BER após aplicado o código de repetição (3,1), no código foi repetido o mesmo bit 3 vezes antes de ser enviado para garantir que ocorre o menor número de erros possíveis. Ou seja, para cada bit verificou-se se o mesmo era 1 ou 0, três vezes diferentes. Dessas 3 vezes, o valor do bit enviado é o que tem o maior número de repetição. Se 1 aparecer 2 vezes e 0 uma vez é enviado o valor de bit 1 mesmo que esteja errado ou vice versa.

Para a BER da primeira opção, é apenas necessário usar o que foi feito no trabalho anterior e assim usando apenas o BSC, sem qualquer tipo de controlo de erros, calcula-se a BER. Após o cálculo das BER's pode se notar que o controlo de erros mais eficaz é o *Cyclic Redundancy Check* (CRC, código de repetição), sendo o código com menos erros em cada BER introduzida e com o retorno de BER mais pequeno, por vezes nula. Notou-se também um melhor desempenho do canal sem qualquer controlo de erros que o canal com a aplicação de código *hamming* (7,4). Apesar do código de *hamming* ser mais eficiente a detetar erros do que a corrigir os mesmos, estes resultados não são os esperados, podendo o código desenvolvido ter algum erro que não foi identificado após a revisão do mesmo. Estes resultados também podem resultar da correção de um bit que não estaria em erro.

A segunda parte deste primeiro exercício é fazer os mesmos passos, de usar o BSC sobre diversos ficheiros mas desta vez com a aplicação da técnica de entrelaçamento (*interleaving*) , técnica que lê a matriz da figura coluna a coluna de cima para baixo.

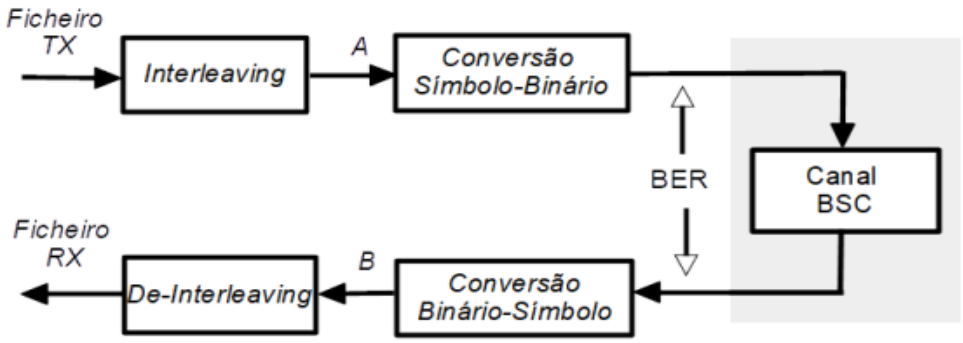

Figura 5 – Aplicação da técnica de entrelaçamento

Um exemplo de entrelaçamento de transmissão é chamando a função, cujo número de colunas da matriz é igual a 4, a mensagem ISEL LEIC G11T42D seria codificada de forma a ficar ILE 12S IGTDELC14. Na mesma mensagem foi introduzida a BER igual a 0.15, sendo que no ponto A foi obtido a mensagem codificada I C1DSL TEEG4LI12 e no ponto B foi obtido a mensagem codificada I`S3TS^dPE G4NY1. Como se pode ver pelos resultados do exemplo dado, dentro do bloco BSC ocorreram erros no envio de bits dentro do canal BSC.

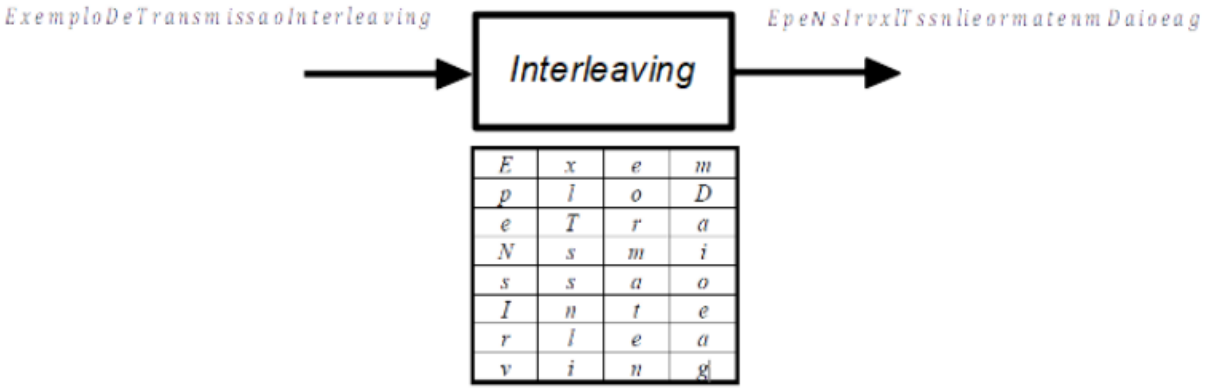

Figura 6 – Exemplo de conceito da técnica de entrelaçamento (*interleaving*)

Neste caso verificou-se resultados semelhantes aos apresentados na primeira parte do exercício. O código de controlo de erros CRC voltou a sair à frente, apresentando o menor retorno de BER e o menor retorno de bits errados. Mais uma vez, devido à sua falta de eficiência de correção de erros, o código de *hamming* retornou um BER maior que a utilização de BSC sem qualquer controlo. Entre a aplicação da técnica de entrelaçamento e apenas a utilização do BSC houve pouca alteração no número de erros, sendo que algumas BER's foram mais altas com a não utilização de entrelaçamento e outras mais altas com a utilização da técnica.

3. Segundo exercício

No segundo e último exercício deste módulo foi necessário usar Arduíno para além da implementação de código no software. A conexão SCD entre software e hardware será feita via USB numa ligação *simplex*, modo de transmissão em sentido unidirecional, onde o Arduíno será o emissor e o PC será o recetor (figura 7). A informação que se deve transmitir são os *N* primeiros termos de uma progressão geométrica, de primeiro termo *u* e razão *r*, como foi implementada num exercício da primeira Aula Prática que também serviu como primeiro exercício de Módulo 1. A informação recebida pelo PC, no nosso caso, é escrita na consola. O objetivo deste exercício é comparar a transmissão de dados com e sem deteção de erros usando a técnica *Fletcher checksum*, mediante um parâmetro de entrada.

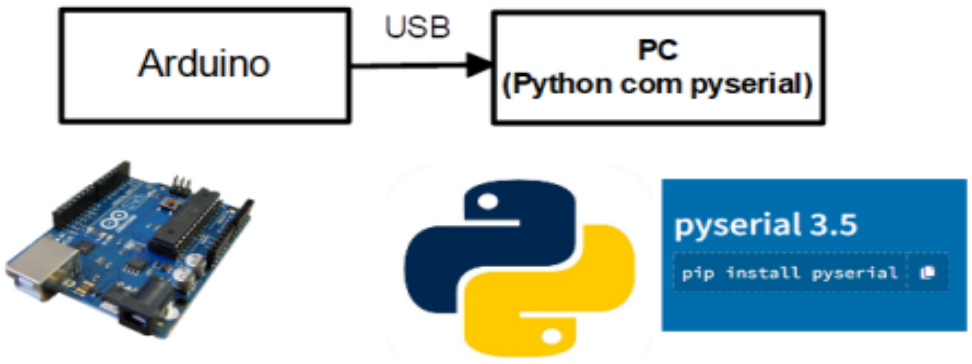

Figura 7 – Ilustração do SCD a funcionar em modo *simplex* com a comunicação do Arduíno (emissor) e o PC (recetor)

```
Available ports:
COM3 - Arduino Uno (COM3)
COM1 - Porta de comunicações (COM1)
select Port: COM3
COM3 - Arduino Uno (COM3)
CLICK ENTER TO START OR WRITE EXIT TO LEAVE:
```

Figura 8 – Interface da aplicação

A técnica de deteção de erros *Fletcher checksum* é um algoritmo utilizado para verificar a integridade dos dados transmitidos. O algoritmo calcula um valor de verificação (*checksum*) com base nos dados de entrada. O valor é então transmitido junto com os dados. Ao receber esses dados, o algoritmo é novamente aplicado aos dados e o novo valor de verificação é comparado ao original. Se os dois valores forem iguais, significa que os dados não foram corrompidos durante

a transmissão, caso contrário, um erro é detetado. A técnica utiliza aritmética modular para calcular os valores de verificação. Os dados de entrada são divididos em palavras de tamanho fixo e os valores de verificação são atualizados iterativamente enquanto os dados são processados. O algoritmo é projetado para detetar erros comuns, como inversões e repetições de bits. Embora a técnica de *Fletcher checksum* seja útil para detetar erros, não é capaz de corrigi-los. Apenas identifica a necessidade e retransmissão dos dados ou de outras ações corretivas.

Para começar foi enviada os primeiros 5 termos de uma progressão geométrica de razão igual a 2 em que o primeiro termo é 3 sem realizar deteção de erros apenas para comprovar o seu funcionamento. Como esperado os valores recebidos foram 3, 6, 12, 24,48.

Um problema da técnica de *Fletcher checksum* é que, por si só, não é projetada especificamente para deteção de erros *burst*. Um erro *burst*, refere-se a um tipo de erro que ocorre em sequências contíguas de bits em um fluxo de dados. Em vez de erros individuais e dispersos, um erro *burst* envolve uma série de bits consecutivos afetados por algum tipo de corrupção. Em uma situação em que ocorrem erros *burst* consecutivos, é possível que o algoritmo de *Fletcher* detete a presença de erros, desde que esses erros afetem as palavras de dados usadas no cálculo do *checksum*. No entanto, a deteção de tal erro não é garantida, pois o algoritmo não é otimizado para esse tipo específico de erro.

Para detetar erros *burst* com a técnica de *Fletcher checksum* foi necessário adicionar lógica adicional para verificar a presença de erros em sequências consecutivas de bits. Para tal, na função *createError* um erro é inserido com uma probabilidade de 40%, onde primeiro é selecionado aleatoriamente o primeiro onde vão começar os erros. Se um erro é introduzido, a representação binária dos dados é obtida e, em seguida, um aleatório número de bits é alterado. Isso inclui a possibilidade de alterar bits consecutivos, permitindo a simulação de erros *burst*. Em seguida, o *checksum de Fletcher* é recalculado com os dados modificados permitindo assim a deteção de erros *burst* juntamente com outros tipos de erros.

Ao iniciar o SCD com o controlo de erros pode-se notar com os resultados que alguns termos não tiveram qualquer tipo de erros (figura 10) enquanto outros tiveram (figura 9).

```
Valor recebido: 3
Checksum do valor recebido: 0x330033
Checksum do valor recebido com erro: 0x1f300c8
Houve erro
```

Figura 9 – Exemplo de que houve erro

```
Valor recebido: 12
Checksum do valor recebido: 0x940063
Checksum do valor recebido com erro: 0x940063
Não houve erro
```

Figura 10 – Exemplo de que não houve erro

O controlo de erros que deteta erros *burst*, deteta também a quantidade de bits que estão errados (figura 11).

```
Valor recebido: 3
Checksum do valor recebido: 0x330033
Número de bits errados: 4
Checksum do valor recebido com erro: 0x9f0069
Houve erro
```

Figura 11 – Exemplo de que houve erros e a quantidade de bits errados.

```
Valor recebido: 12
Checksum do valor recebido: 0x940063
Checksum do valor recebido com erro: 0x940063
Não houve erro
```

Figura 12 – Exemplo de que não houve erro

4. Conclusão

Apesar da possibilidade de erros no desenvolvimento do código *hamming* do primeiro exercício, podemos concluir que todas as funções foram devidamente estudadas, desenvolvidas e corrigidas ao ponto de não desenvolverem qualquer tipo de erro na passagem dos testes realizados. Com isso foi possível fazer todos os exercícios corretamente e obter os resultados esperados.