Universidade Federal De Pernambuco

Departamento de Eletrônica e Sistemas

Matemática Discreta

2021.2

Projeto 2

Dayvison Pedro Pilar da Silva

Lucas Givaldo dos Santos Torres

Gabriela Leite Pereira

Sumário

1 Introdução

Na transmissão de dados, na vida real, às vezes ocorrem problemas, como interferências eletromagnéticas ou erros humanos (como por exemplo, erros de digitação) que chamamos de ruído e que fazem com que a mensagem recebida seja diferente daquela que foi enviada. O objetivo da Teoria dos Códigos Corretores de Erros é desenvolver métodos sistemáticos que permitam detectar e corrigir tais erros. Esta teoria teve início na década de quarenta quando os computadores eram máquinas muito caras e apenas instituições de grande porte como o governos e as universidades tinham condições de mantê-los. Os computadores eram usados para executar tarefas numéricas complexas, como calcular órbitas planetárias com precisão significativa e processar dados estatísticos muito densos. Porém, os computadores demoravam muito tempo para executar tais tarefas e o resultado muitas vezes não era satisfatório ou sequer existia devido à ocorrência de erros.

Não "aceitando" esta contradição entre os computadores conseguirem executar tarefas complexas e tropeçarem em erros relativamente simples, Richard W. Hamming cogitou o seguinte: "se as máquinas podem detectar um erro, por que não podemos localizar a posição do erro e corrigi-lo?". Este questionamento levou Hamming a desenvolver um código capaz de corrigir um erro, se ele for o único. Posteriormente, a criação de Hamming motivou outras pessoas a desenvolverem outros tipos de códigos, cada vez mais eficientes e potentes, como, por exemplo, o Código de Golay (24, 4096, 8) que foi usado pela espaçonave Voyager para transmitir fotografias coloridas de Júpiter e Saturno. Desde então, os códigos tem aprimorado diversos processos na Engenharia Eletrônica, como o código de canal aplicado a imagem e análise de desempenho para códigos de canais, que serão abordadas neste projeto.

2 Fundamentação Teórica

2.1 Análise de desempenho para códigos de canais

O canal Binário Simétrico(BSC) é um modelo mais simples de comunicação bastante utilizado na teoria da informação que dá base para outros modelo complexos de canais. O BSC funciona da seguinte forma: um transmissor envia um bit(0 ou 1), nessa transferência, pode haver um erro de probabilidade p de acontecer e o bit ser transferido de forma invertida.

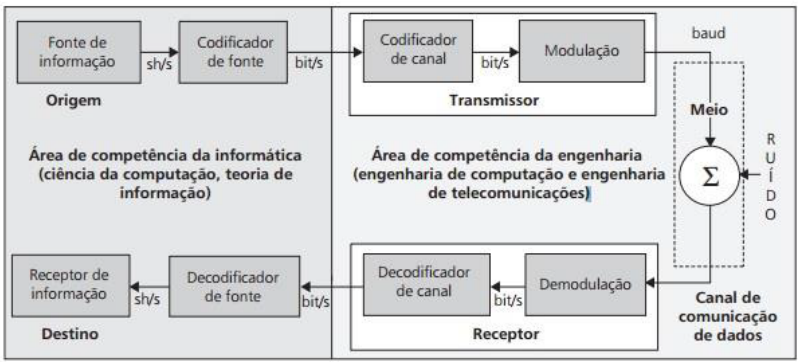


Figura 2.1: Blocos funcionais de um sistema de comunicação de informação genérico

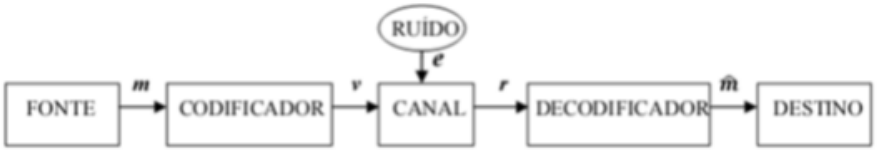


Figura 2.2: Diagrama de como o ruído interfere na transmissão de certa mensagem dependendo de decodificador específico

A mensagem ao passar pelo codificador é transmitida de uma forma onde não só ela está como também uma forma redundante da mensagem é criado na tentativa protegê-la de um possível ruído criado pelo canal de modo independente. Quando o "código" é recebido pelo decodificador, esse tenta com uma palavra recebida, identificar o erro, ou ruído, presente na palavra código a fim de recuperar a mensagem de origem e enviá-la ao destino desejado desde o início.

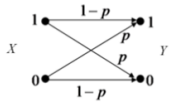


Figura 2.3: BSC com probabilidade p de transição

2.2 Códigos de Hamming

Códigos Hamming são tipos de códigos binários onde a distância mínima d é sempre igual a 3 e o número de linhas é n e colunas $2^{(n-1)}$ onde as colunas são todas as formas distintas de se combinar, em binário, a quantidade de bits n com exceção da coluna nula. Como H não contém o vetor nulo, então d>1, como as colunas são distintas, d>2; como todas as combinações estão presentes na coluna, se combinarmos um coluna com outra, dará uma terceira que também está dentro da matriz, por isso d=3 sempre para Hamming. Essa é uma família de códigos C(2n-1,2n-1-n,3), em que n é maior ou igual a 3, construída por Richard Wesley Hamming (15 de fevereiro de 1915 - 7 de janeiro de 1998) em 1948. Um código de Hamming é um código que permite corrigir um erro ou detectar dois erros.

2.3 Distância de Hamming

Distância de Hamming é o numérico mínimo de substituições necessárias para transformar uma palavra código em outra. Peso de Hamming, denotado pela letra WH(X), corresponde ao numéricos de elementos não nulos de X. Ex: X= [100101], WH(X) = 3. A distância de Hamming entre dois vetores é denotada pelo peso de Hamming da soma desses vetores. Ex:

X= [10110], Y= [01100]; X+Y(em binário) = [11010]; d(X,Y) = 3. A distância mínima de Hamming(dmin) é a menor distância entre dois vetores válidos do código.

2.4 Representação Matricial dos Códigos de Bloco

Se considerarmos os códigos sistemáticos em que: os primeiros k símbolos da palavra de código de n bits compõem a sequência de informação $X = (x_1x_2...x_k)$, e os últimos n - k símbolos representam os bits de verificação ou bits de paridade. $Y = (x_1x_2...x_kc_1c_2...c_{nk}) = (X|C)$ X — Vetor de mensagem C — Vetor de verificação

Cada palavra de código Y é obtida multiplicando o vetor X por uma matriz G, chamada matriz geradora, de dimensões k × n : Y = XG

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & g_{1,k+1} & \cdots & g_{1,n} \\ 0 & 1 & 0 & & 0 & g_{2,k+1} & \cdots & g_{2,n} \\ 0 & 0 & 1 & \cdots & 0 & \vdots & & \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & & \\ 0 & 0 & 0 & \cdots & 1 & g_{k,k+1} & \cdots & g_{k,n} \end{bmatrix} = \left[\mathbf{I}_k \middle| \mathbf{P}\right]$$

$k \times k$ $\quad$ $k \times (n-k)$

$\mathbf{I}k$ — Matriz identidade $k \times k$

$\mathbf{P}$ — Submatriz $k \times (n-k)$

Figura 2.4: Matriz geradora G

Como Y = XG = (X | C) e G = (Ik | P) C = XP. A questão está em determinar a submatriz P para obtermos os valores pretendidos de dmin e Rc. Relacionada com G temos a matriz de verificação de paridade, H, de dimensões n × (n k), tal que GH = 0:

O produto de qualquer palavra de código pela matriz H é um vetor nulo: YH = XGH = 0

2.5 Códigos de Repetição

Em Teoria da Informação, código de repetição é um dos códigos mais básicos de correção de erros, a seu princípio é o de repetir a mensagem pelo bit n vezes, com a finalidade de se

$$\mathbf{H} = \left[\frac{\mathbf{P}}{\mathbf{I}_{n-k}}\right] = \begin{bmatrix} g_{1,k+1} & \cdots & g_{1,n} \\ g_{2,k+1} & \cdots & g_{2,n} \\ \vdots & & \\ g_{k,k+1} & \cdots & g_{k,n} \\ 1 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & 1 \end{bmatrix}$$

Figura 2.5: Matriz de verificação de paridade H

obter uma boa transmissão.

O código de repetição pode corrigir min erros $t = \frac{[d_(min) - 1]}{2} = \frac{[n-1]}{2}$ por palavra de código. Ex: n =3, 001, 010, 100, corresponde ao bit 0, já para 011, 101, 110, o bit 1.

2.6 Capacidade de Detecção e Correção de Erros

A detecção de erros será sempre possível quando o número de erros de transmissão em uma palavra de código seja inferior à distância mínima dmin. No inverso, se o número de erros for igual ou exceder o dmin a palavra errónea pode corresponder a outro vetor válido, fazendo com que os erros não sejam detectados.

• Se um código detecta até l erros por palavra: $d_{min} > l + 1$

• Se um código corrige até t erros por palavra: $d_{min} > 2t + 1$

• Se um código corrige até t erros por palavra e detecta l > t erros por palavra: $d_{min} > t+l+1$

A distância mínima de um código de blocos (n, k) é limitada superiormente por $d_{min} < n-k+1$ (conhecido por Limite de Singleton). Com códigos binários a igualdade só se atinge com códigos de repetição (k = 1). Infelizmente a sua taxa é muito pequena $R_c = \frac{k}{n} = \frac{1}{n}$. Um código com $d_{min} = n - k + 1$ chama-se código separável pela distância máxima ("maximum-distance-separable code", ou código MDS).

2.7 Decodificação por Tabela de Síndrome

Para que haja uma decodificação, passamos por duas etapas, primeiro a detecção de algum erro ao longo do canal, e depois a correção desse erro. A síndrome é um vetor definido por $S = cH_t$ onde c é a palavra código recebida que irá ser decodificada e H é a matriz de paridade de c. Como c = v + e, onde v é a mensagem de origem e e é o erro sistemático do código, então, $S = (v + e)H_t$.Porém, $vH_t = 0$, dai $S = eH_t$.
Esse sistema de equações formado tem solução única se, e somente se, o número de componentes não nulas (peso de e) do vetor e deve ser menor ou igual t. A partir da Síndrome pode-se realizar a detecção do erro. Se $S \neq 0$, pode afirmar com certeza que há um erro presente no código e que a palavra recebida não é do código. Se S = 0, pode-se dizer que a palavra recebida é do código. No entanto, pode haver erro se o vetor e por uma palavra que está contida no código e, mesmo que S = 0, a palavra recebida (v) não é mensagem original c. Nesse caso, diz-se que houve um erro que não pôde ser detectado. Como, nesse caso, o erro seria uma palavra do código, há $2^{(n-1)}$ erros indetectáveis em hamming.

2.8 Códigos de Canal Aplicado a Imagem

2.8.1 Campos de Galois

Definição 1 – Campos são estruturas algébricas formadas por um conjunto de elementos F onde as operações binárias "+" adição e "." multiplicação são definidas. Tais condições devem ser satisfeitas: F é um grupo comutativo sob adição e o elemento identidade é denotado por zero, ou 0. F é um grupo comutativo sob multiplicação e o elemento identidade é denotado por unidade, ou 1. A multiplicação é distributiva sobre a adição, isto é a.(b+c) = a.b + a.c.

Definição 2 – Campos de Galois são campos finitos formados por p elementos, sendo p um número primo, denotados por GF(p). De fato, para qualquer inteiro positivo m, é possível estender GF (p) para um campo de p^m elementos, formando um campo de extensão denotado por $GF(p^m)$. Por razões práticas, o foco deste trabalho será na construção de códigos baseados no campo binário GF(2), ou sua extensão $GF(2^m)$, uma vez que são amplamente

utilizados na transmissão de dados digitais e sistemas de armazenamento por se tratar de uma informação universalmente codificada.

2.8.2 Códigos de Bloco Lineares

A sequência de informação binária é dividida em blocos de comprimento fixo $(m_0, m_1, \ldots, m_{k-1})$ onde cada mensagem consiste em k dígitos de informação. Dessa forma, podem ser formadas 2^k mensagens distintas. O processo de codificação se resume, sob determinadas condições, em transformar cada mensagem em um vetor binário de comprimento n $(c_0, c_1, \ldots, c_{n-1})$ denominado palavra-código. Então, para cada mensagem há um vetor binário correspondente, de forma que o conjunto de palavras-código é denominado Código de Bloco, e será útil apenas no caso das 2^k palavras códigos serem distintas. Um Código de Bloco Linear apresenta menor complexidade de codificação e baseia-se em conceitos da Álgebra Linear e de Campos. A razão $R = k/n$ é denominada Taxa de Código e pode ser interpretada como a quantidade de informação, em bits, sendo codificada por símbolo de transmissão.

Definição 3 - Um Código de Bloco de comprimento n e 2^k palavras-código é denominado Código de Bloco Linear C (n,k) somente se seu conjunto de palavras-código constituem um subespaço k-dimensional de um espaço vetorial V_n de todas n-uplas sobre o campo GF (2).

2.8.3 Codificação

Uma vez que o Código de Bloco Linear C é um subespaço k-dimensional, implica na existência de k vetores linearmente independentes $\overrightarrow{g_0}, \overrightarrow{g_1}, ..., \overrightarrow{g_{k-1}}$ de forma que toda palavra-código $\overrightarrow{c}\ \epsilon$ C seja representada como uma combinação linear desses vetores. Utilizando tal base, forma-se a Matriz Geradora.

$$G = [\overrightarrow{g_0} : \overrightarrow{g_{k-1}}]$$

$$\overrightarrow{c} = \overrightarrow{m}.G(codificao)$$

Definição 4 – um codificador é dito sistemático se podem ser encontrados explicitamente e não alterados na palavra-código. Para uma codificação sistemática de um Código de Bloco Linear C, frequentemente, G é escrita da seguinte forma:

$$G' = [\mathrm{P}\ \mathrm{I}_K] = \begin{bmatrix} p_{0,0} & p_{0,1} & \dots & p_{0,n-k-1} & 1 & 0 & 0 & \dots & 0 \\ p_{1,0} & p_{1,1} & \dots & p_{1,n-k-1} & 0 & 1 & 0 & \dots & 0 \\ p_{2,0} & p_{2,1} & \dots & p_{2,n-k-1} & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ p_{k-1,0} & p_{k-1,1} & \dots & p_{k-1,n-k-1} & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

I_k = Matriz

P – Matriz geradora dos símbolos de paridade k x (n – k)

$$\overrightarrow{c} = \overrightarrow{m}.G = \overrightarrow{m}.[PI_k] = [\overrightarrow{m}P\overrightarrow{m}]$$

A palavra-código divide-se em 2 partes: m referente à mensagem e mP consiste nos símbolos de verificação de paridade. Teorema do Núcleo e da Imagem Sejam U e V espaços vetoriais de dimensão finita sobre um corpo K. Considerando a transformação linear T : U $\rightarrow$ V , então: dim(U) = dim(N (T)) + dim(Im(T)) Ou seja, a soma das dimensões do núcleo e da imagem de T é igual a dimensão do domínio U. Pelo Teorema do Núcleo e da Imagem, uma vez que o Código de Bloco Linear C é um subespaço k-dimensional de um espaço vetorial V_n , deve existir um espaço dual de dimensão (n – k) complementar a C.

Definição 5 - o espaço dual a C (n,k) é o Código Dual de C, denotado por $C^{(n,n-k)}$. Utilizando os vetores da base de C, é formada a Matriz de Verificação de Paridade.

$$H = [\overrightarrow{m} : \overrightarrow{h_{n-k-1}}]$$

De forma que a seguinte condição deve ser satisfeita: $G.H^T = 0$ Uma propriedade importante segue: $c.H^T = 0$ (condição de verificação $\overrightarrow{m}\epsilon C$)

Caso a Matriz Geradora seja sistemática, a Matriz de Verificação de Paridade assume a

seguinte forma: $H = [I_{(n-k)} - P^T]$ Tal condição impõe restrições lineares entre os bits de c chamadas equações de verificação de paridade.

Teorema – Dado um código de bloco linear C e sua matriz de verificação de paridade H. A distância mínima distância $d_{(min)}$ de C é igual ao menor número positivo de colunas de H que são linearmente dependente. Ou seja, todas as combinações de colunas $d_{(min-1)}$ são linearmente independentes. Portanto, há algum conjunto de colunas $d_{(min)}$ que são linearmente dependentes.

2.8.4 Detecção de erros

A passagem por um canal ruidoso não é livre de erros de transmissão, de forma que:

$$\overrightarrow{r} = \overrightarrow{c} + \overrightarrow{e}(vetorrecebido)$$

$$\overrightarrow{e} = [e_1...e_{n-1}], e_i = 1$$

,se houver erro no bit i 0,caso contrário utilizando a condição de verificação citada anteriormente:

$$\overrightarrow{r}.H^T = (\overrightarrow{c} + \overrightarrow{e}).H^T = \overrightarrow{e}.H^T$$

=0 ,se e for palavra do código 0,caso contrário .

O que significa que sendo e igual a uma palavra do código, a palavra $\overrightarrow{r}$ cumpre o teste de verificação de paridade, e os erros de transmissão não são detectados, isto é, há $2^k - 1$ erros indetectáveis.

2.8.5 Decodificação

Definição 8

Capacidade de Detecção:

$$e = d_{min} - 1$$

Capacidade de Correção:

$$t = \frac{d_{min} - 1}{2}$$

Há diferentes formas de corrigir erros e então decodificar $\overrightarrow{r}$. Na seção Implementação tais processos serão praticamente detalhados.

2.8.6 Códigos de Hamming Binários

Os Códigos de Hamming constituem uma família de código lineares (n,k) que são desenhados de forma que satisfazem:

$$k = 2^r - 1 - r$$

$$d_{min} = 3$$

$$Onder = n - k$$

$$Sendoassim$$

$$e = 2$$

$$t = 1$$

3 Metodologia

3.1 Análise de desempenho para códigos de canais

Através da linguagem de programação Python desenvolvemos esta primeira parte do projeto .Tendo como parâmetros a probabilidade de ocorrer um erro em um determinado bit de uma mensagem (p) e a relação Sinal Ruído (SNR) de um canal BSC, com isso geramos gráficos e comparamos os diferentes tipos de códigos. Dessa forma, montamos um algoritmo com o qual, de posse de uma mensagem binária conseguimos codifica-la, simular um envio através de um canal com um ruído, corrigir a interferência produzida pelo canal e decodificá-la para um receptor. Segue o detalhamento de cada um desses passos

3.1.1 Codificação

Para cada código cria-se suas respectivas matrizes G (matriz geradora) e H (matriz de paridades). Com isso, através da multiplicação de uma mensagem m pela matriz geradora G obtém-se tal mensagem em sua forma codificada v.

3.1.2 Simulação de canal

Para simular um envio do código através de um canal gerou-se um vetor de erro e com probabilidade p de erro por bit calculada pela função Q(x) representada por Eq. 1 onde x é calculado por Eq. 2:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} e \frac{-t^2}{2} \, dx$$

$$x = \sqrt{\frac{WRE_b}{N_0}}$$

Onde E_b/N_0 representa a relação sinal ruído do canal (SNR) e R é igual a razão k/n, que é a taxa de informação utilizada na transmissão. Com isso somou-se esse vetor e ao vetor v resultando assim, no vetor r que representa o código com o erro simulado.

3.1.3 Correção do erro

A partir de r achamos a sua síndrome através da multiplicação de r por Ht (matriz de paridades transposta), e detectamos o possível erro. Corrigimos então v fazendo r - e', onde e' é o possível erro encontrado através da comparação da síndrome com a matriz Ht.

3.1.4 Decodificação

Como as matrizes H e G são criadas em sua forma sistemática, onde a matriz identidade encontrada em ambas (à esquerda em H e à direita em G), basta separar os últimos k dígitos de v que já obteremos a mensagem m inicialmente enviada.

3.1.5 Códigos de Canal Aplicado a Imagem

Aqui iremos explicar as principais funções que implementamos no código. As próprias instruções passadas pelo professor guiaram boa parte do raciocínio e dos algoritmos utilizados pelo grupo na elaboração do projeto. Porém, algumas funções que fazem parte do código vieram das bibliotecas abertas numpy, PIL e random.

- bits(n): No padrão RGB (red, blue, green), cada cor é representada por um número entre 0 e 255, em que o zero significa a intensidade mínima (ou ausência) desta cor e o 255, sua intensidade máxima. Desta forma, por exemplo o preto, que é a ausência de cores, é representado pelo vetor [0 0 0] e o branco, que é a mistura das três cores, é representado por [255 255 255]. Desta forma, a função Bits é responsável por receber

um número n [0, 255] e o transforma-lo num vetor de 8 bits. Desta forma o vetor representa n em binário.

- altura, largura e tamanho: como o próprio nome sugere, cada função desta é responsável por receber um arquivo de imagem e retornar suas dimensões em pixel.

- split(vetor,k): Esta função recebe um vetor de tamanho n e o divide em vetores menores, de tamanho k. Em seguida monta uma matriz (n/k, k) onde cada linha é uma fração do vetor recebido como parâmetro.

- *convert_cor*(splited): Recebe um vetor binário já processado pela função split e o transforma num vetor de números que representam cores. Ela pode ser entendida como a função inversa de bits(n).

- *compacta_cor*(vetor): Recebe um vetor binário, e, utilizando as funções split e *compacta_cor*, retorna uma matriz em que suas linhas representam as informações de cor de cada pixel, que futuramente formará uma imagem.

- img2vecbin(img): Recebe um arquivo de imagem e, através dos recursos open e getdata da biblioteca PIL, extrai informações de cor de cada pixel da imagem e os organiza num único vetor binário. Em seguida, ela retorna este vetor.

- vecbin2img(vecbin,M,N): Usando as funções *compacta_cor* justamente com recursos da biblioteca numpy, ela funciona como o inverso da função img2vecbin. Ou seja, recebe um vetor binário com as informações de cor e dois números, M e N. Em seguida, ela monta uma imagem de dimensões MxN de acordo com o vetor recebido.

- noise(N,p): É a função que gera um vetor binário de tamanho N e com probabilidade p de cada bit ter valor 1. É importante na geração de vetores de erro, que serão usados para simulação de ruído em imagens. A implementação é baseada na inserção de um elemento por vez, com sua probabilidade definida por ferramentas da biblioteca random.

- $simular\_ruido(p, vetor\_imagem)$: Simula um ruído através da mudança de bits de

um vetor de imagem ao somar a modulo 2 o vetor da imagem com um vetor de erro gerado pela função noise. Após essa mudança, a função retorna o vetor alterado.

- codificador(msg): É a implementação do código de Hamming (15, 11, 3). A função recebe apenas o parâmetro "mensagem", e retorna a mesma mensagem codificada pelo código de Hamming. A implementação é feita dividindo o vetor mensagem em vários vetores de tamanho 11 usando a função split, e após isso cada vetor de 11 bits é codificado e adicionado a outro vetor. A codificação é baseada em adicionar um bit de paridade em cada posição potência de 2.

- mensagem tem 11 bits, foram necessários 4 bits de paridade, nomeados p1, p2, p4 e p8. Esses coeficientes foram calculados da seguinte forma:

- p1 analisa um bit e pula um bit, p2 analisa 2 bits e pula 2 bits, p4 analisa 4 bits e pula 4 bits, e p8 analisa 8 bits e pula 8 bits. As análises começam a partir do próprio bit de paridade, e os coeficientes são determinados somando a módulo 2 os bits analisados, com exceção do bit de paridade que está sendo calculado.

- Determinados os bits de paridade de cada subdivisão do vetor, eles são adicionados em suas respectivas posições em cada subvetor de 11 bits. Após a codificação, todos os bits são lidos por laços for e adicionados a um novo vetor único, nomeado de *palavra_codigo*, e retornado pela função.

- decodificar(*palavra_codigo*): É a função responsável por fazer o caminho inverso da função codificador. Recebendo o parâmetro "*palavra_codigo*", retorna a mensagem decodificada. A implementação foi feita de forma análoga ao procedimento feito pela codificação, pois calculava os bits de paridade esperados da mesma forma que foram calculados p1, p2, p4 e p8 na função codificação, mas nessa função, os valores são adicionados às novas variáveis pv1, pv2, pv4 e pv8. Antes de fazer isso, foram armazenados nas novas variáveis p1, p2, p4, p8 os valores referentes às posições 0, 1, 3, 7, respectivamente, para serem usados depois. Calculados os valores de pv1, pv2, pv4, pv8, era feita uma verificação de erro da seguinte forma:

- A variável erro era pré-estabelecida com o valor -1. Eram comparados os valores de p com pv, e se não fossem coincidentes, era adicionado à variável erro o valor do coeficiente de p. Por exemplo, se p1 for diferente de pv1, a variável erro é incrementada de 1. Ou se p2 for diferente de pv2, erro é incrementado de 2. Com os casos verificados, se o erro for maior ou igual a zero, o coeficiente na posição referente ao valor de erro é alterada, ou seja, se erro for igual a 2, o bit na posição 2 do vetor é alterado.

- Para a obtenção da mensagem após a correção de erro, são adicionados a um novo vetor "mensagem" os bits de palavra código com exceção dos bits de paridade. E a mensagem é retornada pela função, já decodificada.

- contar(vetor): Para codificar um vetor usando o código de Hamming, é importante que o vetor tenha um valor de tamanho múltiplo de 11. Mas nem sempre isso acontece. Devido a isso, essa função foi implementada para verificar se resto da divisão do tamanho do vetor por 11 era igual a zero. Caso não seja, são adicionados bits com valor zero no final do vetor até que o meso14 possa ser codificado pelo código de hamming. Para contar o número de zeros adicionados, foi usada uma variável contador, inicialmente com valor zero, que era incrementada de um a cada vez que era adicionado um bit zero no vetor. Por fim, os bits adicionados são removidos com a função pop, e a função retorna a variável contador.

- $codificar\_imagem$(vetor): Na função contar, foi visto que para que seja possível codificar com o código de Hamming (15, 11, 3), é necessário adicionar bits até que o tamanho do vetor seja múltiplo de 11. Isso também é feito nessa função, da mesma forma que foi implementado na função contar. Depois disso feito, o vetor foi subdividido em vetores de tamanho 11 com a função split, e os mesmos foram codificados pela função codificador e adicionados em outro vetor. O solicitado era obter um vetor único codificado, então com dois laços for, foram percorridos todos os vetores codificados e os valores foram adicionados em um novo vetor, declarado como $vetor\_definitivo$, o qual seria retornado pela função.

- $decodificar\_imagem$(vetor, contador): Faz o inverso do função $codificar\_imagem$.

Recebendo os parâmetros vetor e contador, é feita uma divisão no vetor em vetores menores de tamanho 15, após isso, é usado um laço for para decodificar cada subvetor, e armazená-lo em um novo vetor. Com os vetores decodificados, cada número de cada vetor foi adicionado a um novo vetor único, para que toda a mensagem ficasse em um único vetor. Por último, foi usada a função pop do número de vezes do contador para remover os últimos bits que foram adicionados para ser possível a codificação. Após isso, o vetor de imagem é retornado.

- transmissao(enviada ,so_imagem, $imagem\_hamming$): Mostra a taxa de transmissão de informação através de um simples código. É feita uma comparação entre os vetores enviada e so_imagem, contando o número de bits diferentes entre os dois vetores. A mesma coisa é feita comparando os vetores enviada e so_imagem, obtendo-se também o número de bits diferentes entre esses últimos dois vetores. Feito isso, a taxa de transferência de energia é calculada subtraindo o tamanho da mensagem pelo número de erros, e depois dividindo-se pelo tamanho de mensagem. Foi feito isso para a imagem sem ser codificada antes de aplicar o ruído e com a imagem codificada pelo código de Hamming, e foram retornados os dois valores.

4 Análise

4.1 Análise de desempenho para códigos de canais

De acordo com a teoria da codificação do código de Hamming, o código pode ter até um bit corrigido. Como o código de Hamming corrige no máximo um erro por código, quanto menor a probabilidade de erro, maior a probabilidade da correção do mesmo, e quanto maior a probabilidade do erro, menor a probabilidade de correção. Foram realizados testes com diversas probabilidades para analisar as imagens e as taxas de transferência obtidas. Aplicando valores baixos de probabilidade de erro, espera-se que a mensagem codificada possua maior transferência de energia que a mensagem não codificada. Como pedido nas orientações do projeto, foi feito um primeiro teste aplicando um ruído com valor p de 0.0001, ou seja, 0.01% de chance de ocorrer um erro em um bit da imagem. Não foi possível ver diferença na imagem, uma vez que esta probabilidade é relativamente baixa, sendo, os erros que aconteceram, provavelmente corrigidos. Mas, observando a taxa de transferência de energia, o código de Hamming teve aproximadamente 1,0 de taxa de transferência, enquanto a taxa de transferência de uma imagem com apenas um ruído aplicado teve aproximadamente 0.99989605.

É interessante mencionar que nesse caso, a probabilidade de erro era tão baixa que o código de Hamming pode corrigir todos os erros visíveis por um limite de 16 casas decimais. O aumento na taxa gerado pela codificação foi de 0,00010395. Aumentando o valor de p para 0.001, já é possível enxergar minúsculos ruídos nos pixels da imagem que não foi codificada. Observando os valores das taxas de transmissão de energia, a imagem codificada teve esse valor aproximado de 0.99997838, enquanto a imagem não codificada teve o valor

de 0.99902137. O aumento da taxa de transmissão foi de 0,00095701. Observando 0.01, os ruídos ficam muito mais visíveis na imagem que não foi codificada. As taxas obtidas foram 0.99006446 para a imagem não codificada e 0.99806769 para a imagem codificada, mostrando um aumento na taxa de 0.00800323.

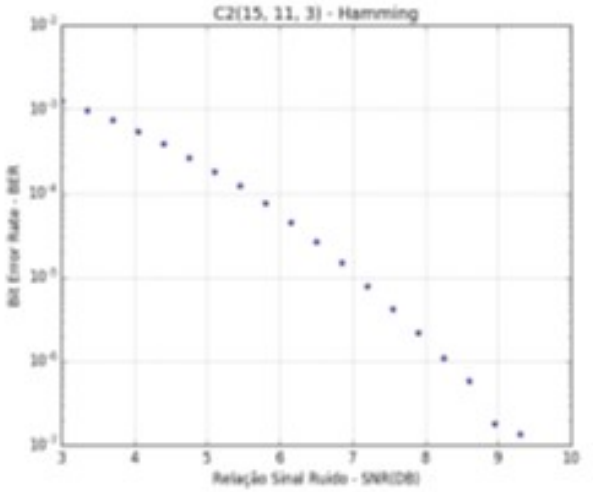
Foi feito um teste com uma probabilidade p de 0.05, as duas imagens estavam com muito ruído, mas foi possível observar uma pequena diferença de ruído entre as duas, onde a codificada ainda estava com mais eficiência. as taxas de transferência foram 0.94995168 para a imagem não codificada e 0.96334746 para a imagem codificada, mostrando ainda a vantagem de usar o código de Hamming através do ganho de 0,01339578. O código de Hamming realmente estava melhorando a eficiência da transmissão de informação. Foi visto que a taxa de transferência de informação retornado pela função ainda foi superior pelo código de Hamming, mostrando até aqui ser eficiente na transmissão de dados. 16 No código implementado, cada mensagem possuía 11 bits.

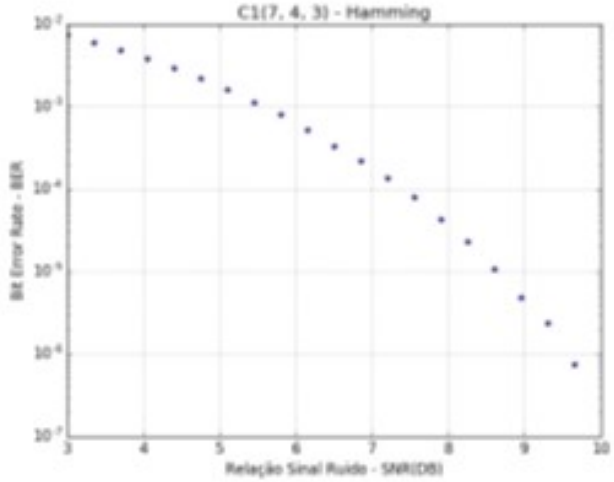
Sabendo que só é possível corrigir um bit por código transferido, a probabilidade máxima da ocorrência de erros que faz com que o código de Hamming seja eficiente é 1/11, ou seja, a probabilidade máxima teórica de erros que vai valer a pena usar o código de Hamming é 9.09%. Para verificar se na prática isso era satisfeito, foi atribuído um valor p de 1/11 e foram observados os resultados. Visivelmente, as imagens possuem ruídos iguais em p igual a 1/11. Observando os valores para taxa de transferência de energia, foram obtidos valores muito aproximados de 0.90900000 para a imagem codificada e não codificada. Nesse caso, podemos ver que não há ganho de transmissão de energia, nem perda, é apenas um p "crítico".

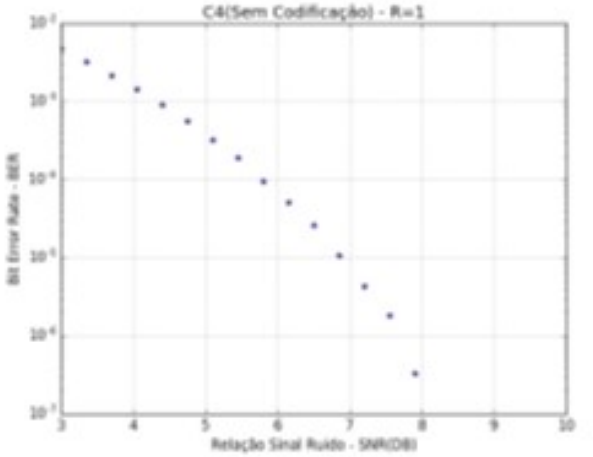
Como foi visto, altas probabilidades de erro geram baixa eficiência do código de Hamming, daí espera-se que a taxa de transmissão de dados na mensagem codificada com um ruído aplicado não seja muito melhor que a taxa de transmissão de dados numa mensagem sem codificação e com ruído. Foi feito um teste com p igual a 0.15, e foram obtidos valores 0.85019409 para a imagem não codificada, e 0.82989054 para a imagem codificada, mostrando que houve uma perda de 0.02030355 de taxa de transmissão de informação. Devido a isso, o código de Hamming não é eficiente para valores altos de probabilidade de ruído.

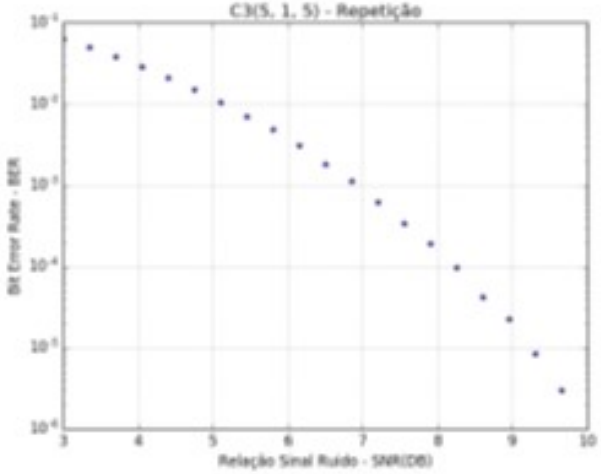
4.2 Códigos de Canal Aplicado a Imagem

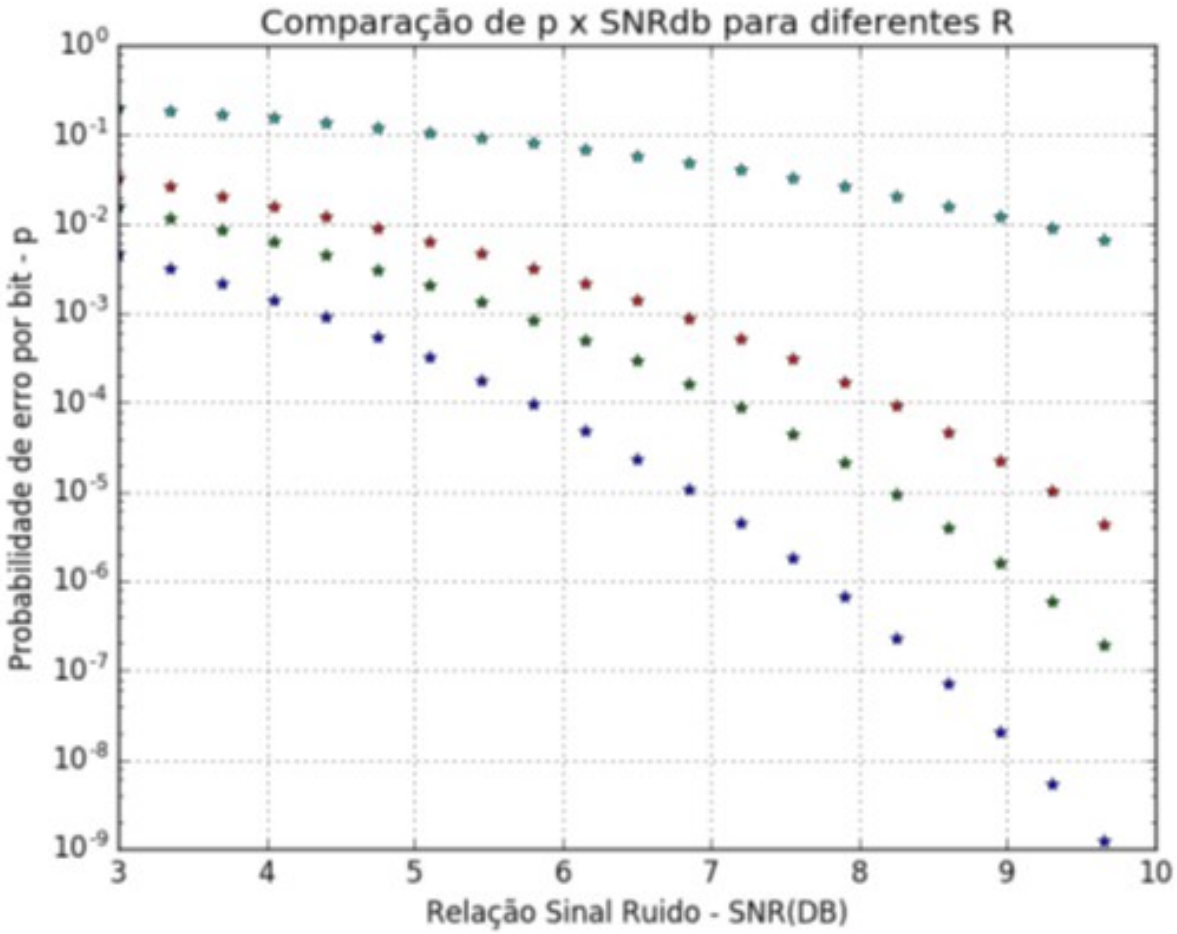
Observando os gráficos abaixo podemos concluir que nossas previsões foram comprovadas, pois se aumentarmos R, como podemos ver na Eq. 2, o valor de x também aumentará, fazendo com que o valor de p caia como podemos ver na Eq. 1. Dessa forma, chegamos à conclusão que, entre esses códigos, por mais inusitado que seja, o código com R = 1, ou seja, sem codificação foi o que se mostrou mais eficiente dentre os analisados.










Comparação de p x SNRdb para diferentes R
Probabilidade de erro por bit - p
Relação Sinal Ruido - SNR(DB)
10^0
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^-8
10^-9
3
4
5
6
7
8
9
10

5 Conclusão

O projeto foi importante para sabermos a utilização dos assuntos estudados, como código de hamming, síndrome, distância de hamming, etc, na prática pois não havia possibilidade de realização das atividade básicas e específicas solicitadas sem a teoria de códigos, necessária na implementação do código fonte, feito em linguagem Phyton. O código binário é um tipo de código onde é introduzido uma mensagem de dois bits de entrada (0 ou 1) e, no decorrer da transmissão, existe uma probabilidade p, da saída ser alterada(onde é 1, ser 0, e onde é 0, ser 1). O projeto teve o objetivo de analisar o melhor código binário para determinados "r" escolhidos. Era esperado que se não houvesse nenhuma codificação, R=1, o código teria o melhor desempenho. Na análise pudemos comprovar que o esperado realmente aconteceu, e na medida que R aumenta, a eficiência do código se torna menor.

Códigos são muito importantes na transmissão de dados, pois quando os dados enviados são modificados por ruídos, é possível que haja a correção de alguns desses erros. Neste projeto, foi implementado o código de Hamming (15,11,3), que corrigia no máximo um erro por mensagem. Através de testes, foi possível ver que a eficiência desse código só era válida para pequenos valores de probabilidades da ocorrência de erros, pois o máximo que ele corrigia era um bit a cada mensagem de 11 bits.

6 Referências Bibliográficas

- LIN, Shu; J. COSTELLO, Daniel. Error Control Codingn: Fundamentals and Applications. Prentice-Hall, 1983
- K. MOON, Todd. Error Correction Code
- Campello, Ricardo, apostila de sistemas discretos.
- Notas de aula da disciplina

7 Anexos

7.1 Códigos em Python

```
1
2  import math
3  import sys
4  import random
5  import random as rand
6  import scipy as spy
7  import numpy as np
8  import matplotlib.pyplot as plt
9  import scipy
10 import scipy.interpolate as ip
11 from scipy import special
12 def codificarc1(msg, n):
13     G = [[1, 1, 0, 1, 0, 0, 0], [1, 0, 1, 0, 1, 0, 0], [0, 1, 1, 0,
           0, 1, 0], [1, 1, 1, 0, 0, 0, 1]]
14     cod = np.dot(msg, G)
15     for i in range(0, n):
16         cod[i] = cod[i]%2
17         return cod
18 def decodificadorc1(cod, k, n, Ht, S):
19     teste = 0
20     vetor = np.zeros((n), dtype=np.int)
```

```
21      if (S != [0, 0, 0]).all() == 1:
22          for i in range(0, n):
23              if S[0] == Ht[i][0]:
24                  if S[1] == Ht[i][1]:
25                      if S[2] == Ht[i][2]:
26                          pos = i
27                          teste = 1
28          vetor[pos] = (vetor[pos] + 1)%2
29      final = [vetor[n-4], vetor[n-3], vetor[n-2], vetor[n-1]]
30      return final
31  def codificarc2(msg, n):
32      G = [[1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 1, 0,
           0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
33          [1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 1,
              0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
34          [0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 1, 1, 0,
              0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
35          [1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0], [1, 0, 1, 1,
              0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
36          [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], [1, 1, 1, 1,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
37          [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
38      cod = np.dot(msg, G)
39      for i in range(0, n):
40          cod[i] = cod[i]%2
41          return cod
42  def decodificadorc2(cod, k, n, Ht, S):
43      teste = 0
44      vetor = np.zeros((n), dtype=np.int)
45      if (S != [0, 0, 0, 0]).all() == 1:
46          for i in range(0, n):
47              if S[0] == Ht[i][0]:
```

```
48                  if S[1] == Ht[i][1]:
49                      if S[2] == Ht[i][2]:
50                          if S[3] == Ht[i][3]:
51                              pos = i
52                              teste = 1
53          vetor[pos] = (vetor[pos] + 1)%2
54      final = [vetor[n-11] ,vetor[n-10] ,vetor[n-9] ,vetor[n-8] ,vetor[
           n-7] ,vetor[n-6] ,vetor[n-5] ,vetor[n-4],
55              vetor[n-3], vetor[n-2], vetor[n-1]]
56      return final
57  def codificarc3(msg, n):
58      cod = np.zeros(n, dtype=np.int)
59      G = [1, 1, 1, 1, 1]
60      for i in range(0, n):
61          cod[i] = msg[0]*G[i]
62      for i in range(0, n):
63          cod[i] = cod[i]%2
64      return cod
65  def decodificadorc3(cod, n):
66      cont_zeros = 0
67      cont_uns = 0
68      for i in cod:
69          if i == 0:
70              cont_zeros += 1
71          else:
72              cont_uns += 1
73      if cont_zeros > cont_uns:
74          for i in range(0, n):
75              cod[i] = 0
76      else:
77          for i in range(0, n):
78              cod[i] = 1
```

```
79      return cod
80  def noise(n, p):
81      erro = np.random.binomial(1, p, n)
82      return erro
83  def main():
84      SNRdb = np.arange(3, 10, 0.35)
85      SNR = np.zeros(len(SNRdb))
86      for y in range(0, 20):
87          SNR[y] = math.pow(10, SNRdb[y]/10)
88      print("Entre com o codigo a ser analizado: ")
89      print("(1) C1(7,4,3) - Hamming")
90      print("(2) C2(15,11,3) - Hamming")
91      print("(3) C3(5,1,5) - Repeti  o")
92      print("(4) C4(Sem Codifica  o) - R=1")
93      print("Entre com o numero referente    opcao: ", end='')
94      op = int(input())
95      if op == 1:
96          k = 4
97          n = 7
98          msg1 = np.zeros((k), dtype=np.int)
99          cod = codificarc1(msg1, n)
100         BER = np.zeros(20)
101         for i in range(0, 20):
102             Ht = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0,
                    1], [0, 1, 1], [1, 1, 1]]
103             E =0
104             M = 0
105             p = scipy.special.erfc(math.sqrt((2*k*SNR[i])/n))
106             for b in range(0, 2000000):
107                 erro = noise(n, p)
108                 for j in range(0, n):
109                     cod[j] = 0
```

```
110                     cod[j] = (cod[j] + erro[j])%2
111                 S = np.dot(cod, Ht)
112                 for z in range(0, n-k):
113                     S[z] = S[z]%2
114                 final = decodificadorc1(cod, k, n, Ht, S)
115                 if (msg1 == final).all() == 1:
116                     M = M + 1
117                 else:
118                     E = E + sum(final)
119                     M = M + 1
120             BER[i] = E/(k*M)
121         plt.semilogy(SNRdb, BER, "*")
122         plt.title('C1(7, 4, 3) - Hamming')
123         plt.ylabel('Bit Error Rate - BER')
124         plt.xlabel('Rela   o Sinal Ruido - SNR(DB)')
125         plt.grid(True)
126         plt.show()
127     elif op == 2:
128         k = 11
129         n = 15
130         msg2 = np.zeros((k), dtype=np.int)
131         cod = codificarc2(msg2, n)
132         BER = np.zeros(20)
133         for i in range(0, 20):
134             Ht = [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0,
                      1], [1, 1, 0, 0], [1, 0, 1, 0],
135             [1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1], [1, 1, 1, 0],
                    [1, 1, 0, 1], [1, 0, 1, 1],
136             [0, 1, 1, 1], [1, 1, 1, 1], [0, 1, 1, 0]]
137             E = 0
138             M = 0
139             for b in range(0, 2000000):
```

```
140                 p = scipy.special.erfc(math.sqrt((2*k*SNR[i])/n))
141                 erro = noise(n, p)
142                 for j in range(0, n):
143                     cod[j] = 0
144                     cod[j] = (cod[j] + erro[j])%2
145                 S = np.dot(cod, Ht)
146                 for z in range(0, n-k):
147                     S[z] = S[z]%2
148                 final = decodificadorc2(cod, k, n, Ht, S)
149                 if (msg2 == final).all() == 1:
150                     M = M + 1
151                 else:
152                     E = E + sum(final)
153                     M = M + 1
154             BER[i] = E/(k*M)
155         plt.semilogy(SNRdb, BER, "*")
156         plt.title('C2(15, 11, 3) - Hamming')
157         plt.ylabel('Bit Error Rate - BER')
158         plt.xlabel('Rela  o Sinal Ruido - SNR(DB)')
159         plt.grid(True)
160         plt.show()
161     elif op == 3:
162         k = 1
163         n = 5
164         msg3 = [0]
165         cod = codificarc3(msg3, n)
166         BER = []
167         for i in range(0, 20):
168             E = 0
169             M = 0
170             for b in range(0, 2000000):
171                 p = scipy.special.erfc(math.sqrt((2*k*SNR[i])/n))
```

```
172                 erro = noise(n, p)
173                 for j in range(0, n):
174                     cod[j] = 0
175                     cod[j] = (cod[j] + erro[j])%2
176                 final = decodificadorc3(cod, n)
177                 if(msg3 == final).all() == 1:
178                     M = M + 1
179                 else:
180                     E = E + 1
181                     M = M + 1
182             BER.append(E/(k*M))
183         plt.semilogy(SNRdb, BER, "*")
184         plt.title('C3(5, 1, 5) - Repeti  o')
185         plt.ylabel('Bit Error Rate - BER')
186         plt.xlabel('Rela  o Sinal Ruido - SNR(DB)')
187         plt.grid(True)
188         plt.show()
189     elif op == 4:
190         BER = []
191         k = 3
192         n = 3
193         msg4 = np.zeros((k), dtype=np.int)
194         for i in range(0, 20):
195             E = 0
196             M = 0
197             for b in range(0, 2000000):
198                 p = scipy.special.erfc(math.sqrt((2*SNR[i])))
199                 erro = noise(n, p)
200                 for j in range(0, 3):
201                     msg4[j] = 0
202                     msg4[j] = (msg4[j] + erro[j])%2
203                 if sum(msg4) == 0:
```

```
204                     M = M + 1
205                 else:
206                     E = E + sum(msg4)
207                     M = M + 1
208             BER.append(E/(k*M))
209         plt.semilogy(SNRdb, BER, "*")
210         plt.title('C4(Sem Codifica  o) - R=1')
211         plt.ylabel('Bit Error Rate - BER')
212         plt.xlabel('Rela  o Sinal Ruido - SNR(DB)')
213         plt.grid(True)
214         plt.show()
215 main()
```

Code Listing 7.1: Python example

7.2 Resultados dos códigos em Python

Figura 7.1: p = 0.0001

Figura 7.2: p = 0.001

Figura 7.3: p = 0.01

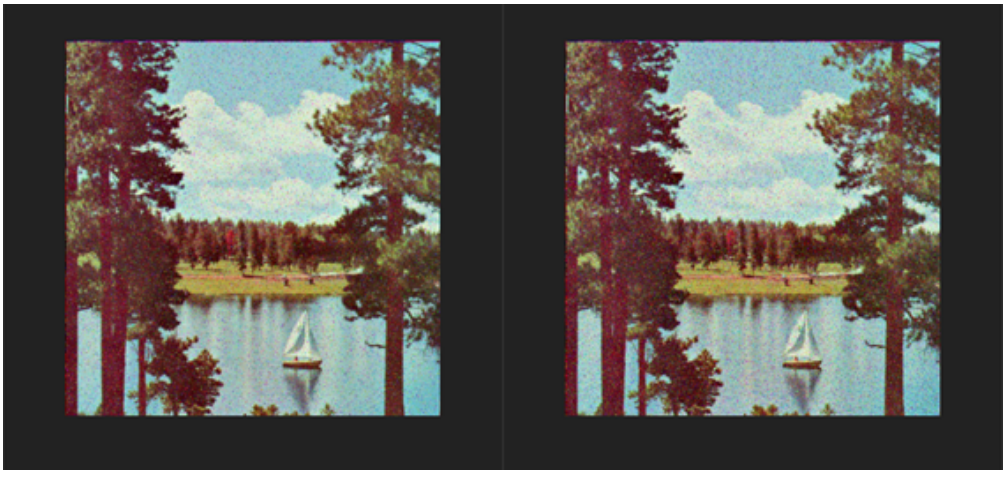


Figura 7.4: p = 0.05

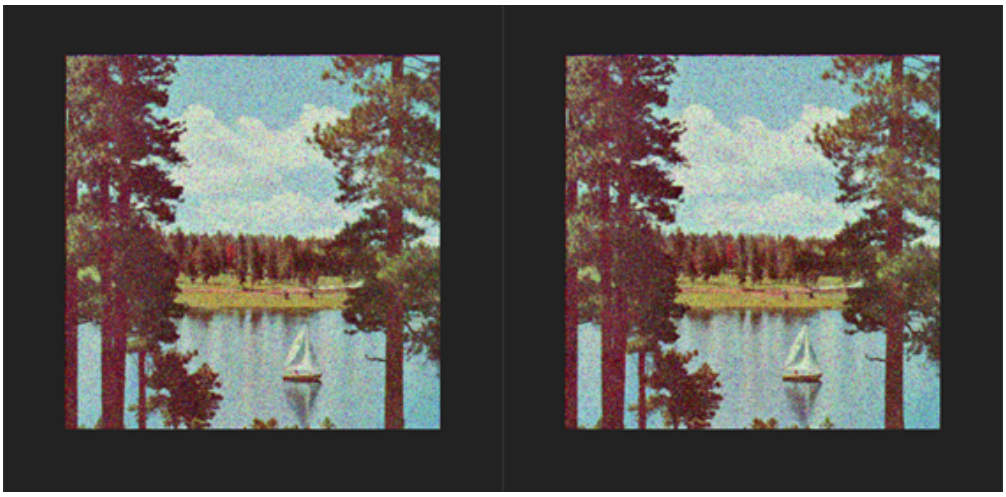


Figura 7.5: p = 1/11

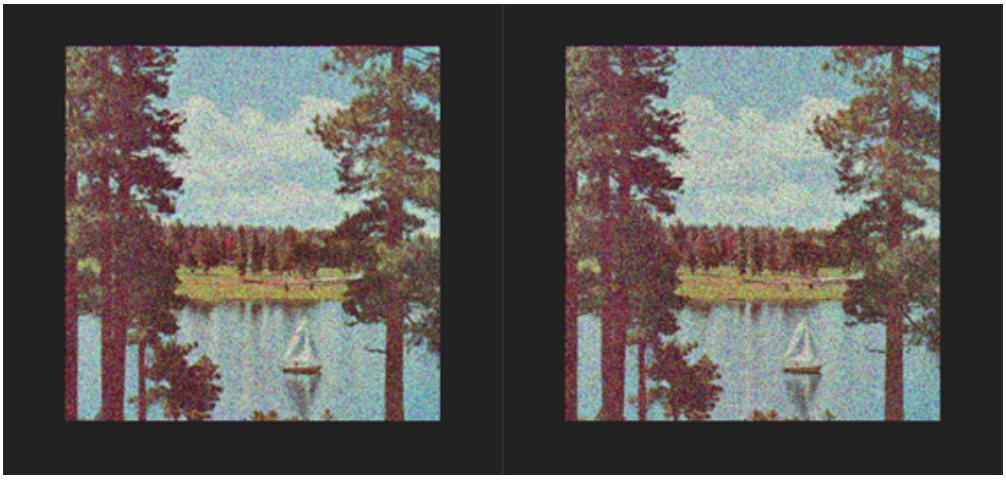


Figura 7.6: p = 0.15