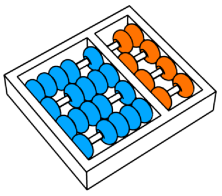

Universidade Estadual de Campinas
Instituto de Computação

Tarefa 1 - Introdução ao Processamento Digital de Imagem (MO443)

Esteganografia

André Igor Nóbrega da Silva (RA:203758)

Docente:
Prof. Dr. Hélio Pedrini

Campinas
Março, 2022

1 Introdução

A Esteganografia é uma técnica milenar cujo objetivo é esconder informação em meios conhecidos apenas pelo remetente e pelo destinatário da mensagem. A própria palavra deriva do grego, em que *estegano* = esconder e *grafia* = escrita (ROCHA; COSTA; CHAVES,). Uma das utilizações iniciais da técnica foi no século V, na Pérsia. O grego Histaeus raspou a cabeça de um escravo, escreveu uma mensagem e esperou o cabelo crescer novamente para enviá-lo à Aristagolas de Mileto. Aristagolas, então, raspou a cabeça do escravo e recebeu a mensagem que incitava uma revolta contra os persas (ROCHA; COSTA; CHAVES,).

Neste trabalho, implementa-se uma técnica comum de esteganografia aplicada a imagens digitais. A ideia é a de modificar os 3 planos de bits menos significativos da imagem de modo a inserir uma mensagem oculta, seja um texto ou uma imagem. As imagens trabalhadas possuem 3 bandas referentes aos canais de cores RGB. Dessa forma, para uma imagem com w colunas e h linhas, em que cada pixel é formado por 8 bits, é possível guardar até:

$$Tamanho_{mensagem} = w \times h \times n\_planos\_bits \times n\_bandas = w \times h \times 3 \times 3$$

bits de informação.

Este documento é dividido da seguinte maneira: esta primeira seção apresenta uma breve introdução ao tema e ao trabalho desenvolvido; a seção 2 mostra as especificidades dos programas desenvolvidos em Python para codificação e decodificação; a seção 3 discorre sobre os detalhes da solução desenvolvida; a seção 4 apresenta os resultados de experimentos propostos; a seção 5 aponta as limitações dos programas e a seção 6 apresenta as conclusões.

2 O Programa

Este trabalho consistiu no desenvolvimento de 2 programas: codificar.py e decodificar.py. Ambos foram implementados no Python 3.9.10 e utilizam as bibliotecas: *numpy* 1.21.2, *imageio* 2.9.0.

2.1 Estrutura de Pastas

O projeto foi dividido em 3 pastas: *src*, *sample_images*, *sample_text* e *outputs*. A primeira contém os dois programas da especificação, bem como um arquivo *utils.py*, com funções auxiliares. A segunda possui exemplos de imagens coloridas com tamanhos variados obtidas do link http://www.ic.unicamp.br/ helio/imagens_coloridas/. A terceira possui exemplos de texto de tamanho variado. A última é o diretório padrão para guardar as imagens e textos resultantes dos programas *codificar.py* e *decodificar.py*, respectivamente.

2.2 Codificar.py

O programa *codificar.py* possui os seguintes parâmetros de entrada:

- Imagem RGB de entrada no formato .png
- Texto a ser codificado no formato .txt
- Plano de bits no qual codificar o texto
- Imagem de saída no formato .png

De acordo com a estrutura de pastas do projeto, um exemplo de execução seria:

python3 codificar.py ../sample_images/watch.png ../sample_texts/sample1.txt 0 coded_image.png

Nesse caso, a saída seria a imagem **coded_image.png**, dentro do diretório *outputs*. É importante destacar que é necessário executar o programa de dentro da pasta *src*.

2.3 Decodificar.py

O programa *decodificar.py* possui os seguintes parâmetros de entrada:

- Imagem RGB com texto codificado de entrada no formato .png
- Plano de bits no qual a mensagem se encontra
- Texto de saída no formato .txt

De acordo com a estrutura de pastas do projeto, um exemplo de execução seria:

python3 decodificar.py ../outputs/coded_image.png 0 decoded_text.txt

Nesse caso, é necessário que já exista a imagem com a mensagem secreta *coded_image.png*. O programa salvará o texto resultante no diretório *outputs* com o nome **decoded_text.txt**. É importante destacar que é necessário executar o programa de dentro da pasta *src*.

3 Solução Desenvolvida

3.1 Leitura de dados

Como os programas lidam tanto com imagem quanto com texto, é necessário realizar leitura e preparação desses tipos de dados.

As imagens de entrada, independentemente de conterem texto codificado, são lidas por meio da função *imread* da biblioteca *imageio*. Seu tipo é, portanto, *iio.core.util.Array* e possuem uma representação matricial no formato $height, width, bands$. Os valores dos pixels variam no conjunto $[0, 1, ..., 255]$.

Os textos de entrada, por sua vez, são arquivos no formato .txt e podem ser abertos e lidos com funções tradicionais do Python, como a *readlines*. Imediatamente após a leitura,

transforma-se todo o texto em uma única *string* e adiciona-se o caractere "\0" representando o final do texto (do inglês, *end of file* (EOF)). Em seguida, utiliza-se o código ASCII para transformar essa *string* em uma sequência binária, em que cada caractere representa 8 bits. Esses bits são todos concatenados em um *numpy array* e estão prontos para serem inseridos na imagem.

O trecho de código a seguir mostra como esse processo foi implementado:

```
import numpy as np

def text2bits(text, encoding='utf-8', errors='surrogatepass'):
    """
    Converts a given text (string) to its binary representation in the ASCII code.
    Reference: https://stackoverflow.com/questions/7396849/convert-binary-to-ascii-and-vice-versa
    """
    bits = bin(int.from_bytes(text.encode(encoding, errors), 'big'))[2:]
    return bits.zfill(8 * ((len(bits) + 7) // 8))

def read_input_txt(input_txt):
    with open (input_txt, "r") as myfile:
        data=myfile.readlines()

    data = ''.join(data)
    data = data + '\0' # adding EOF signal
    bits_secret_message = text2bits(data)
    arr_secret_message = np.array([int(char) for char in bits_secret_message], dtype = np.uint8)

    return arr_secret_message
```

Listing 1: Leitura e transformação de um arquivo .txt.

3.2 Codificação de Texto em Imagem

A codificação implementada consiste em substituir os bits menos significativos por uma sequência de valores que representam os bits do código ASCII de um texto de entrada.

Primeiramente foi necessário acessar os diversos planos de bits das imagens de entrada. Para isso, utilizou-se as funções *packbits* e *unpackbits* da biblioteca *numpy*. A função *unpackbits* desempacota os bits de um array numpy do tipo uint8 em um novo array com valores binários. Um exemplo de utilização dessas funções é dado a seguir:

```
>>> a = np.array([2], dtype = np.uint8)
>>> np.unpackbits(a)
array([0, 0, 0, 0, 0, 0, 1, 0], dtype=uint8)
>>> b = np.unpackbits(a)
>>> b
array([0, 0, 0, 0, 0, 0, 1, 0], dtype=uint8)
>>> np.packbits(b)
```

```
array([2], dtype=uint8)
```

Listing 2: exemplo de utilização do *unpackbits* e *packbits*

Como cada imagem possui 3 dimensões, é preciso definir um eixo (*axis*) no qual os bits serão desempacotados. Nesse caso, escolheu-se desempacotar os bits ao longo das colunas. A Figura 1 mostra um exemplo de imagem com 3 canais de cores (RGB), 3 linhas e 3 colunas. Ao utilizar a função *unpackbits* ao longo das colunas, as dimensões H e B se mantém, enquanto W é multiplicado por 8. Os primeiros 8 valores da primeira linha e da primeira banda agora se referem aos 8 bits do canal *red* do primeiro pixel da imagem. Dessa maneira, por meio da notação de indexação do *numpy*, ao acessar o elemento IMAGEM[0, 7, 0], estaremos acessando o bit menos significativo do primeiro pixel.

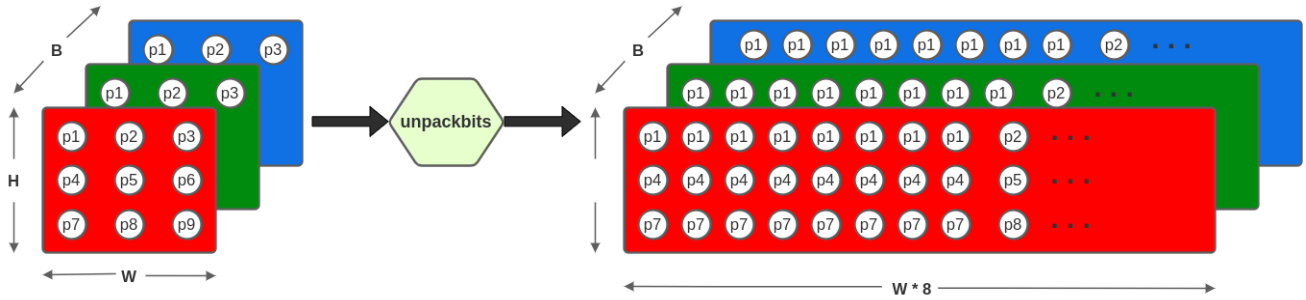

Figure 1: Ilustração da função *unpackbits*

A partir de uma representação dos bits da imagem e de uma representação em bits do texto a ser inserido, precisamos definir uma ordem de inserção ao longo de 3 dimensões: linhas, colunas e bandas. Decidiu-se inserir os bits referentes ao texto na ordem banda -> coluna -> linha, de modo que, de acordo com a Figura 1, o primeiro bit modificado seria o p1 vermelho, depois o p1 verde, depois o p1 azul, depois o p2 vermelho, depois o p2 verde, depois o p2 azul e assim por diante. Essa escolha facilita a operação vetorial de inserção dos bits da imagem, já que podemos acessar todos os bits que serão modificados por meio da seguinte sintaxe:

- IMAGEM[: 7::8, :] - acessa todos os bits do último plano de bits da imagem (plano 0);
- IMAGEM[:, 6::8. :] - acessa todos os bits do penúltimo plano de bits da imagem (plano 1);
- IMAGEM[:, 5::8, :] - acessa todos os bits do ante-penúltimo plano de bits da imagem (plano 2).

Uma escolha de implementação é a de que o usuário pode escolher um dentre os 3 últimos planos de bits de preferência para inserção do texto. Porém, se o plano de bits especificado não for grande o suficiente, **utiliza-se uma ordem de preferência para a inserção nos outros dois planos de bits permitidos**. Para reduzir a influência da mensagem na imagem, a ordem padrão é a inserção no último plano, em seguida no penúltimo e, por fim, no ante-penúltimo. Dessa maneira, se o usuário escolhe a inserção de uma mensagem muito grande no plano de bits 2 (ante-penúltimo), a ordem de inserção da mensagem será 2 -> 0 -> 1, conforme podemos observar no exemplo de execução a seguir:

```
python3 codificar.py ../sample_images/baboon.png ../sample_texts/sample3.txt
    2 coded_image.png
##################################################
Trying to fit remaining message in bit plane 2 ...
Warning: message didn't fit entirely in bit plane 2, with 1354200 message
   bits remaining.

##################################################
Trying to fit remaining message in bit plane 0 ...
Warning: message didn't fit entirely in bit plane 0, with 567768 message
   bits remaining.

##################################################
Trying to fit remaining message in bit plane 1 ...
Sucess: message fits entirely in bit plane 1

Saving output image as ../outputs/coded_image.png
```

Listing 3: exemplo de ordem de prioridade dos planos de bits

Um dos principais desafios na implementação do código *codificar.py* foi o de **determinar o índice de acesso da matriz da imagem de acordo com o tamanho da mensagem a ser codificada**. A sintaxe de acesso mostrada anteriormente acessa todos os bits de um determinado plano de bits. Porém, se a mensagem possuir uma quantidade de bits inferior à quantidade de bits do plano inteiro, torna-se necessário indexar de acordo com o tamanho da informação.

Para facilitar esse processo e manter as operações vetorizadas, realizamos uma checagem em relação ao tamanho, um *reshape* da matriz da imagem e uma variável temporária, conforme ilustrada no trecho de código a seguir:

```

def encrypt_image_bit_plane(input_image: iio.core.util.Array , ascii_text:
   np.array, bitplane: int = 7):
    """
    Encrypts an ASCII text into an image bit plane.
    The order in which the image is filled is: bands -> columns -> rows, i.e
   . the message first occupies the
    3 bands of first column and first row, then the 3 bands of second column
    and first row and so forth.

    Parameters:
    input_image: image in which the text will be hided.
    ascii_text: a numpy array of 1s and 0s representing an ASCII text
    bit_plane: a integer representing which bit plane to encrypt the message
   . Could be 7 - least significative;
               6 - second least significative; 5 - third least significative

    Returns:
    output_image: image with hidden text
    remaining_text: the remaining message that wasn't able to fit in a
   single image bit plane
```

```
17      """
18      unpacked_bits_image = np.unpackbits(input_image, axis = 1)
19
20      # temp variable that will be reshaped to store ascii_text
21      temp = unpacked_bits_image[:, bitplane::8, :].copy()
22      temp = temp.reshape(-1)
23
24      # check to see if message fits in bit_plane
25      image_available_space = np.prod(input_image.shape)
26      if image_available_space < len(ascii_text):
27          print("Warning: message didn't fit entirely in bit plane {}, with {}
        message bits remaining.\n".format(-bitplane + 7, len(ascii_text) -
    image_available_space))
28          # filling available space
29          temp[:] = ascii_text[:image_available_space]
30          remaining_text = ascii_text[image_available_space:]
31      else:
32          print('Sucess: message fits entirely in bit plane {}\n'.format(-
    bitplane + 7))
33          temp[:len(ascii_text)] = ascii_text[:]
34          remaining_text = np.array([-1])
35
36      # returning temp original shape and assigning it to the original image
37      temp = temp.reshape(unpacked_bits_image[:, bitplane::8, :].shape)
38      unpacked_bits_image[:, bitplane::8, :] = temp
39
40      output_image = np.packbits(unpacked_bits_image, axis = 1)
41
42      return output_image, remaining_text
```

Listing 4: Operação de codificação do texto na imagem

Na linha 21, definimos a variável temporária com todos os bits do plano de bits especificado e em seguida realizamos um *reshape* para obter um vetor unidimensional. Na linha 26, realizamos uma checagem para verificar se o espaço disponível no plano de bits é menor que o tamanho total da mensagem. Caso seja, podemos inserir a mensagem (até o ponto suportado) em todo o plano de bits (conforme a linha 29). Caso contrário, é preciso indexar o vetor unidimensional temporário apenas até o tamanho da mensagem (conforme linha 33). Ao final, realizamos um *reshape* de volta ao formato original da matriz e atribuímos isso à imagem de saída.

3.3 Decodificação de uma Imagem com Texto Escondido

Para o processo de decodificação, a questão dos índices também é importante: é preciso encontrar o índice do caractere que representa o final da mensagem escondida. Conforme descrito na seção 3.1, sempre adicionamos o caractere "\0" ao final dos textos de entrada. Conforme o código ASCII, a representação deste símbolo é o 00000000. Dessa maneira, precisamos percorrer os bits do plano de bits especificado, checando se cada conjunto de 8 bits corresponde à esse valor.

Para realizar isso, novamente é necessário atentar para a ordem dos planos de bits. O usuário

deve indicar como parâmetro de entrada o plano inicial da mensagem, que deve coincidir com o plano especificado no momento de codificação. O programa percorrerá, a partir daí, os planos de bits na mesma ordem definida na seção anterior, iniciando no plano dado pelo usuário e em seguida os dois restantes, na ordem crescente de significância.

O processo de decodificação de um único plano de bits é mostrado no trecho de código a seguir, referente à função *uncrypt_text_from_image*.

```
1
2  def uncrypt_text_from_image(input_image: iio.core.util.Array, bit_plane: int
       = 7):
3      """
4      Uncrypts a sequence of bits representing a sequence ASCII characters
      from
5      an input image bit plane.
6      The order in which the text is obtained is: bands -> columns -> rows.
7
8      Parameters:
9      input_image: image with hidden text
10     bit_plane: a integer representing the bit plane in wich the message is
      hidden.
11                 Could be 7 - least significative;
12                 6 - second least significative;
13                 5 - third least significative
14
15     Returns:
16     secret_message: sequence of ASCII characters representing the decoded
      secret message
17     EOF_index: index of EOF character.
18     """
19     # unpacking bits from image
20     unpacked_bits_image = np.unpackbits(input_image, axis = 1)
21
22     # selecting chosen bit plane
23     bit_plane_message = np.packbits(unpacked_bits_image[:,bit_plane::8,:].
      reshape(-1))
24
25     # searching for end of file indicator ('\0')
26     EOF_index = np.where(bit_plane_message == 0)
27
28     if (EOF_index[0].size == 0):
29         print("Logging: the message ending is not in bit plane {}\n".format
      (-bit_plane + 7))
30         secret_message = ''.join(str(v) for v in list(unpacked_bits_image[:,
       bit_plane::8, :].reshape(-1)))
31     else:
32         print("Logging: end of message found in bit plane {}\n".format(-
      bit_plane + 7))
33         secret_message = ''.join(str(v) for v in list(unpacked_bits_image[:,
       bit_plane::8, :].reshape(-1)[:EOF_index[0][0]*8]))
34
```

```
35      return secret_message, EOF_index
```

Listing 5: Decodificação de uma mensagem em um plano de bit

Iniciamos a função realizando o desempacotamento de bits, da mesma maneira que na seção passada. Em seguida, acessamos apenas o plano de bits especificado na entrada da função e fazemos um empacotamento, conforme a linha 23 do código. Ao realizarmos esse empacotamento, estamos juntando os bits 8 a 8, o que realiza uma transformação dos caracteres em ASCII para inteiro. A representação inteira do caractere "\0" é o próprio 0, assim, por meio da linha 26, podemos obter o índice do caractere que representa o EOF. Caso esse índice seja vazio, assumimos que o final da mensagem não está no plano de bits especificado e que devemos prosseguir na ordem de prioridade procurando nos demais planos de bits. Caso esse índice não seja vazio, podemos ter certeza de que ele representa o final da mensagem e devemos extrair os bits apenas até o índice do EOF (conforme linha 33).

Como percorremos os conjuntos de 8 bits (referentes aos caracteres ASCII) exatamente da mesma maneira que a codificação foi realizada, podemos garantir que o primeiro inteiro referente ao 0 é de fato o final da mensagem. Destaca-se que a variável EOF_index se refere ao índice do EOF na variável *bit_plane_message*, que contém os inteiros referentes aos bits. É preciso multiplicar esse índice por 8 para obter o índice correto na variável *unpacked_bits_image*, que possui os bits desempacotados, prontos para serem acessados.

4 Resultados

Como estamos utilizando apenas os 3 planos de bits menos significativos, a inserção dos textos nas imagens não produz quase nenhuma diferença. Dessa forma, torna-se importante visualizar os planos de bits separadamente para percebermos a distorção causada pela inserção das mensagens.

Realizamos testes com a imagem *peppers.png*, que possui uma dimensão de 512 linhas por 512 colunas. Em cada banda dessa imagem, cabem até $512 \times 512 \times 3 = 786432$ bits. As amostras de textos possuem tamanhos diversos:

- Texto 1: 449064 bits - menor que 1 plano de bits;
- Texto 2: 898048 bits - maior que 1 plano de bits, mas menor que 2 planos de bits;
- Texto 3: 2140632 bits - maior que 2 planos de bits, mas menor que 3 planos de bits;
- Texto 4: 3531584 - maior que 3 planos de bits.

Os experimentos consistiram em codificar cada um dos textos na imagem *peppers.png*. O resultado para cada experimento pode ser observado nas Figuras 2 a 5. A imagem de referência, bem como o resultado de cada experimento são mostrados na Figura 6. Apenas a banda de cor vermelha está sendo mostrada em todas as imagens.

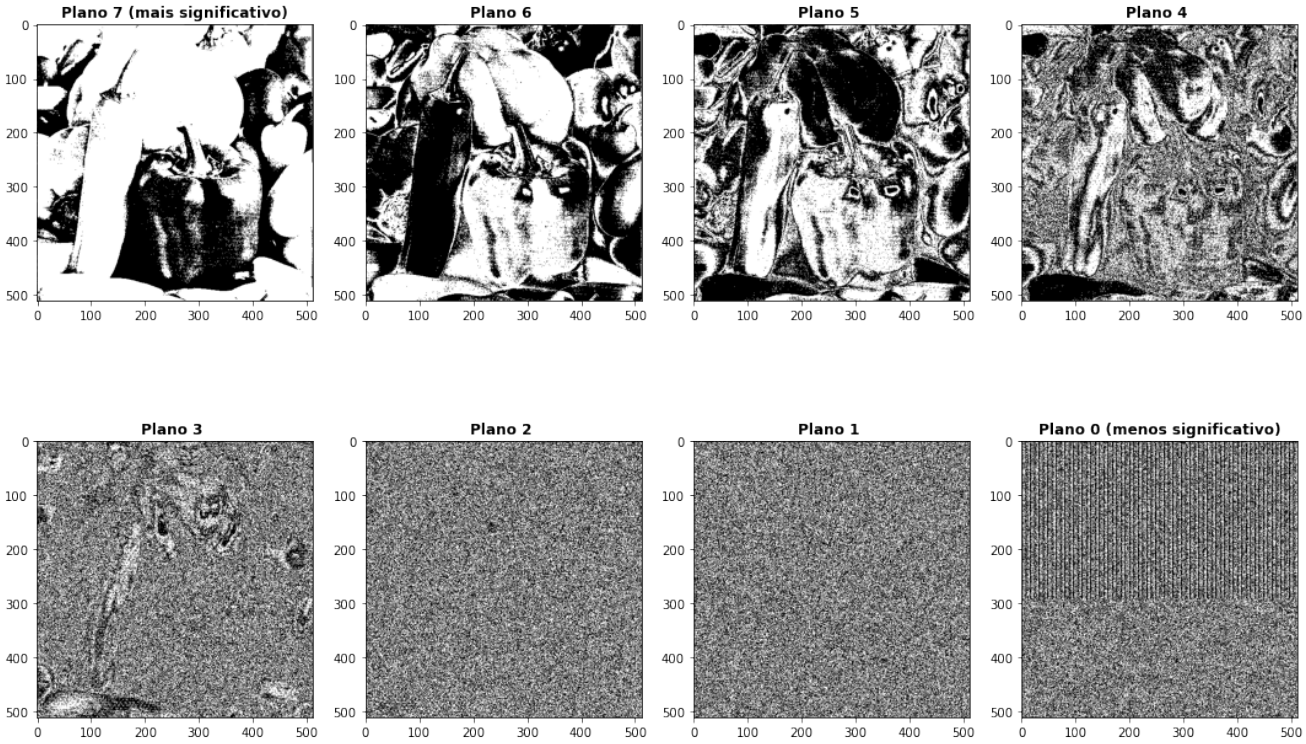

Figure 2: Planos de bits do experimento 1

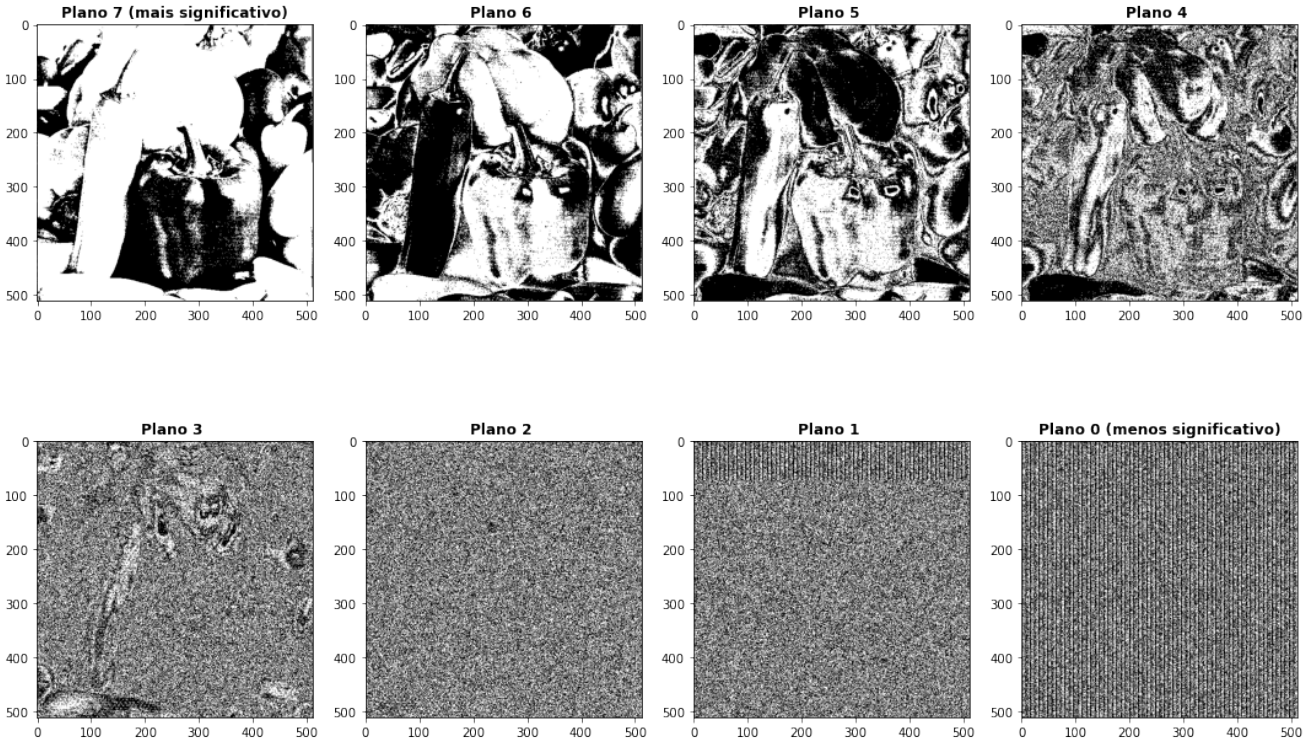

Figure 3: Planos de bits do experimento 2

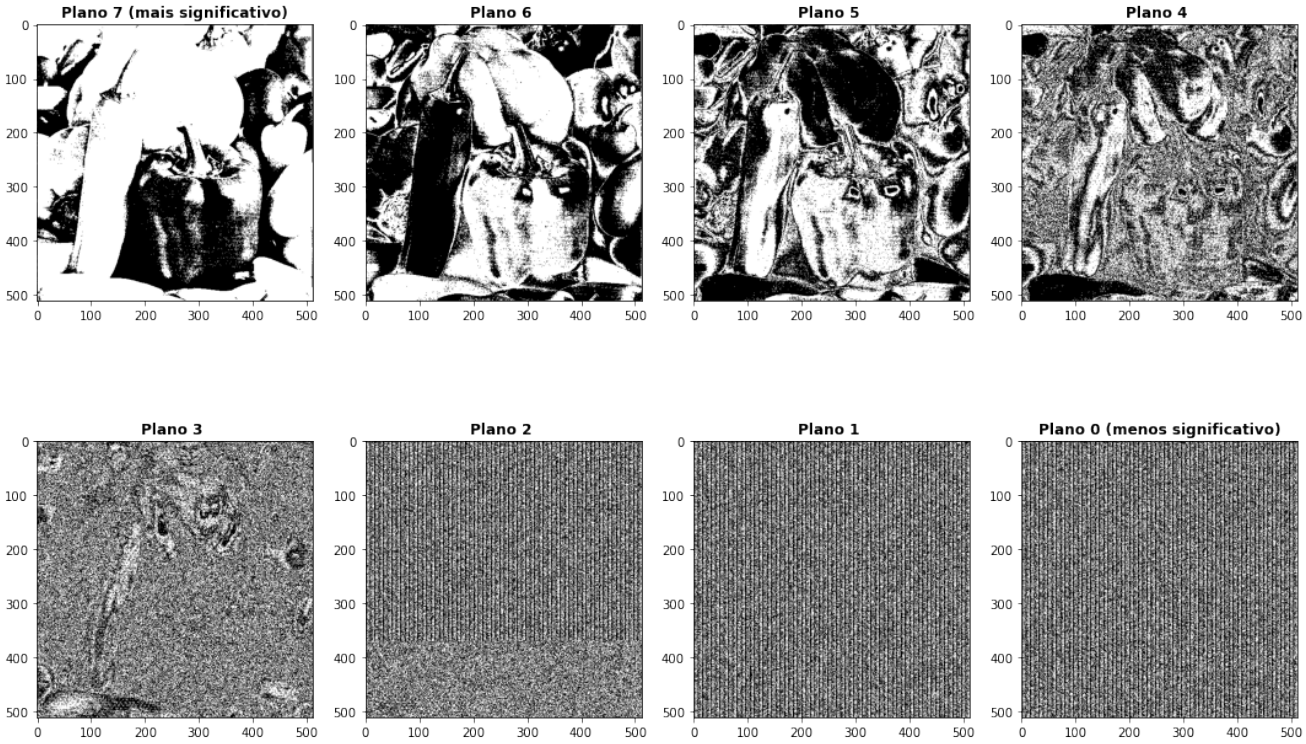

Figure 4: Planos de bits do experimento 3

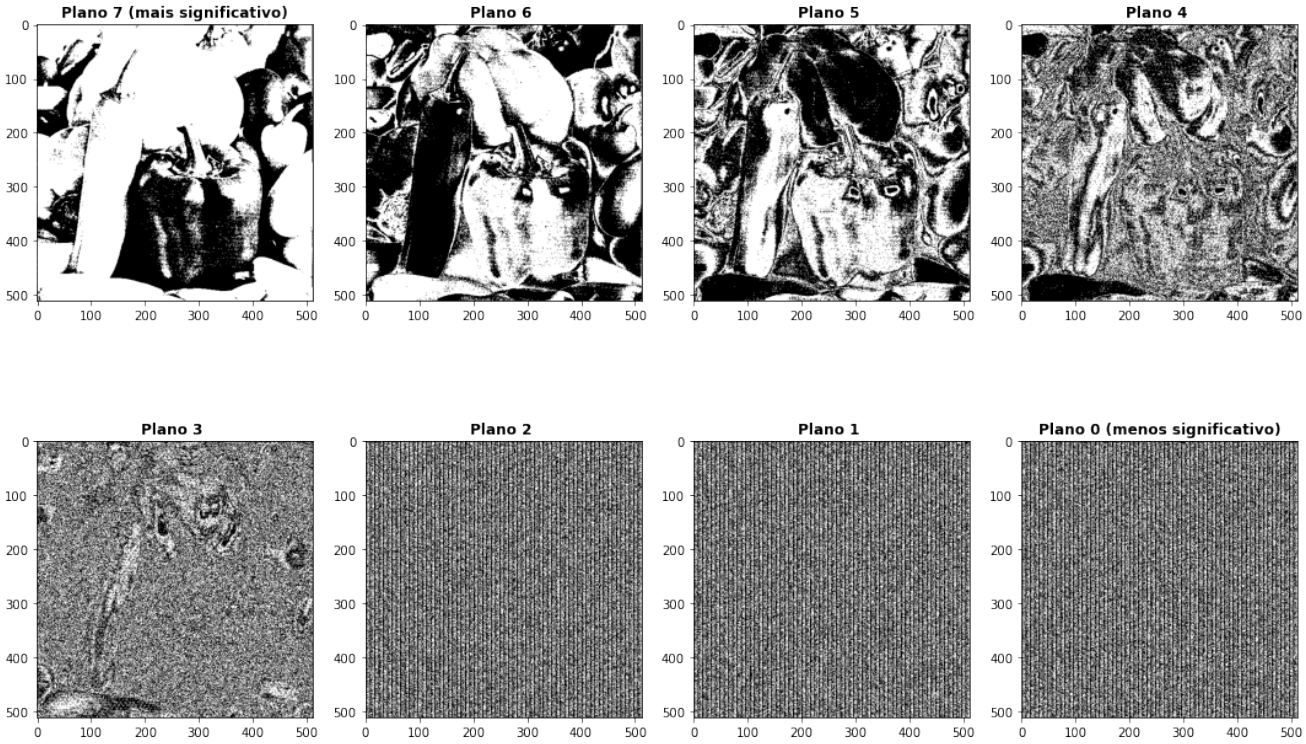

Figure 5: Planos de bits do experimento 4

Figure 6: Comparação da imagem de referência com os resultados dos experimentos

Embora haja bastante ruído nos planos de bits menos significativos, é possível notar a faixa que as mensagens ocupam. Na Figura 2, por exemplo, podemos perceber uma divisão na imagem, por volta da metade da altura do último plano de bit. Esse é o ponto em que há o término da mensagem. Na Figura 4, por sua vez, os últimos 2 planos de bits são completamente ocupados pela mensagem, enquanto no plano 2 é possível observar o término do texto.

Nos arquivos complementares a este relatório, é possível observar o resultado da decodificação das imagens destes quatro experimentos nos arquivos *decoded_text1.txt*, *decoded_text2.txt*, *decoded_text3.txt*, *decoded_text4.txt* no diretório *outputs*. Percebe-se que nos experimentos de 1 a 3 foi possível decodificar inteiramente a mensagem, de modo que os arquivos são idênticos aos *sample1.txt*, *sample2.txt* e *sample3.txt*, respectivamente. Já no experimento 4, a mensagem é menor do que a *sample4.txt*, já que não coube nos 3 planos de bits permitidos.

5 Limitações

Os programas aceitam apenas imagens no formato PNG e arquivos de texto no formato .txt. Há uma verificação referente a quantidade e tipos de argumentos de entrada.

A maior limitação dos programas se refere à ordem do plano de bits utilizada. Caso o usuário realize a codificação especificando o plano de bits 2, por exemplo, e em seguida realize a decodificação especificando o plano de bits 0, não é possível prever exatamente o resultado. Caso a mensagem seja muito grande de modo que use os 3 planos de bits, será possível realizar a decodificação, porém, a ordem do texto estará trocada. Caso a mensagem seja pequena de modo que coubesse apenas no plano de bit especificado, o programa apresentará erro de execução durante a decodificação, já que encontrará caracteres inválidos (que pertencem à imagem e não ao texto) no código ASCII.

6 Conclusões

Neste trabalho, desenvolveu-se dois programas *codificar.py* e *decodificar.py* de modo a implementar uma técnica simples de esteganografia utilizando os planos de bits menos significativos de uma imagem.

De acordo com o modo de indexação proposto, foi possível realizar toda a codificação e decodificação utilizando operações vetorizadas da bilblioteca numpy. As alterações nas imagens são imperceptíveis ao utilizar apenas os planos de bits especificados.

References

ROCHA, A.; COSTA, H.; CHAVES, L. Camaleão: um software para segurança digital utilizando esteganografia. 1