

# MC920 - Trabalho 4

Yuri Luiz de Oliveira  
188802

30 de Dezembro de 2020

## 1 Introdução

O trabalho consiste em esconder mensagens em imagens coloridas através da técnica de esteganografia. Esta técnica permite extrair a mensagem da imagem modificada sem que as mudanças nas intensidades de vermelho, azul e verde sejam perceptíveis.

## 2 Ambiente de desenvolvimento e testes

A solução para o trabalho foi implementada e testada em Python 3.8.5, com auxílio das bibliotecas numpy, opencv, matplotlib, argparse. O sistema operacional utilizado foi Ubuntu 20.04.

### 2.1 Preparando o ambiente

Para instalar as bibliotecas necessárias basta entrar na pasta do trabalho e rodar o comando

```
$ pip3 install -r requirements.txt
```

## 3 Rodando a solução

A solução aceita argumentos de linha de comando e deve ser executada através do interpretador de python 3. O seguinte comando codifica a imagem presente em "imagens/baboon.png", no plano de bits 0, com o conteúdo contido dentro do arquivo "text/hello\_world.txt" e salva no arquivo "images\_codified/baboon\_hello\_world\_bp0.png".

```
$ python3 steganography.py \
    -m codify \
    -ii images/baboon.png \
    -ti text/hello-world.txt \
    -bp 0 \
    -io images_codified/baboon_hello_world_bp0.png
```

O seguinte comando decodifica a imagem presente em ”images\_codified/baboon\_hello\_world\_bp0.png”, no plano de bits 0, e salva o conteúdo da mensagem decodificada no arquivo de texto ”text\_decodified/baboon\_hello\_world\_bp0.txt”.

```
$ python3 steganography.py \
    -m decodify \
    -ii images_codified/baboon_hello_world_bp0.png \
    -bp 0 \
    -to text_decodified/baboon_hello_world_bp0.txt
```

### 3.1 Rodando a solução para todas as imagens, mensagens e planos de bits

O seguinte comando codifica todos os arquivos de textos contidos no diretório ”text/” nas imagens contidas no diretório ”images/” com planos de bits 0 (é possível passar qualquer plano de bit entre 0 e 7) e salva no diretório ”images\_codified/”.

```
$ ./codify-all 0
```

O seguinte comando decodifica todas as imagens contidas no diretório ”images\_codified/” e salva os arquivos de texto resultantes no diretório ”text\_decodified/” com planos de bits 0 (é possível passar qualquer plano de bit entre 0 e 7).

```
$ ./decodify-all 0
```

Obs.: Os comandos codify-all e decodify-all aceitam somente um plano de bits por vez. Para rodar para todos os planos de bits, é necessário rodar os comandos uma vez para cada plano de bits.

## 4 Detalhes e decisões de implementação

As mensagens são lidas através de arquivos de texto (.txt) e interpretadas como bits para ser possível salvar em um plano de bits da imagem. As imagens são representadas como bytes (valores de 8 bits) manipuladas através dos seguintes operadores binários: << (shift esquerdo), >> (shift direito), & ('e' bit a bit), | ('ou' bit a bit) e ~ (negação bit a bit). Estes operadores operam sobre cada bit do valor. Vide seção ”Operadores binários” para explicações sobre os funcionamentos destes operadores.

Para facilitar a manipulação, as imagens são processadas unidimensionalmente, ou seja, para salvar a mensagem na imagem, o vetor *numpy* que possui o valor dos pixels da imagem é redimensionado de forma que fica com uma única dimensão. O vetor *numpy* que representa uma imagem de dimensão 512 por 512 terá dimensão:

$$512 \times 512 \times 3 = 786.432,$$

pois as intensidades de pixels ficam na mesma dimensão. No vetor unidimensional, aplicou-se operadores binários combinados de forma que, para cada intensidade de bits da imagem, o bit no plano alvo seria 0 ou 1, dependendo do bit da mensagem. É importante notar que vetores *numpy* suportam operadores binários, aplicando-os a cada valor do vetor. Isto foi crucial para a implementação completamente vetorizada da solução.

## 4.1 Operadores binários

### 4.1.1 Shift esquerdo

Todos os bits são deslocados  $n$  posições para a esquerda e zeros são colocados à direita. Exemplo:

$$1_{10} \ll 2_{10} = 4_{10} = 100_2$$

## 4.2 Shift direito

Todos os bits são deslocados  $n$  posições para a direita e zeros são colocados à esquerda. Exemplo:

$$4_{10} \gg 2_{10} = 1_{10} = 1_2$$

### 4.2.1 'e' bit a bit

O resultado terá 1 nas posições em que ambos os operandos possuem 1; 0 caso contrário. Exemplo:

$$5_{10} \& 1_{10} = 101_2 \& 001_2 = 001_2 = 1_{10}$$

### 4.2.2 'ou' bit a bit

O resultado terá 1 nas posições em que um dos os operandos possuem 1; 0 caso contrário. Exemplo:

$$6_{10} | 1_{10} = 110_2 | 001_2 = 111_2 = 7_{10}$$

### 4.2.3 negação bit a bit

Inverte o valor de cada bit do número. Exemplo:

$$\sim 5_{10} = \sim 101_2 = 010_2 = 2_{10}$$

## 5 Experimentos

Os experimentos foi feitos utilizando a mensagem contida no arquivo de texto "text/lorem-ipsum.txt", que possui 57 linhas, 2044 palavras e 12308 caracteres nas imagens contidas dentro do diretório "images/". Este arquivo foi selecionado para os testes, pois sua codificação toma aproximadamente metade da imagem da Monalisa. A mensagem cabe em todas as imagens utilizas no experimentos.

## 5.1 Imagens utilizadas

### 5.1.1 baboon.png

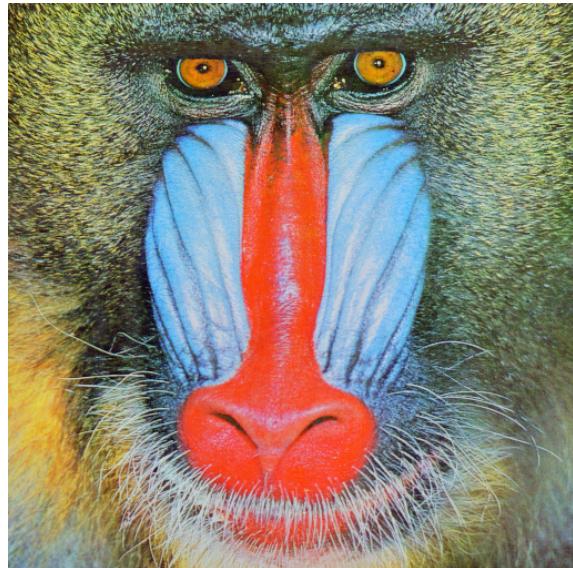


Figure 1: Imagem retirada de [https://www.ic.unicamp.br/~helio/imagens\\_coloridas/baboon.png](https://www.ic.unicamp.br/~helio/imagens_coloridas/baboon.png)

### 5.1.2 monalisa.png



Figure 2: Imagem retirada de [https://www.ic.unicamp.br/~helio/imagens\\_coloridas/monalisa.png](https://www.ic.unicamp.br/~helio/imagens_coloridas/monalisa.png)

### 5.1.3 peppers.png



Figure 3: Imagem retirada de [https://www.ic.unicamp.br/~helio/imagens\\_coloridas/peppers.png](https://www.ic.unicamp.br/~helio/imagens_coloridas/peppers.png)

### 5.1.4 watch.png



Figure 4: Imagem retirada de [https://www.ic.unicamp.br/~helio/imagens\\_coloridas/watch.png](https://www.ic.unicamp.br/~helio/imagens_coloridas/watch.png)

## 5.2 Resultados obtidos

### 5.2.1 Plano de bits 0

Imagens codificadas no plano de bits 0 geram resultados imperceptíveis, como visto nas figuras a seguir:

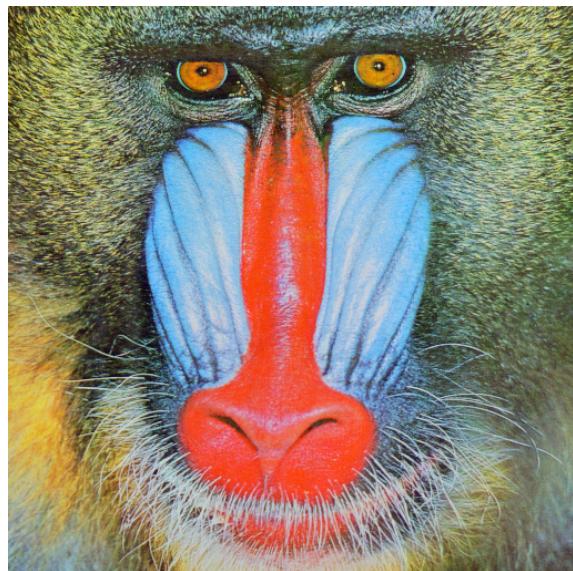


Figure 5: Imagem baboon.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 0



Figure 6: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 0



Figure 7: Imagem peppers.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 0



Figure 8: Imagem watch.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 0

Obs.: Para os planos de bits 1 a 7 utilizaremos somente será mostrada somente a imagem resultante da Monalisa, pois as alterações da codificação cortam esta imagem aproximadamente no meio.

### 5.2.2 Plano de bits 1

Ainda não é perceptível alterações na imagem.



Figure 9: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 1

### **5.2.3 Plano de bits 2**

Ainda não é perceptível alterações na imagem.



Figure 10: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 2

### **5.2.4 Plano de bits 3**

No plano de bits 3 começa-se a haver indícios de alterações na imagem, embora muito sutis.

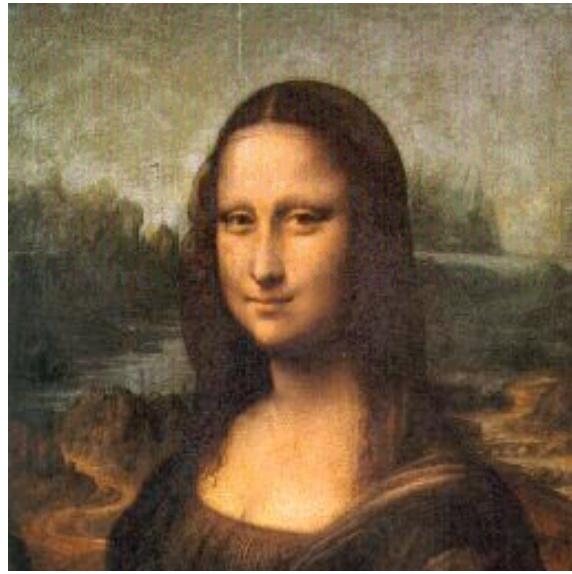


Figure 11: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 3

Para facilitar a visualização das alterações da imagem, veja a Figura 12. Nesta figura é visível um pouco de ruído na porção de cima da imagem.

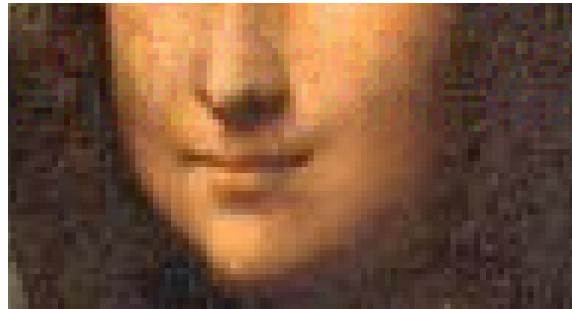


Figure 12: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 3 com zoom próximo ao meio da imagem

#### 5.2.5 Plano de bits 4

A partir do plano de bit 4 é bastante perceptível a diferença entre as metades superior (com a mensagem) e inferior (inalterada) da imagem. A cada plano de bit havendo ruído mais intenso.

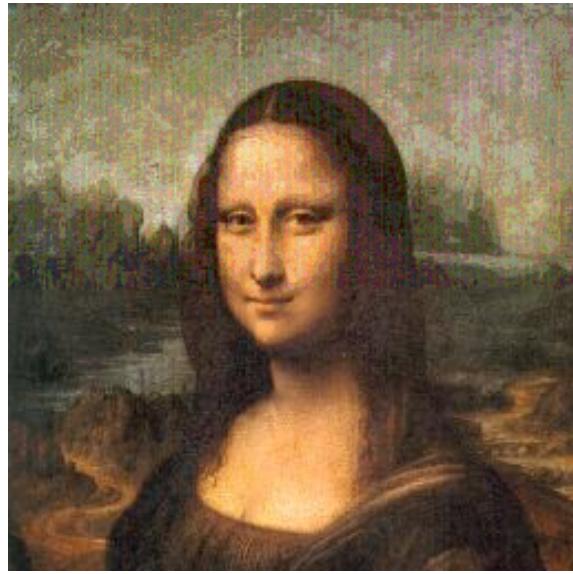


Figure 13: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 4

#### 5.2.6 Plano de bits 5



Figure 14: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 5

### 5.2.7 Plano de bits 6

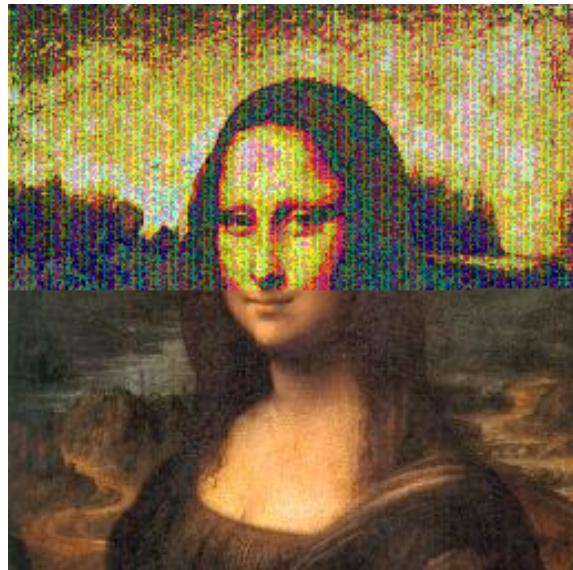


Figure 15: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 6

### 5.2.8 Plano de bits 7

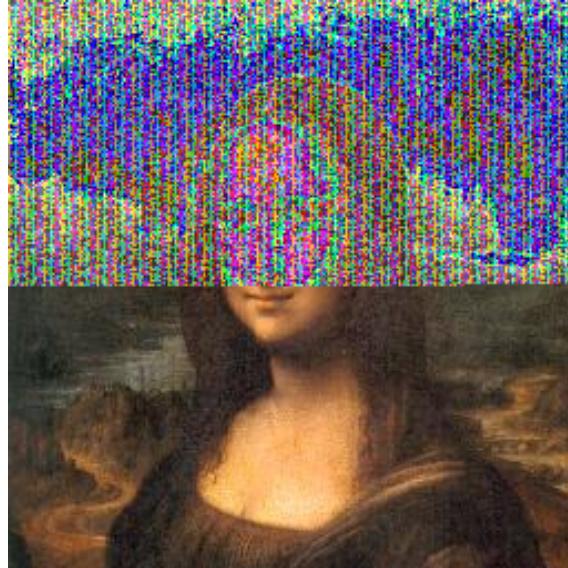


Figure 16: Imagem monalisa.png codificada com mensagem contida em lorem-ipsum.txt no plano de bits 7

## 5.3 Análise dos resultados

Como esperado, ao codificar a mensagem em planos de bits mais significativos as alterações na imagem ficaram mais aparentes. Vamos analisar matematicamente a diferença entre codificação no plano de bits:

sendo  $n$  o plano de bit utilizado na codificação, a alteração nas intensidades de vermelho, verde e azul de cada pixel é expressa por

$$2^n,$$

Ou seja, um crescimento exponencial para cada plano de bits. Temos, então, possíveis alterações de (iniciando do 0 e terminando no 7):

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

Entre os bits 0 e 7, temos diferença 127 em cada RGB de cada pixel da imagem.

#### 5.4 Identificando se existem mensagem

Para analisar se existe alguma mensagem escondida em algum plano de bit específico da imagem, podemos analisar os planos de bits separadamente. A seguir, vê-se os planos de bits 0, 1, 2 e 7 da Monalisa codificada com a mensagem contida em "text/lorem-ipsum.txt" no plano de bits 0.



Figure 17: Plano de bits 0 da imagem codificada da Monalisa no plano de bit 0 com mensagem contida em "text/lorem-ipsum.txt"

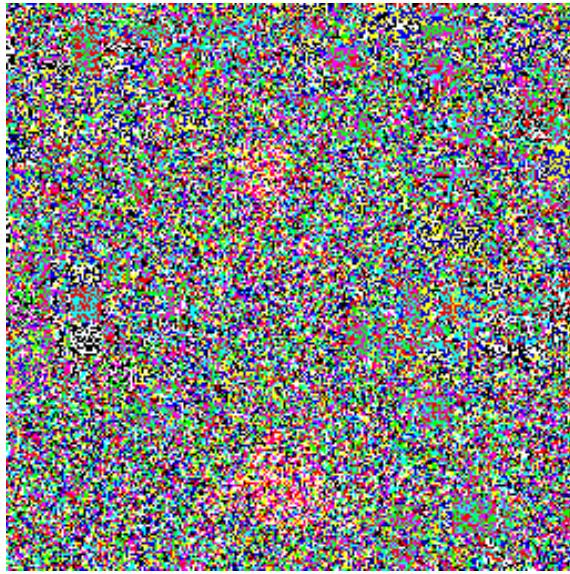


Figure 18: Plano de bits 1 da imagem codificada da Monalisa no plano de bit 0 com mensagem contida em "text/lorem-ipsum.txt"

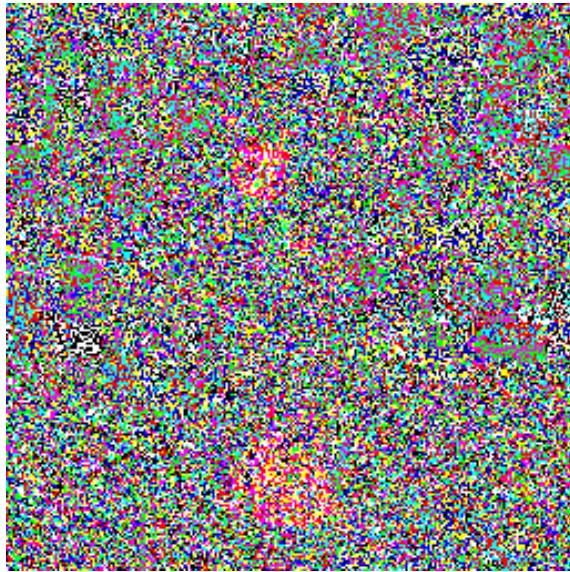


Figure 19: Plano de bits 2 da imagem codificada da Monalisa no plano de bit 0 com mensagem contida em "text/lorem-ipsum.txt"

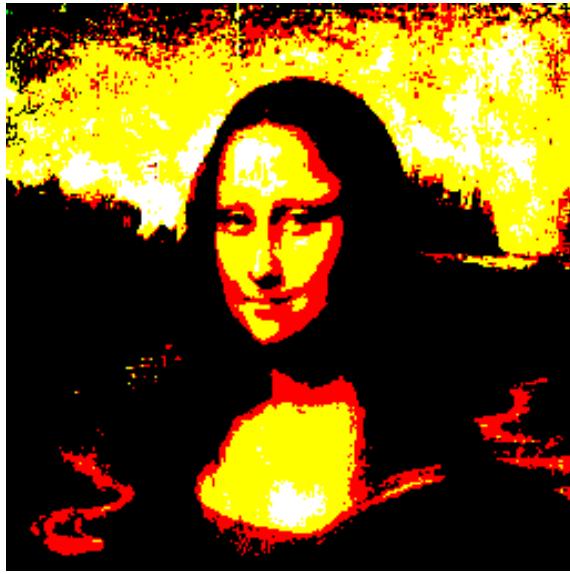


Figure 20: Plano de bits 7 da imagem codificada da Monalisa no plano de bit 0 com mensagem contida em "text/lorem-ipsum.txt"

É visível que há uma mensagem na metade superior do plano de bits 0, mesmo, na imagem codificada (Figura 6) no plano de bits 0 não havendo diferenças perceptíveis. Enquanto isso, os outros planos de bits da imagem se mantém intactos.

## 6 Principais falhas durante o desenvolvimento

O método *tostring* do *numpy* retorna uma byte string, que ao ser salva em um arquivo de texto resultava em "b'Hello world'". Entendi, então, que é necessário formatar a string como *utf-8* para que fosse salva corretamente.

## 7 Limitações

Dado que a implementação depende de caractere delimitador, não é possível utilizar este caractere no interior da mensagem, uma vez que a mensagem será interpretada como finalizada assim que encontrar o segundo delimitador. Utilizando '@' como delimitador, por exemplo, impossibilita-se o envio de endereços de e-mail, por exemplo. Adotou-se esta abordagem a fim de manter a solução tão simples quanto possível. Ainda, há uma limitação de tamanho de mensagem, em que ela deve caber (codificada) dentro da imagem.

## 8 Conclusões

A técnica de esteganografia pode ser muito útil para transmitir mensagens de forma secreta, principalmente quando não se espera que haja algo escondido na imagem. Entretanto, é um método que, se identificado por agentes interceptadores, é facilmente decifrável, pois, descobrindo-se em que plano de bits encontra-se a mensagem, é necessário somente decodificar em cima daquele plano de bits. Diferentemente de criptograma por chave, por exemplo, na qual, mesmo havendo uma interceptação da mensagem, só é possível decifrá-la através de uma senha.