

# Trabalho 4 - Esteganografia

MC920 - Introdução ao Processamento de Imagem Digital

Victor Ferreira Ferrari  
RA 187890  
vferrari@mpc.com.br

31 de Outubro de 2019

## 1 Introdução

Uma clássica técnica para esconder mensagens dentro de imagens é a esteganografia. Uma implementação consiste em alterar bits de cada pixel (se a imagem for colorida, cada banda também) para esconder uma mensagem, em código ASCII.

Analizando os planos de bits de uma imagem, percebe-se que quanto mais alta a ordem, ou seja, mais significativo o bit analisado, mais ele representa a imagem final. Assim, os bits menos significativos não são tão representativos, e alterações neles podem não ser muito visíveis na imagem completa. Por isso, normalmente os bits alterados para esconder a mensagem são os menos significativos.

O intuito do projeto é implementar a técnica de esteganografia em mensagens coloridas, alterando um bit de cada banda de cor de cada pixel da imagem para esconder uma mensagem, lida de um arquivo de texto. O programa deve ser utilizado para alterar os bits menos significativo, mas ele aceita alteração de qualquer dos 8 bits.

## 2 Características do Programa

O programa foi feito na linguagem *Python*, com auxílio das bibliotecas externas NumPy e OpenCV (CV2). Os argumentos são passados na execução via `argv`, permitindo modificar diversas condições de execução.

Dois programas foram implementados: um para codificar uma mensagem em uma imagem, e outro para decodificar uma mensagem e uma imagem. A imagem deve ser colorida.

Os argumentos do programa de codificação `encode.py` podem ser:

- `in_img`: Arquivo que contém a imagem a ser codificada;
- `in_txt`: Nome do arquivo de texto contendo a mensagem;
- `bit_plane`: Plano de bits a ser alterado, do menos significativo (0) ao mais significativo (7). Deve ser entre 0 e 3 para melhor resultado, mas aceita até 7;
- `out_img`: Nome para a imagem de saída, codificada;
- `--folder`: opcional, pasta de saída (precisa existir).

Os argumentos do programa de codificação `decode.py` podem ser:

- `img`: Arquivo que contém a imagem a ser codificada;

- `bit_plane`: Plano de bits que contém a mensagem, do menos significativo (0) ao mais significativo (7);
- `out_txt`: Nome do arquivo de texto de saída, para conter a mensagem decodificada;
- `--folder`: opcional, pasta de saída (precisa existir).

## 3 Implementação

Como visto na seção 2, dois programas separados foram implementados para codificação e decodificação.

Na codificação, após a leitura do arquivo-texto, é feita a transformação do texto em uma sequência de códigos ASCII. Isso é feito pela função `fromstring` da biblioteca *NumPy* para separação de caracteres, combinado com a função `view` para transformação em inteiros.

Esse vetor de inteiros é transformado em vetor de bits (elementos 0 ou 1) pela função `unpackbits`. Junto com essa transformação, é adicionado um código que indica "final de mensagem" (EOM), que é o código 254. Assim, o programa não consegue incluir mensagens que contém o caractere com código 254.

Com a sequência de bits a serem adicionadas à imagem, é feita a codificação a partir da representação unidimensional (*flattened*) da imagem. São separados os primeiros `n` pixels, sendo `n` o número de bits da sequência, e a codificação é feita por meio de operações com bits e máscaras. As máscaras dependem do plano de bits a ser alterado, deslocando um bit "1" pelo número do plano, e invertendo para a máscara correspondente ao bit "0". Se o elemento da sequência é "1", utiliza a primeira máscara. Senão, utiliza a máscara invertida. A imagem é retornada à forma original e salva.

O processo de decodificação é o oposto, começando com a extração dos bits por meio da representação *flattened* da imagem, deslocando o valor pelo plano de bits a ser recuperado e realizando uma operação AND com o valor 1 para recuperar apenas o bit. Para transformar em códigos ASCII de novo, basta utilizar a função `packbits` após separar em conjuntos de 8 bits.

Com a sequência de códigos, é encontrado o EOM para separar apenas os correspondentes à mensagem, e a função `tostring` retorna os códigos a texto, que é salvo em um arquivo recebido como argumento.

As imagens e os textos são salvos na pasta "Outputs/" (caso outra pasta não tenha sido passada como argumento na chamada).

## 4 Resultados e Comparação

Foram obtidos resultados para 4 imagens coloridas. As imagens de entrada estão disponíveis em [https://www.ic.unicamp.br/~helio/imagens\\_coloridas/](https://www.ic.unicamp.br/~helio/imagens_coloridas/). Todos os testes feitos estão na entrega, com os textos na pasta "Texts/" e os resultados (imagens codificadas e textos recuperados) na pasta "Outputs/". As imagens originais podem ser vistas na figura 1, em ordem de tamanho.

### 4.1 Teste Inicial

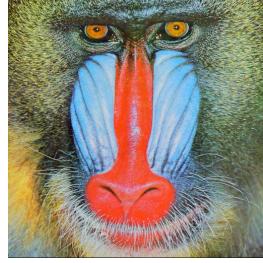
Inicialmente, foi utilizada a frase abaixo para testar o funcionamento do programa, na imagem 1a.



(a) Monalisa



(b) Peppers



(c) Baboon



(d) Watch

Figura 1: Imagens para esconder textos.

*O Hélio é um ótimo professor que deveria ministrar computação gráfica no semestre que vem!*

Após a decodificação, o texto foi recuperado sem perda alguma, exatamente como o original, e com o mesmo significado verdadeiro. Observando os planos de bits, as alterações não são visualmente óbvias pois o texto é muito pequeno, alterando poucos pixels, mas pode ser vista com análise mais minuciosa. Para percebermos a presença de um texto na imagem, precisamos testar textos mais longos. Como foi visto que o programa funciona, podemos testar outras características.

## 4.2 Textos Longos

Com a certeza do funcionamento do programa, foram codificados textos mais longos nas maiores imagens. Na figura 1b foi codificado o *script* completo do filme "Bee Movie", e na figura 1d foram codificados livros de domínio público adquiridos em puro ASCII via [Projeto Gutenberg](#).

O *script* foi codificado no **segundo** plano de bits da imagem, e os resultados podem ser vistos na figura 2.

Mesmo com um texto maior, observando a figura 2a não é possível perceber a presença do texto, mesmo estando em um plano de bits mais significativo. Isso demonstra como o método é efetivo em esconder texto na imagem. O texto foi recuperado exatamente como o original.

Porém, é possível perceber a presença da mensagem olhando os planos de bits de cada cor referentes ao bit 2, onde foi inserida a mensagem. Nesses planos, é perceptível o padrão similar à imagem final na segunda metade da imagem (pelo menos nas cores 0 e 1), mas um padrão diferente surge na metade superior da imagem, referente ao texto.

Ou seja, isso comprova que, caso interceptada, é possível facilmente descobrir se uma imagem contém texto escondido ou não, e em qual plano de bits ela se situa, apenas analisando os planos de bits. Então não é o modo mais seguro e secreto de se enviar uma mensagem codificada.

Por último, foram inseridos livros na imagem 1d. Os resultados para "O Médico e o Monstro", inserido no plano de bits 2, estão na figura 3.

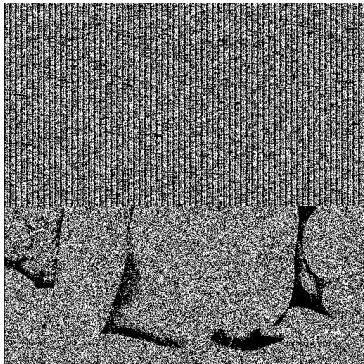
O texto foi recuperado exatamente como o original.

Nesse caso, é perceptível na imagem completa codificada a presença do texto, mesmo em planos menos significativos. Isso se deve a uma característica da imagem, que possui detalhamento nos bits menos significativos, então esse plano é representativo de detalhes na imagem final. Alterando-o, é possível enxergar o resultado na imagem final. Isso é mais perceptível nas partes escuras da imagem, nos cantos superiores (veja o canto superior direito para o melhor exemplo).

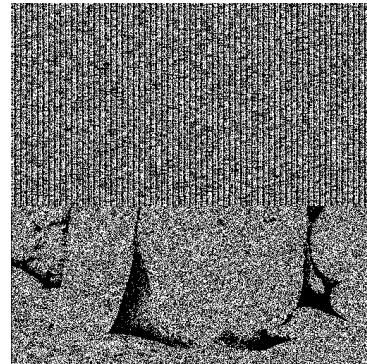
Observando os planos de bits, é comprovado que representam diversos detalhes da imagem, ao contrário da figura 1b, que possui algum detalhamento mas não tanto para resultar em grandes mudanças na imagem se alterado.



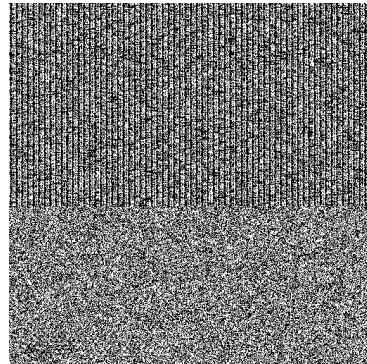
(a) Imagem Com Texto



(b) Plano de Bits 2: Cor 0



(c) Plano de Bits 2: Cor 1

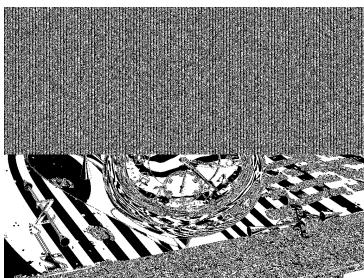


(d) Plano de Bits 2: Cor 2

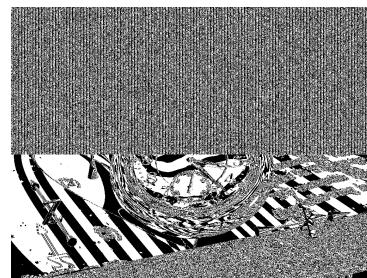
Figura 2: Imagem resultante após codificação do script de *Bee Movie*.



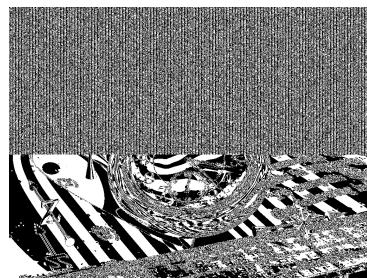
(a) Imagem Com Texto



(b) Plano de Bits 2: Cor 0



(c) Plano de Bits 2: Cor 1



(d) Plano de Bits 2: Cor 2

Figura 3: Imagem resultante após codificação do livro ”O Médico e o Monstro”.

Porém, algo diferente é visto na figura 2d, que não possui quase detalhamento nenhum. Isso se dá pela ausência de detalhes da cor que ele representa na imagem. Então, uma estratégia

para melhor resultado seria detectar, seja *a priori* ou em tempo de execução, alguma cor que não é muito utilizada, e colocar o texto apenas na banda correspondente a ela.

Os resultados para o livro ”Metamorfose”, inserido no plano de bits 0, estão na figura 4.

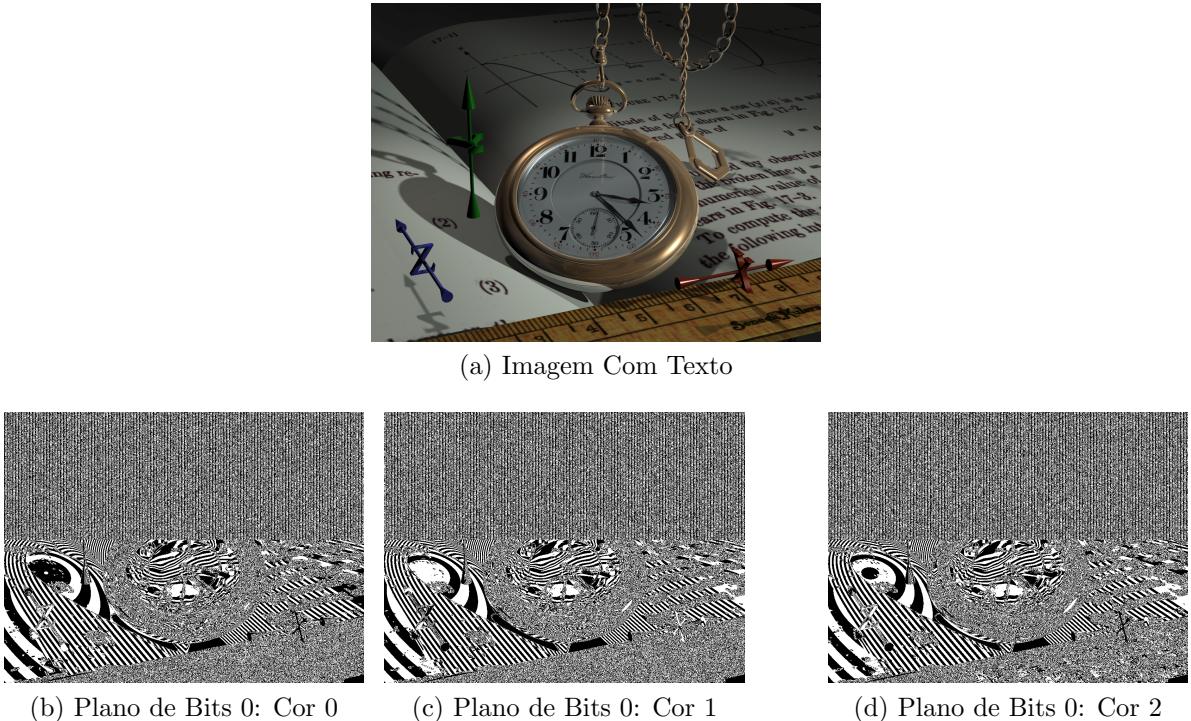


Figura 4: Imagem resultante após codificação do livro ”Metamorfose”.

O texto foi recuperado exatamente como o original.

Percebe-se pela imagem 4a que o texto não está aparente, mesmo estando na mesma imagem que o livro anterior e o texto sendo grande também. Isso se deve à escolha do plano de bits menos significativo. Nessa imagem, ainda há algum detalhamento no plano 0, mas é bem menor que o visto no plano de bits 2, então ao alterá-lo o resultado é bem menos perceptível na imagem final.

Em todos esses testes, também é perceptível o espaço ocupado pelo texto na imagem, e que ainda sobra espaço após a inserção de um *script*, que não é um texto pequeno. Isso evidencia a quantidade de informação que uma imagem é capaz de comportar, mesmo em um bit por cor por pixel.

Outros textos foram codificados em imagens e, como dito anteriormente, estão disponíveis na pasta ”Texts/”.

## 5 Conclusão

Os testes feitos foram bem-sucedidos em esconder mensagens em imagens, e depois recuperá-las sem perdas ou alterações. Assim, conclui-se que é uma técnica viável de codificação. Porém, muitos dos modos atuais de envio de imagens envolvem compressão com perdas, e nesses casos a mensagem provavelmente se perde.

Embora seja difícil perceber se há uma mensagem escondida na imagem apenas com a imagem completa, ao separar os planos de bits é fácil de perceber se há uma mensagem, e em qual deles está, pois a alteração é visível, principalmente em textos mais longos.

Se a imagem possui muitos detalhes em planos de bits menos significativos, o resultado também pode ser visto na imagem codificada completa.

Quanto maiores as dimensões da imagem e menor o texto, mais difícil é a visualização por planos de bits. Imagens de maiores dimensões também suportam textos mais longos. Uma imagem de 3840 x 2160 pixels suporta quase 25 milhões de bits, ou mais de 3 milhões de caracteres, considerando apenas um plano de bits a ser alterado. Ou seja, em uma imagem grande, é possível codificar múltiplos livros maiores que os vistos neste trabalho.